THE UNIVERSITY *of* EDINBURGH

Detecting and Preventing Unargmaxable Outputs in Bottlenecked Neural Networks

Andreas Grivas

Doctor of Philosophy
Institute for Language, Cognition and Computation
School of Informatics
The University of Edinburgh
2024

Abstract

Deep Neural Networks (DNNs) with a large number of outputs are ubiquitous for Artificial Intelligence (AI). For example, Large Language Models (LLMs) generate sentences from a vocabulary of hundreds of thousands of output tokens. Crucially, the output layer of these models typically receives as input a dense feature representation having far fewer dimensions than the output. We call such an output layer a **bottlenecked classifier**. It is known that bottlenecked classifiers reduce the expressivity of DNNs (Yang et al., 2018) and that in theory some outputs may be impossible to predict (Demeter et al., 2020), but there have been no concrete examples of this situation in the literature due to the lack of precise tools and terminology. This thesis fills this gap. We demonstrate examples where bottlenecked classifiers cause DNNs to have outputs that are impossible to predict irrespective of the input. We name such outputs **unargmaxable** and introduce tools to detect them in LLMs and multi-label classifiers. But detection can only get us so far, the impact of this thesis is in showing that we can prevent them in the presence of domain knowledge. By imposing structure on bottlenecked classifiers we guarantee that all outputs consistent with our domain knowledge are argmaxable.

Lay summary

Imagine slicing a pizza three times through its center. How many slices do you get? “How is this relevant to Artificial Intelligence (AI)?”, I hear you ask. Well, current successful AI models, like ChatGPT, work by encoding inputs, such as text and images, into a virtual space. This virtual space is continuous, but the decisions our models need to make, such as the next words ChatGPT generates, are discrete. In order to make decisions, current models slice this virtual space, like a pizza, i.e. in half through its center, and add a specific topping to half of it. For example, ChatGPT adds ham to one half, mozzarella to another and pineapple to the third half. How many slices did you get? 6? Marina is vegan, lactose intolerant and hates pineapple. Of the 6 slices, is there a slice with toppings she can eat? In this thesis, we show that current AI models do not always combine the toppings in the way we like: there are pizza slices we want that do not exist. We do so by providing tools to check the topping combinations of these high dimensional pizzas and show how to make sure that all slices with toppings that satisfy our preferences exist.

Acknowledgements

Adam, I am extremely grateful for having you as my supervisor, thank you for helping me find my own voice, for supporting me while I discovered which problems resonate with me, for channelling your passion for teaching and communicating clearly, for teaching me the value of a good example and for teaching me how to surpass the hurdle of my shitty first drafts to write a complete and well-structured paper. The past 4 years contained challenges that were much larger than one may have expected, and you have always been there for me, I deeply appreciate that.

Antonio, I am super fortunate to have you as my supervisor, you joined Edinburgh when I was short-circuiting (see what I did there?) at the most challenging time of my PhD and inspired me to push harder. Thank you for passionately insisting that: "guarantees matter!" and for devoting so much time and energy towards our discussions and papers; this thesis and this Andreas would have been much lower rank had we not met, I am extremely grateful for all your help and guidance.

Bea, thank you for exposing me to interesting problems and encouraging me and supporting me throughout my time in Edinburgh. Many problems in this thesis were itches that developed while I was working with you on Clinical NLP; I couldn't have asked for a better preparation for the PhD!

Thank you to my examiners, **Iain Murray** and **Vlad Niculae**, for the impeccable organisation of the viva, the thorough, detailed and insightful feedback on the thesis and the interesting deeper questions and discussion during the viva examination. Also, thank you to **Ivan Titov** and **Mark Steedman** for giving me helpful and constructive feedback and pointers during my annual reviews and **Bonnie Webber** for her feedback and for putting together fun MSc projects that were a pleasure to work on.

Nikolay Bogoychev, thank you for indulging with me in endless pizza and random discussions. Also, thank you for bringing the stolen probability paper to the AGORA reading group, I wonder if we would have written the softmax paper had this not happened.

Asif Khan, **Jesse Sigal**, **Eric Munday** and **Nick McKenna**, thank you for lightening up the day in the office. Asif, thanks for clearly explaining new concepts to me and for all our shared experiences that lie on the manifold of academia, papers, beer, martial arts, etc. Asif, *I still haven't started boxing. Mango?*

Sue Liu, thank you for the "relentlessly practical" quote, it rings true, methinks. Thank you for many cunning plans and fun discussions over lunch. **Clara Vania**,

thank you for mentoring me and encouraging me during my MSc project! **Sameer Bansal**, thank you for feedback and many deep discussions and advice about the PhD! **Sabine Weber**, thank you for being my lockdown accountability buddy!

Dear **AGORA** and **APRIL**, **Naomi Saphra**, **Sharon Goldwater**, **Yevgen Matusevych**, **Kate McCurdy**, **Seraphina Goldfarb-Tarrant**, **Katarzyna Pruś**, **Elizabeth Nielsen**, **Ida Szubert**, **Oli Liu**, **Coleman Haley**, **Ramon Sanabria**, **Lorenzo Loconte**, **Filippo Corponi**, **Nickil Maveli**, **Rickey Liang**, **Emile van Krieken**, **Adrián Javaloy**, **Leander Kurscheidt**, **Lena Zellinger**, thank you for all the paper suggestions, the feedback on papers and for making the PhD fun!

Dear **LTG** and **Clinical NLP**, **Claire Grover**, **Claire Llewellyn**, **Richard Tobin**, **Arlene Casey**, **Elaine Farrow**, **Daniel Duma**, **Hang Dong**, **Michael Poon**, **Emma Davidson**, **Honghan Wu**, **Heather Whalley**, **William Whiteley**, thank you for being such a kind and cool group of people to work with! **Matúš Falis**, thank you for the feedback and many deep and honest discussions.

I am also very grateful to everyone who made the PhD fun and from whom I learned a lot: **Chantriolnt-Andreas Kapourani**, **Rafael-Michael Karampatsis**, **Stephen Graham**, **Laurie Burchell**, **David Wilmot**, **Laura Perez-Beltrachini**, **Pasquale Minervini**, **Antreas Antoniou**, **Nikita Moghe**, **Boris Mitrovic**, **Leonie Bossemeyer**, **Agostina Calabrese**, **Tom Sherborne**, **Parag Jain**, **Andrea Carmantini**, **Aida Tarighat**, **Matthew Di Meglio**, **Lushi Chen**, **Rimvydas Rubavicius**, **Sander Bijl de Vroe**, **Carol Chermaz**, **Vidminas Vizgirda**, **Craig Innes**, **Henry Gouk**, **Shay Cohen**, **Aryo Gema**, **Bartosz Dzionek**, **Chang Shu**, **Arushi Goel**, **Kiyoon Kim**, **Panagiotis Efstratiadis** and **Tim Vieira**.

Thank you to **Leonie Bossemeyer**, **Leander Kurscheidt**, **Asif Khan** and **Marina Potsi** for feedback on the thesis draft!

I also want to thank everyone at the **Edinburgh Commonwealth Pool**, **Mary's Milkbar** and **Edinburgh Makerspace** for making it possible for me to maintain a reward function during the sparse reward period of thesis writing.

Thank you to my childhood friends, **Alexandros Eleftheriou**, **Emmanuel Markonis**, **Theodora-Dorita Tsourouktsoglou**, **Giannis Daskalakis**, **Charlotte Eleni Handford** and **Maria Antimisiaris**.

Penultimately, I want to thank my parents, **Chris** and **Giannis Grivas**, for being the best parents I could have asked for, thank you mum and dad! Also, my grandparents, **Stella Griva** and **Dennis Turner**, and my godmother, **Eleni Efthimiadou**.

Lastly, and most importantly, I am infinitely grateful to **Marina Potsi**, for accepting

to be the main character of my lay summary and my life, and for her unconditional love and support.

All the text in this dissertation has low-perplexity under my human natural language model.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Andreas Grivas)*

To mum and dad.
In loving memory of Bubble.

Contents

List of Figures

List of Tables

List of Listings

List of Definitions

0.1 How to Navigate this Thesis

Maybe you should demonstrate this idea with an example.

—Adam Lopez [a]

[a] Credits to Janie for this perfect example of Adam.

This thesis is expository in nature; we will give connections between disparate models via a unifying geometric lens. We will summarise the high level picture and we will provide a lot of examples. We signpost such sections as below.

A First Definition

Def. 0.1.1. This is a definition.

A First Property

Property 0.0. This is a property.

A First Proposition

Proposition 0.0. *This is a proposition: a mathematical fact that follows from a bigger result/theorem.*

A First Lemma

Lemma 0.0. *This is a lemma: a mathematical fact that has a proof and is derived to assist towards proving a theorem.*

A First Theorem

Thm. 0.0. This is a theorem: a non-trivial mathematical fact that has a proof.

Example 0: A First Example

This is an example.

A Code Example

This is a Python 3.8 code example.

A First Linear Programme (LP)

This is a LP: an optimisation problem we use to detect (un)argmaxable outputs.

Rabbit Hole: A non-trivial research project

This is an idea that seems interesting but may need a significant time investment.

Summary

At the end of key sections we will recapitulate the most important takeaways.

0.2 Notation

Fundamentals

We define $[n]$ to mean the set of numbers from 1 to n: $\{1, 2, \ldots, n\}$. $\binom{n}{k}$ is the **binomial coefficient** and represents the number of ways we can choose k elements from a set of n elements. We will work in $\mathbb{R}^n$.

Matrices and Vectors

We use capitalised boldface to represent matrices, e.g. $\mathbf{W}$, boldface for vectors, e.g. $\mathbf{w}$, and normal font, e.g. w, for scalars. We represent the **transpose** of $\mathbf{W} \in \mathbb{R}^{n \times d}$ as $\mathbf{W}^\top \in \mathbb{R}^{d \times n}$. We represent the **inverse** of $\mathbf{W} \in \mathbb{R}^{n \times n}$ as $\mathbf{W}^{-1} \in \mathbb{R}^{n \times n}$. We use $\det \mathbf{W}$ to denote the **determinant** of a matrix.

Indexing Matrices and Vectors We index vectors and matrices using subscripts, i.e. w_i is the i^{th} element of $\mathbf{w}$, $\mathbf{w}_i$ is the i^{th} row of $\mathbf{W}$ and $\mathbf{w}_{\cdot i}$ is its i^{th} column.

Common Matrices and Vectors We use $\mathbf{1}_n$ and $\mathbf{0}_n$ to represent the vector of n ones and n zeroes, respectively. When the dimension of the vector is clear from context, we may drop the subscript. $\mathbf{I}_n$ is the $n \times n$ **identity matrix**. Its rows and columns are $\mathbf{e}_i$ and $\mathbf{e}_{\cdot i}$, $i \in [n]$, respectively. These are known as the standard basis vectors in linear algebra and as one-hot vectors in Machine Learning (ML).

Mean of Matrices and Vectors We denote the **mean** of a vector, $\mathbf{x} \in \mathbb{R}^n$, as $\overline{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^{n} x_i$. For a matrix, $\mathbf{W} \in \mathbb{R}^{n \times d}$, we define the **column mean matrix**, $\overline{\mathbf{W}}_{\text{col}} = \frac{1}{n}\mathbf{1}_n\mathbf{1}_n^\top\mathbf{W}$ and the **row mean matrix**, $\overline{\mathbf{W}}_{\text{row}} = \frac{1}{d}\mathbf{W}\,\mathbf{1}_d\mathbf{1}_d^\top$.

Slicing Vectors We introduce notation to represent a slice of a vector, i.e. a subset of elements with contiguous indices. We use $\mathbf{s} = \mathbf{w}_{\text{a:b}}$ to mean $\mathbf{s} = [w_a, w_{a+1}, \ldots, w_b]^\top$. If we omit a we start at index 1, and if we omit b we end at the last index.

Indexing Subsets of Matrices and Vectors We can also select parts of vectors and matrices by choosing indices that need not be contiguous. In such a case, we assume the indices are ordered in increasing order. We define a submatrix $\mathbf{A}_{I,J}$ of $\mathbf{A}$ as the matrix formed by choosing the rows indexed by the set I and the columns indexed by the set J. In a similar vein, $\mathbf{W}_I$ denotes the submatrix formed by selecting only the rows indexed by I, and $\mathbf{W}_{:,J}$ denotes the submatrix formed by selecting only the columns indexed by J. We use $\mathbf{W}_{-i}$ as shorthand for $\mathbf{W}$ with row i dropped, i.e. $\mathbf{W}_{-i} = \mathbf{W}_I$ where $I = \{j : j \in [n], j \neq i\}$ and n is the number of rows. Correspondingly, $\mathbf{W}_{:,-i}$ is shorthand for $\mathbf{W}$ with column i dropped.

Concatenation We denote matrix concatenation using brackets. I.e. concatenating the columns of $\mathbf{A} \in \mathbb{R}^{n \times a}$ and $\mathbf{B} \in \mathbb{R}^{n \times b}$ gives us $\mathbf{C} = [\mathbf{A}\ \mathbf{B}]$, where $\mathbf{C} \in \mathbb{R}^{n \times (a+b)}$. Concatenating rows of the matrix works analogously, and we denote it as follows. For $\mathbf{A} \in \mathbb{R}^{a \times d}$ and $\mathbf{B} \in \mathbb{R}^{b \times d}$, we get $\mathbf{C} = \begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix}$, $\mathbf{C} \in \mathbb{R}^{(a+b) \times d}$.

Norms We represent the **Euclidean norm** (l2 norm) of a vector $\mathbf{w}$ as $\|\mathbf{w}\|_2 = \sqrt{\mathbf{w}^\top \mathbf{w}} = \sqrt{\sum_i w_i^2}$.

Matrix Parametrised Objects Consider a matrix $\mathbf{W} \in \mathbb{R}^{n \times d}$, $n \geq d$.[1] We proceed to interpret the matrix in the following ways.

- $\mathcal{H}(\mathbf{W}) =$ the **hyperplane arrangement** defined by $\mathbf{W}$. Each row of the matrix defines a normal vector.
- $\mathcal{A}(\mathbf{W}) =$ the **regions** formed by $\mathcal{H}(\mathbf{W})$. These are the subset of **orthants** in $\mathbb{R}^n$ that are intersected by the columnspace of $\mathbf{W}$. The regions are known as **chambers** in Hyperplane Arrangements (Aguiar et al., 2017) and **topes** in the Oriented Matroids literature (Björner et al., 1999).

[1] I apologise to mathematicians for making this a tall matrix, but in Machine Learning (ML) we write $\mathbf{y} = \mathbf{W}\mathbf{x}$ and not $\mathbf{y} = \mathbf{W}^\top \mathbf{x}$.

Sets

We denote sets using capitalised non-bold fonts, e.g. set S. We denote the **cardinality** of a set, S, using $|S|$, so for example $|[n]| = n$. We use $\binom{[n]}{k}$ to define the set of cardinality k subsets of $[n]$, i.e. $\binom{[n]}{k} = \{I \subset [n] : |I| = k\}$.

Expressions

We use the **Iverson bracket**, $[]$, to evaluate an expression to 0 or 1. For example, $y \cdot [(y \mod 2) = 1], y \in \mathbb{N}$, produces y when y is odd and zero otherwise.

Outputs

We use y to represent a single output and $\mathcal{Y}$ to represent the set of all possible outputs. We also introduce notation to unify the connection between outputs in Multi-Class Classification (MCC) and Multi-Label Classification (MLC) and their representations used when we discuss argmaxability.

- MCC
  - We use $\triangle$ to represent categories.
  - We use $\hexagon$ to represent rankings of the categories.
- MLC
  - We use $\square$ to represent subsets, i.e. label assignments.

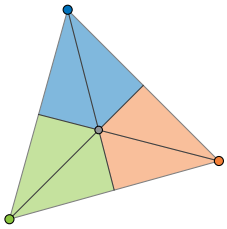
We use the notation with shapes above to help the outputs adhere semantically to their related polytopes and hyperplane arrangements.

Polytopes

We represent polytopes with shapes and use a subscript to represent their affine dimension. We introduce notation for the Simplex, $\triangle_d$, the Permutohedron, $\hexagon_d$, and the Cube, $\square_d$.

Hyperplane Arrangements

We introduce notation for the Boolean Arrangement, $\boxplus_d$, the Braid Arrangement, $\circledast_d$ and the Voronoi Tesselation, $\triangle_d$. We note that $\triangle_d$ is not technically a hyperplane arrangement, but it is closely related to $\circledast_d$, as we show in Section 2.4.3.



1 Introduction

Unargmaxable; adjective*; Impossible to rank as the best.*
This parrot was the best, it was da bomb, the cream of the crop,
the greatest thing since sliced bread, the bee's knees, the ant's pants,
the cat's pyjamas, the fox's socks, the flea's eyebrows, the dog's bollocks.
Unfortunately, it is now unargmaxable.

— Adaptation of The Dead Parrot Sketch, Monty Python.

Deep Neural Networks (DNNs) are the bee's knees. They have brought incredible advances to challenging problems that until recently seemed out of reach for practical applications: they can accurately translate text across multiple languages, caption images, convert speech to text and text to speech, generate functional code and solve problems that have an enormous search space, such as playing games like Go and predicting how proteins can be folded. Most models for the above applications have a very large number of outputs. To name a famous example, Large Language Models (LLMs) [1] such as GPT 3.5 generate text by predicting one output token at a time from a vocabulary of ≈100,000 tokens.

The key to these advances has been scale, both of the training data and the model parameters, which have grown by stacking more and more layers into DNNs. But the inherent complexity that comes with scale has shrouded our understanding and created the misconception that DNNs are always expressive enough to predict all possible outputs. In this thesis, we show that contemporary DNNs that have a large number of outputs (see Table 1.1) are bottlenecked in expressivity by their last layer. As a result, perhaps surprisingly, *DNNs have outputs that can provably never be predicted for any input to the last layer.*

[1] We use the term LLM liberally. A LM is Large (L) if it has been pretrained on a large number of tokens, e.g. more than 1 billion tokens (Rogers et al., 2024). As such, BERT and GPT qualify as LLMs.

LLM	# Input Features d	# Output Vocabulary Size n	Source
LLaMa 2	4096	32,000	Touvron et al. (2023)
OLMo	4096	50,304	Groeneveld et al. (2024)
GPT-3.5 Turbo	4096	100,276	Carlini et al. (2024) and Finlayson et al. (2024)
Command-R	8192	256,000	Huggingface

Table 1.1: We compare the number of input features (d) to the number of output tokens (n) for LLMs released circa 2024. As can be seen, $d \ll n$ across all models, i.e. the output layers of LLMs are **bottlenecked classifiers**. In fact, d was not released for GPT-3.5, but it was reverse-engineered by Carlini et al. (2024) and Finlayson et al. (2024) by exploiting the constraints of bottlenecked classifiers.

Promise of this thesis By reading this thesis you will understand why a DNN output layer that has more outputs than inputs (see Fig. 1.1) limits the expressivity of the whole DNN. We call such an output layer a **bottlenecked classifier** (Yang et al., 2018) for reasons we will explain in Section 3.3. A bottlenecked classifier introduces geometric constraints that make some outputs impossible to predict (Cover, 1965; Cover, 1967; Demeter et al., 2020). This thesis will equip you with tools to detect such problematic outputs and methods to prevent them.

Motivation While less expressive than fully-parametrised classifiers, bottlenecked classifiers are ubiquitous as they provide an attractive trade-off between expressivity and efficiency. As such, they are used in all recent LLMs (see Table 1.1) and in many multi-label classifiers used for document classification. *But what if I told you that your gazillion parameter DNN classifier can never predict a given output? What if your DNN needs to predict that output when deployed in the real world?* [2] *Would this worry you?*

[2] In our experiments we show that there exist test set examples that can never be predicted.

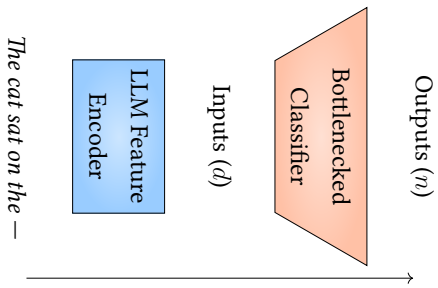


Figure 1.1: Example LLM architecture. LLMs have a Bottlenecked Output Layer ($d < n$).

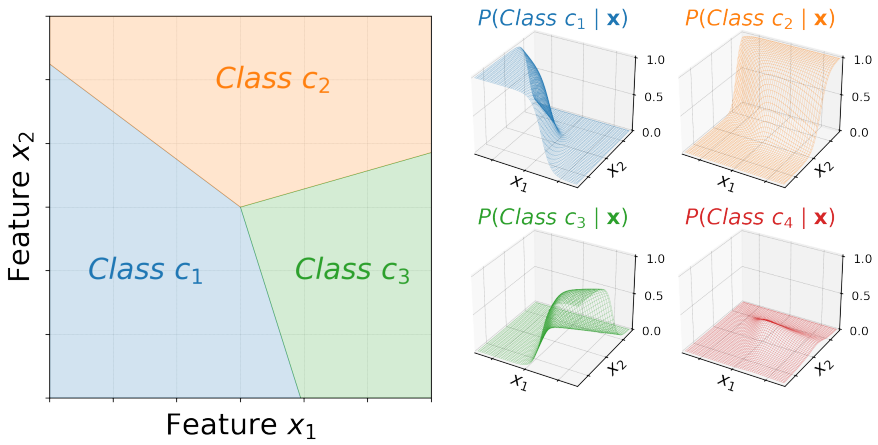


Figure 1.2: **Problem: Unargmaxable categories**. A multi-class classifier with 4 outputs, c_1, c_2, c_3, and c_4, and 2 inputs x_1, x_2. On the right, we plot the probability of each category for all $x_1, x_2 \in \mathbb{R}$. On the left, we color each input according to the category with the largest probability. While c_1, c_2 and c_3 surface as regions, c_4 does not. Category c_4 is **unargmaxable**: there is no input in feature space for which c_4 has the largest probability. We will use to recall this problem/plot. We will revisit it in the next two chapters as we add more nuance and uncover when and why unargmaxable categories arise (Figs. 2.6, 2.16 and 3.12).

1.1 Problem: Unargmaxable Outputs

Multi-Class Classification (MCC)

We illustrate such a problematic case in Fig. 1.2. We consider a multi-class classifier where each input is assigned a single **category** from a vocabulary. For example, consider a classifier that takes as input a news article and classifies it into one of the categories : c_1=health, c_2=sport, c_3=entertainment and c_4=politics. We encode the news article using a DNN text encoder, such as BERT (Devlin et al., 2019), and keep the two most salient features, x_1 and x_2.

As can be seen in Fig. 1.2, we have a bottlenecked output layer, since we have 4 output categories but only 2 inputs. On the right, we see the probability assigned to each category for any point $(x_1, x_2) \in \mathbb{R}^2$ in the input space. On the left, we see the input space of the classifier coloured according to the category that is assigned the largest probability. We see a consequence of the bottleneck: there is no position in the input space for which category c_4 is assigned the largest probability. We call such outputs that cannot be predicted for any input **unargmaxable**. As we will see in Section 3.5, unargmaxable categories *may* arise in classifiers that have a large number of categories, but this can be avoided, as we will explain in Chapter 5.

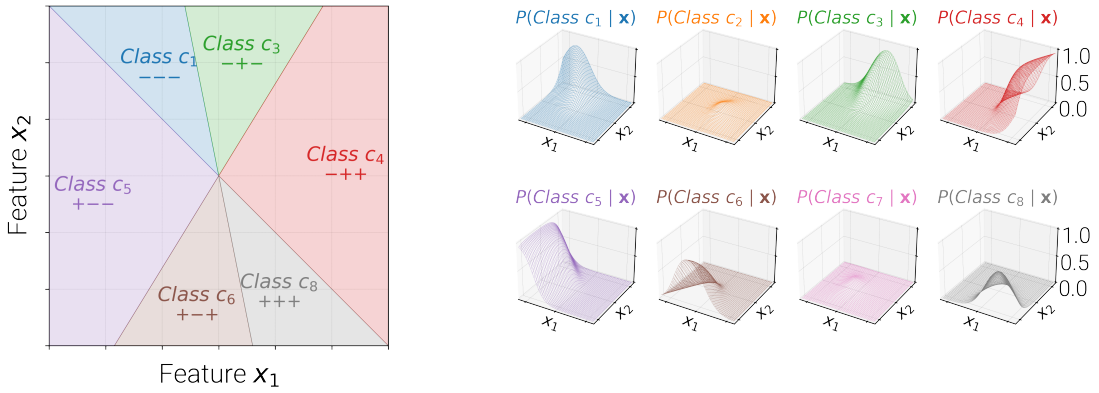


Figure 1.3: **Problem: Unargmaxable label assignments**. A multi-label classifier with 8 outputs and 2 inputs x_1, x_2. As before, on the right we plot the probability of each output for all $x_1, x_2 \in \mathbb{R}$. On the left, we color the inputs according to which of the 8 label assignments is assigned the largest probability. As can be seen, label assignments c_2 and c_7 are **unargmaxable**. Crucially, for this bottlenecked classifier it is impossible to find parameters that make all 8 label assignments argmaxable, 2 of them must be unargmaxable. We will gradually understand why this must be the case as we revisit this plot in (Figs. 2.7, 2.15 and 3.11).

Multi-Label Classification (MLC)

On the other hand, some modelling choices make unargmaxable outputs unavoidable. Consider a multi-label classifier which assigns to each input multiple labels from a large label vocabulary of size n. For example, let us downscale and adapt our previous text classification example. Suppose the multi-label classifier takes as input a news article, but instead of assigning a single category, it labels it with any subset of $n = 3$ labels: l_1=health, l_2=sport, l_3=entertainment. We represent the outputs of the classifier using n binary labels: for each label we use a $+$ if the label is assigned and a $-$ otherwise. For example, a news article could be about both health and sport, but not entertainment. We represent such an output as $l_1 = +$, $l_2 = +$ and $l_3 = -$, or $++-$ for short. We call each such complete specification of labels a **label assignment**. In general, there are 2^n possible label assignments in total, i.e. $\{+, -\}^n$, so 8 label assignments for our example.

In this scenario, if we construct each binary classifier to have fewer than n input features, unargmaxable outputs *must* exist (see Section 3.5). By must exist, we mean that irrespective of which parameters our neural network converges to during training, a subset of label assignments will be unargmaxable. We illustrate this in the example of Fig. 1.3, where we can see that 2 out of 8 label assignments are unargmaxable.

We encourage the reader to verify that irrespective of how we draw the 3 straight lines through the origin to separate out the label assignments, we can only form 6 regions; there will always be 2 unargmaxable label assignments. We invite the sceptical reader to convince themselves by dragging the arrows on the left side of the following interactive visualisation: https://viz.unargmaxable.ai/duality/.

1.1.1 Practical Repercussions

Until now, we have discussed the problem of unargmaxability on toy examples to assist exposition. But unargmaxability is not a toy problem: we now highlight the impact of unargmaxability on applications with two examples from real models/datasets.

Unargmaxable Tokens in Machine Translation (MT)

We begin with an example that arises in publicly available models (Tiedemann, 2020) that translate between languages (Section 4.2). In this example, the categories are the vocabulary tokens of the target language. Such Language Models (LMs) have many more outputs than inputs, e.g. the number of vocabulary tokens is in the tens of thousands but the number of inputs to the last layer are in the hundreds. As such, some output vocabulary tokens may be impossible to predict. In Chapter 4, we demonstrate that this is the case with some released MT models. There is a Bulgarian to English MT model for which the subword *erecti* is unargmaxable and an English to Russian model for which the subword *Предварительны* is unargmaxable. The subword can be inflected to form *Предварительный*, which means "preliminary".

While having unargmaxable tokens may not drastically impact model performance, their presence can be a big problem. Firstly, while LLMs can paraphrase sentences or use an alternative tokenisation to sidestep unargmaxable tokens, there are cases where this may not as easy, e.g. consider the case of names and identifiers. Secondly, unargmaxable tokens in LLMs are an open door for adversarial attacks (Aghakhani et al., 2021; Zhu et al., 2019). In fact, the tokens need not even be unargmaxable for us to have safety concerns, it suffices for the tokens to be undertrained (Geiping et al., 2024). It has been shown that inputting under-trained tokens can make LLMs behave in undesirable ways and their output can deteriorate to gibberish at best and to offensive or harmful at worst (Geiping et al., 2024; Rumbelow et al., 2023). Thirdly, the presence of unargmaxable tokens means we are wasting parameters and compute on tokens that can rarely be predicted (Land et al., 2024).

Next, we turn to multi-label classification where having unargmaxable outputs can be even more problematic. This is because in addition to the above considerations, unargmaxable outputs directly impact model generalisation: we will show that models can have unargmaxable test set examples.

Unargmaxable Label Assignments in Multi-Label Classification

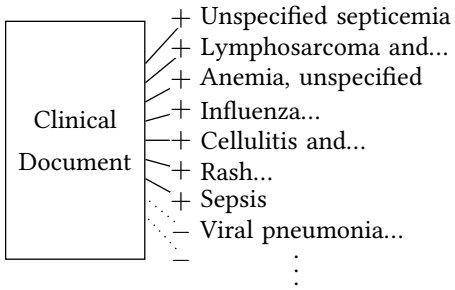


Figure 1.4: The above label assignment is **unargmaxable**; it can never be assigned to any input document for this model.

We now turn to a model which assigns each document multiple labels from a large label vocabulary (Section 4.1). More specifically, we consider a model that tags clinical documents with findings and diagnoses. As such, unargmaxable outputs are a safety concern, since a model may misdiagnose a patient because a given combination of findings is unargmaxable. If this model is used for decision making, such mistakes can directly affect the health of patients. To highlight this problem in a real world scenario, we trained a model on a clinical dataset and analysed its parameters. In Fig. 1.4, we illustrate a set of labels from the test set of the dataset that cannot be assigned to any clinical document by this model. This is a fundamental drawback for such a safety-critical domain.

1.1.2 So What?

Unargmaxable outputs can impact our models in multiple ways.

1. **Generalisation**. Unargmaxable outputs hinder the generalisation ability of our model. While an output may be improbable at training time, there is no guarantee that we will not observe it at test time. Models with unargmaxable outputs cannot be robust to distribution shifts which make some rare/unseen outputs more frequent.

2. **Fairness**. Unargmaxable outputs directly impact model fairness, since certain model outputs, further from being underrepresented, may not be represented at all. As a result, underrepresented groups may find no representation in the

outputs of such models (Menon et al., 2021a). Moreover, harmful biases in our data may be propagated and further exacerbated by our model (Hooker, 2021).

3. **Safety**. Unargmaxable outputs are a weakness that leave our models vulnerable to adversarial attacks. One can even create adversarial attacks that specifically exploit unargmaxability (Aghakhani et al., 2021; Zhu et al., 2019).

4. **Trustworthiness**. Choosing a bottlenecked output layer is an architectural modelling choice that can lead to unargmaxable outputs. This is counter-intuitive and we assume the majority of ML practitioners is not aware of this problem. As such, we believe that architectural modelling choices should be explicitly mentioned as assumptions [3] so that they can be taken into account. Without knowledge of the assumption, unargmaxability can be an unintended consequence. Since unargmaxable outputs are surprising both to end users and to ML practitioners, this can lead to loss of trust in systems.

Why have we not noticed unargmaxable outputs? While bottlenecked classifiers have been shown to be less expressive (Yang et al., 2018), and it is theoretically known that this loss in expressivity can make some outputs impossible to predict (Cover, 1965; Cover, 1967; Demeter et al., 2020), as far as we are aware, we are the first to unambiguously demonstrate the existence of unargmaxable outputs in practice. But models with unargmaxable outputs can be severely limited, *why have we not noticed this empirically*?

First of all, since we mostly train our models using maximum likelihood estimation, it is unlikely that the most frequent outputs would be unargmaxable. Moreover, as we will see in our experiments, unargmaxable outputs are more likely to exist when we shrink the dimensionality of the last layer (e.g. when it is in the tens or low hundreds), which is rarely the case anymore. Lastly, when unargmaxable outputs exist, we will not easily notice their existence. This is because when training DNNs, we have a methodological blindspot: we treat the dimensionality of the layers as a hyperparameter to be chosen by maximising a target metric on a held out set of examples. Under such treatment, we will tend to increase the dimensionality until we get diminishing returns. If a large percentage of needed outputs are unargmaxable, we will notice this in our metrics and scale the model since this will improve our performance. On the other hand, if a smaller percentage of outputs is unargmaxable,

[3] For example, similar to how conditional independence assumptions are usually mentioned.

these can go undetected by commonly used metrics, since these are often insensitive to long tail phenomena. For example, for F1 score in MLC, it would be hard to notice unargmaxable label assignments, since F1 would give credit for partially correct assignments. We would need to use exact match as a metric to notice. [4]

Can scale solve argmaxability? Scaling up the dimensionality of the last layer may help for engineering purposes, but not for the purpose of this thesis. The reason we reject scaling as an approach is because:

- We want to understand the problem; scaling the model just sweeps the problem under the rug of high dimensions.[5]
- While scaling the model may make unargmaxable outputs less likely, we obtain no guarantees that we have solved the problem for training examples, let alone for test set examples. Although we could verify our model for some examples of interest using the tools we shall develop in Chapter 4, there are too many outputs to verify all of them. Moreover, if we provide no guarantees, our model is left vulnerable to adversarial attacks (Aghakhani et al., 2021; Zhu et al., 2019).
- Even if scaling the model solved unargmaxability, scaling is not always possible. We may need to compress the model to run it on limited hardware. Alternatively, the set of outputs may be so large that we are forced to use a small number of input features due to computational constraints.
- Most importantly, we reject scaling because in this thesis we will provide a better solution. As alluded to earlier, in Chapter 5 we will show how we can solve this problem with guarantees in a way that is also amenable to scaling.

1.1.3 Scope: Bottlenecked Output Layers

We have introduced the problem of unargmaxability and seen practical issues that can arise as a consequence. We now more precisely define which output layers we will focus on and where these are used.

[4] Presumably, unargmaxability would be an even more serious issue in the early 2000s, when Maxent classifiers and feature engineering were common. However, as we alluded to in the introduction, it is mostly with deep learning that models have become accurate enough for us to focus on such limitations. When models were less expressive there were more pressing issues to be solved.

[5] OpenAI's core values at the time of writing (early 2024) mention: "When in doubt, scale it up".

Our unargmaxability results hold for DNNs that have linear output layers. For such models, we can think of the model as consisting of two parts; a) a complex non-linear feature encoder, followed by b) a **linear classifier**.

Bottlenecked Classifier

Def. 1.1.1. We say that a neural network output layer is a **bottlenecked classifier** if it is a **linear classifier** that has **more outputs than inputs**.

We use $\mathbf{W}$ to refer to a bottlenecked classifier (Section 3.3.1). Bottlenecked classifiers are ubiquitous in DNNs. They are the go-to output layer for many models, such as LLMs, MT models and more generally, models for problems with structured outputs, e.g. clinical coding systems that tag documents with multiple findings.

1.2 Claim of this Thesis

Claim of this thesis

Bottlenecked classifiers may have unargmaxable outputs, but such outputs can be detected and prevented.

1.3 Outline and Contributions

In Chapter 2, we introduce the main characters of our thesis: i) the outputs of the classifiers we discussed in this chapter, and ii) the matrix $\mathbf{W}$. We ask: **Which outputs are argmaxable/unargmaxable for a given $\mathbf{W}$?** We build the answer gradually by introducing geometric representations for the outputs. We show that $\mathbf{W}$ constrains these representations, making some outputs unargmaxable, and illustrate that in some scenarios there *may* be unargmaxable outputs while in others there *must* be unargmaxable outputs. To answer our question, we distil the output representations to discrete objects called sign vectors, and answer the question of unargmaxability generally, but abstractly, via the framework of Oriented Matroids (Björner et al., 1999).

In Chapter 3, we ground what we learned to the case of DNNs, i.e. we explain how $\mathbf{W}$ arises as an architectural choice in the output layers of neural networks (Section 3.3). We introduce the term "unargmaxability" to precisely define this phenomenon and revisit the question of which outputs may/must be unargmaxable for MLC and MCC

in this more concrete setting. We then explain how unargmaxability arises as a consequence of the reduced expressivity of bottlenecked classifiers.

In Chapter 4, we ask: **do unargmaxable outputs exist for models used in practice?** In order to answer this question, we introduce algorithms for detecting unargmaxable outputs for MCC and MLC. We then use these to show that unargmaxable outputs do indeed exist; we show that multi-label classifiers trained with a narrow bottleneck have unargmaxable test set examples (Section 4.1) and find tokens that can never be predicted in released MT models (Section 4.2).

In Chapter 5, we ask: can we prevent the unargmaxable outputs we found in Chapter 4? That is, **can we parametrise output layers such that all outputs of interest are argmaxable by construction?** We answer affirmatively for the case of sparse outputs in MLC, by showing how to reparametrise the output layer such that all such outputs are argmaxable (Section 5.1). We then move on to MCC and show how to guarantee that all top-k rankings of categories are argmaxable (Section 5.2). Lastly, we discuss a rule of thumb for choosing the number of features of an output layer.

Finally, in Chapter 6, we conclude that unargmaxability is an understudied problem that can impact DNNs in practice. Bottlenecked classifiers are intrinsically unreliable and can hinder the generalisation of our DNNs, but this need not be the case. We can prevent unargmaxable outputs by imposing structure on the parameters of bottlenecked classifiers: we can align unargmaxability constraints with known constraints in our datasets. However, we highlight a pressing limitation that arises from our research: the problem of hyperplane overcrowding. We conclude the thesis by elaborating on this limitation and discussing future directions.

Summary of Publications This thesis expands on the following two papers:

- (Paper 1) "Low-Rank Softmax Can Have Unargmaxable Classes in Theory but Rarely in Practice" (ACL 2022).
- (Paper 2) "Taming the Sigmoid Bottleneck: Provably Argmaxable Sparse Multi-Label Classification" (AAAI 2024).

We reframe the content of both papers in terms of detecting unargmaxable outputs and preventing them (see Table 1.2). To achieve this, we include novel background material (Chapter 2 and Sections 3.4 and 3.5) which unifies both papers from a broader perspective. We also include novel unpublished material, i.e. we show how to guarantee that all k-top rankings of categories in MCC are argmaxable (bottom right of Table 1.2).

Main Chapters of the Thesis		
	Unargmaxable Outputs	
	Detect	Prevent
Multi-Label Classification	Section 4.1 (Paper 1)	Section 5.1 (Paper 2)
Multi-Class Classification	Section 4.2 (Paper 2)	Section 5.2 (New)

Table 1.2: We summarise the contributions of the main chapters of the thesis and how that aligns with the published papers the thesis is based on. In Chapter 4 we expand on the first column, i.e. we show how to detect unargmaxable outputs and we empirically find unargmaxable outputs in MLC and MCC models. In Chapter 5, we expand on the second column and show how to provide guarantees that all target outputs are argmaxable, subject to a sparsity assumption on the outputs of MLC and a top-k ranking assumption on the outputs of MCC.

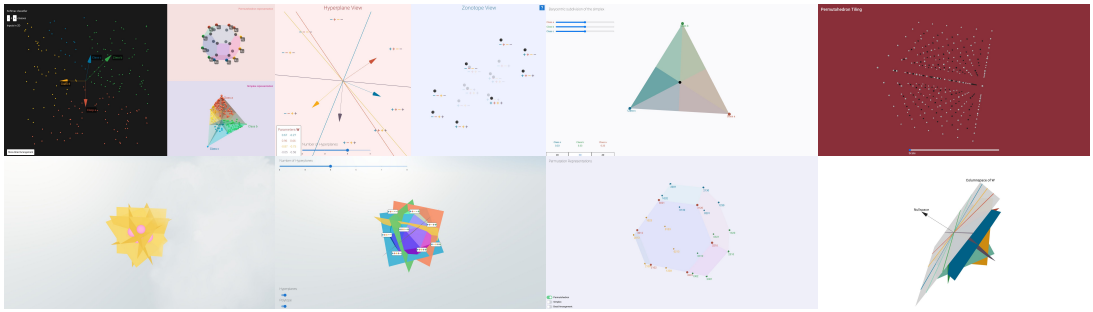
Contributions

Code In this thesis we introduced code and tools to detect unargmaxable outputs for MLC and MCC. We have made these publicly available:

- https://github.com/andreasgrv/unargmaxable
- https://github.com/andreasgrv/sigmoid-bottleneck

Our tools have been used by Brody et al. (2023) to check for "unselectable" keys in attention mechanisms.

Interactive Visualisations As an additional contribution, we have endeavoured to make the thesis accessible by providing interactive visualisations of the problem and main concepts of the thesis. We tabulate the links to the visualisations in Fig. 1.5.



Decision Boundaries of a Softmax Bottleneck
Chebyshev Balls
Hyperplane Arrangement/Zonotope Duality
Alternating Polytope
Barycentric Subdivision of the Simplex
Permutation Representations
Permutohedron Tiling of 3D Space
Slicing the Braid Arrangement

Figure 1.5: Interactive visualisations that demonstrate the problem of unargmaxable outputs and introduce the concepts we need in this thesis in a an accessible way.

We close this chapter by restating the main contributions of the thesis.

Summary of Contributions

- We define (un)argmaxability and answer which outputs are (un)argmaxable for a bottlenecked classifier via the theory of Oriented Matroids.
- We introduce tools to unambiguously detect unargmaxable outputs in MLC and MCC and demonstrate that unargmaxable outputs exist for models used in practice.
- We demonstrate via experiments that we can prevent unargmaxable outputs in MLC under minor sparsity assumptions on the outputs and show how to guarantee that all top-k rankings are argmaxable for MCC.
- We introduce a rule of thumb for choosing the number of feature dimensions d for bottlenecked classifiers: if we want to predict all subsets of $r \leq k$ labels in MLC or all top-k rankings of categories in MCC, we should choose $d > 2k + 1$.
- Lastly, we identify the limitation of hyperplane overcrowding which is an important challenge to overcome if we want to compress linear output layers of DNNs.

2 Argmaxability

If you are not too ambitious, it can be a pleasure to realize that you have discovered something previously known, because at least you know you were on the right track.

—I. M. Gel'fand

In Chapter 1, we witnessed a crime: we saw examples of MCC and MLC that had outputs that were unargmaxable. We alluded to the culprit: a bottlenecked classifier, which we represent by a matrix $\mathbf{W}$ (see Section 3.3). However, we have not yet discussed how or why the crime occurred. In this section, we begin to unravel the crime by sketching a more nuanced picture of its main characters:

- The potential victims: the outputs of the classifiers we consider.
- The culprit: the matrix $\mathbf{W}$ that restricts the outputs, making some unargmaxable.

We ask: **Which outputs are argmaxable/unargmaxable for a given $\mathbf{W}$?** The answer is: the covectors/vectors of the Oriented Matroid associated with $\mathbf{W}$. Another crime has just been committed; the author threw jargon at you without introducing it. But this was not out of disrespect, it was to illustrate that most of this mathematical language is quite opaque and means very little without the right representations. Our goal throughout the chapter is to build our notation, jargon, representations and intuition such that this result makes sense when we next encounter it at the end of this chapter. Then, when we have connected argmaxability to properties of $\mathbf{W}$, we will connect $\mathbf{W}$ to neural networks in the next chapter.

This chapter gives an accessible exposition to the mathematical concepts introduced in Björner et al. (1999), Stanley (2004), and Ziegler (1995). We need these to understand the problem of unargmaxable outputs. Below are some further pointers if you want to read about these subjects in more detail.

- In this thesis we interpret the parameters of a linear classifier as vertices of a polytope, P. See Ziegler (1995, Chapter 0) for examples of general polytopes which are relevant when discussing MCC and Ziegler (1995, Chapter 7.3) for zonotopes, which are relevant for MLC. If we partition feature space according to which outputs are the argmax of the linear classifier, we obtain the normal fan of P (Ziegler, 1995, Chapter 7).

- The regions of the normal fan we are interested in are intersections of halfspaces. As such, they are regions of hyperplane arrangements. For more details on hyperplane arrangements, see Stanley (2004) and Federico Ardila's excellent online lectures.

- The properties we need for argmaxability can be abstracted away to properties of sign vectors. Such structures on sign vectors are called oriented matroids, see Ziegler (1995, Chapter 6) and Björner et al. (1999).

2.1 Output Representations

The examples of outputs that are unargmaxable for MCC and MLC differ. In this section, we sketch a rudimentary picture of them and their differences. As we advance our story further in this chapter, we add nuance to our main characters by introducing more representations for them. Outputs arise as vertices of polytopes (in Section 2.3) and as regions of hyperplane arrangements (in Section 2.4). Once we understand these representations, we distil them into a more abstract representation: sign vectors (in Section 2.5.1) —which is the essential representation for understanding argmaxability. Since we work with probabilistic classifiers, we will also explain how our representations are the support of probability distributions (in Section 2.6). After explaining these representations, we will be ready to move on to Chapter 3 and see when unargmaxable outputs arise in bottlenecked classifiers.

While outputs can be a very broad set of objects, for the purposes of this thesis we will only need three particular outputs: **Categories**, **Rankings** and **Subsets**.

2.1.1 Categories △

In MCC the outputs are **categories**. Since we will focus on LLMs and MT, the instances we will often be working with will be vocabulary tokens of a LLM or a MT model. We use △ to represent categories to foreshadow the connection of categories to the vertices of the simplex △ (Definition 2.3.1) and regions of the Voronoi Tesselation ⧋ (Definition 2.4.9).

2.1.2 Rankings ⬡

In MCC, we can also think of **rankings** of the categories. Imagine ordering the tokens of a LLM according to the probability assigned to each token by the model. Geometric representations for rankings are the vertices of the permutohedron ⬡ (Definition 2.3.2) and the regions of the Braid Arrangement ⯐ (Definition 2.4.8).

For this thesis, rankings are synonymous with permutations. A **permutation**, $\boldsymbol{\sigma}$, defined on a set of n elements is an ordering of the elements and is defined as a bijection from $[n]$ to $[n]$. Namely, $\boldsymbol{\sigma}$ is a function $\sigma(i) = j, \quad i, j \in [n]$ that is one to one and onto. We define S_n to be the set of all possible permutations of n elements. $S_n = \{\boldsymbol{\sigma} : \{1, 2, 3, \ldots, n\} \leftrightarrow \{1, 2, 3, \ldots, n\}\}$.

2.1.3 Subsets □

In MLC the outputs are **subsets**, i.e. the subset of labels from the label vocabulary that we assign to the input. We call each subset of assigned labels a **label assignment**. Geometric representations for subsets are the vertices of the cube □ (Definition 2.3.3) and the regions of the Boolean Hyperplane arrangement ⊞ (Definition 2.4.7).

We now switch gears for the next two sections to introduce prerequisites for the more nuanced representations of outputs that follow: polytopes (Section 2.3) and hyperplane arrangements (Section 2.4).

2.2 Linear Dependencies

Waitress: (...) You can't have egg bacon span and sausage without the span.

Wife (shrieks): I don't like span!

Vikings (singing): Span span span span. Lovely span! Wonderful span!

—Adaptation of the Spam Sketch, Monty Python

In the next two sections we develop concepts we need in order to introduce the more nuanced realisations of our main characters and their connection to argmaxability. We will discuss the outputs that can be produced by $\mathbf{W}$ from two perspectives:

- Hulls and spans: the outputs that $\mathbf{W}$ can express.
- Linear dependencies: the constraints which limit the outputs $\mathbf{W}$ can express.

2.2.1 Spans and Hulls

We begin by building spaces of outputs by defining operations on collections of vectors. We assume the reader is familiar with the notions of vector spaces and subspaces, see Strang (2016, Section 3.1). We intend any operation, f, we define for a collection of vectors to hold equivalently for matrices; i.e. we interpret the matrix as the collection of column vectors in the matrix. E.g. $f(\mathbf{V}) = f(\mathbf{v}_{\cdot 1}, \mathbf{v}_{\cdot 2}, \ldots, \mathbf{v}_{\cdot d})$, where $\mathbf{V} \in \mathbb{R}^{n \times d}$ and $\mathbf{v}_{\cdot i} \in \mathbb{R}^n$ is a column vector. We will sometimes need to apply operations on the row vectors, in which case we use $f(\mathbf{V}^\top) = f(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_d)$.

When we define the operation with two arguments, we mean the operation applied using the specific second argument. When we omit the second argument, we mean that we apply the operation to all valid second arguments to construct a space, also known as the hull. We will introduce the following pairs of operations / hulls:

- Linear combinations $\rightarrow$ linear hulls (also known as subspaces),
- Affine combinations $\rightarrow$ affine hulls (also known as affine subspaces),
- Convex combinations $\rightarrow$ convex hulls (also known as (bounded) Polytopes)

As we will see, Affine and Convex combinations differ from Linear combinations by applying additional constraints to the coefficients.

Linear Combination

Linear Combination We define a linear combination of a set of vectors $\mathbf{v}_{\cdot 1}, \mathbf{v}_{\cdot 2}, \ldots, \mathbf{v}_{\cdot d}$ using coefficients $\mathbf{c} \in \mathbb{R}^d$ as:

$$\text{linear}\,(\mathbf{V}, \mathbf{c}) = \sum_{i=1}^{d} c_i \mathbf{v}_{\cdot i}, \quad c_i \in \mathbb{R} \tag{2.1}$$

Linear Hull Given a set of vectors in $\mathbb{R}^d$, we can express any vector in the subspace as the sum of the $\mathbf{v}_{\cdot i}$ scaled by real valued coefficients c_i. We obtain the **linear hull** of $\mathbf{V} \in \mathbb{R}^{n \times d}$:

$$\text{linear}(\mathbf{V}) = \sum_{i=1}^{d} c_i \mathbf{v}_{\cdot i}, \quad \forall c_i \in \mathbb{R} \tag{2.2}$$

$$= \mathbf{V}\mathbf{c}, \quad \forall \mathbf{c} \in \mathbb{R}^d \tag{2.3}$$

Intuitively, $0, 1, 2, \ldots, (d-1)$ dimensional subspaces in $\mathbb{R}^d$ are the origin, lines through the origin, planes through the origin, ..., and hyperplanes through the origin.

Affine Combination

Affine Combination We define an affine combination of a set of vectors $\mathbf{v}_{\cdot 1}, \mathbf{v}_{\cdot 2}, \ldots, \mathbf{v}_{\cdot d}$ using coefficients $\mathbf{c} \in \mathbb{R}^d : \sum_{i=1}^{d} c_i = 1$ as:

$$\text{affine}(\mathbf{V}, \mathbf{c}) = \sum_{i=1}^{d} c_i \mathbf{v}_{\cdot i} \quad c_i \in \mathbb{R} : \sum_{i=1}^{d} c_i = 1 \tag{2.4}$$

Affine Combination = Linear Combination with Vector Offset We note that in neural network layers it is common to think of an affine layer as a linear layer with a bias, i.e. a vector offset. To be consistent with the literature, we will therefore write the affine hull in this form. Below we show how to rewrite an affine combination of d vectors as a linear combination of $d-1$ vectors plus a vector offset.

Example 1: Affine = Linear with Vector Offset

Note that due to the constraint $\sum_{i=1}^{d} c_i = 1$, we can eliminate c_d by writing it as $c_d = 1 - \sum_{i=1}^{d-1} c_i$. By doing so, we can rewrite an affine form as a linear form plus a vector offset $\mathbf{v}_{\cdot d}$. So we have:

$$\text{affine}(\mathbf{V}, \mathbf{c}) = \sum_{i=1}^{d} c_i \mathbf{v}_{\cdot i}, \quad c_i \in \mathbb{R} : \sum_{i=1}^{d} c_i = 1 \tag{2.5}$$

$$= \sum_{i=1}^{d-1} c_i \mathbf{v}_{\cdot i} + c_d \mathbf{v}_{\cdot d}, \quad c_i \in \mathbb{R} : \sum_{i=1}^{d} c_i = 1 \tag{2.6}$$

$$= \sum_{i=1}^{d-1} c_i \mathbf{v}_{\cdot i} + \left(1 - \sum_{i=1}^{d-1} c_i\right) \mathbf{v}_{\cdot d}, \quad c_i \in \mathbb{R} \tag{2.7}$$

$$= \underbrace{\sum_{i=1}^{d-1} c_i \left(\mathbf{v}_{\cdot i} - \mathbf{v}_{\cdot d}\right)}_{\text{linear}} + \underbrace{\mathbf{v}_{\cdot d}}_{\text{offset}}, \quad c_i \in \mathbb{R} \tag{2.8}$$

Affine Hull An affine subspace is a linear subspace that is offset by a vector so that it does not go through the origin. We construct the **affine hull** of $\mathbf{V} \in \mathbb{R}^{n \times d}$, where $\mathbf{V}' = \mathbf{V}_{:,-d}$:

$$\text{affine}(\mathbf{V}) = \sum_{i=1}^{d-1} c_i \mathbf{v}_{\cdot i} + \mathbf{v}_{\cdot d}, \quad \forall c_i \in \mathbb{R} \tag{2.9}$$

$$= \mathbf{V}'\mathbf{c} + \mathbf{v}_{\cdot d}, \quad \forall \mathbf{c} \in \mathbb{R}^{d-1} \tag{2.10}$$

Intuitively, $0, 1, 2, \ldots, (d-1)$ dimensional affine subspaces in $\mathbb{R}^d$ are points, lines, planes ... and hyperplanes, respectively.

Convex Combination

Convex Combination We define a convex combination of a set of vectors $\mathbf{v}_{\cdot 1}, \mathbf{v}_{\cdot 2}, \ldots, \mathbf{v}_{\cdot d}$ using coefficients $\mathbf{c} \in \mathbb{R}^d : c_i \geq 0, \sum_{i=1}^d c_i = 1$ as:

$$\text{conv}(\mathbf{V}, \mathbf{c}) = \sum_{i=1}^{d} c_i \mathbf{v}_{\cdot i} \quad c_i \in \mathbb{R} : \sum_{i=1}^{d} c_i = 1, \quad c_i \geq 0 \tag{2.11}$$

Convex Hull We use convex combinations to build polytopes (Section 2.3). We define a polytope in terms of its extreme points. These extreme points form a boundary that encloses all other points. We capture this intuition formally by defining a polytope as the convex hull of its vertices.

Convex Hull

Def. 2.2.1. The **convex hull** of $\mathbf{V} \in \mathbb{R}^{n \times d}$ is:

$$\text{conv}(\mathbf{V}) = \sum_{i=1}^{d} c_i \mathbf{v}_{\cdot i}, \quad \forall c_i : \sum_{i=1}^{d} c_i = 1, \quad c_i \geq 0 \tag{2.12}$$

It will be important to differentiate between points that are **on the convex hull**, i.e. on the boundary, versus points that are internal to the convex hull, which we say are in the **relative interior**.

Relative Interior

Def. 2.2.2. The **relative interior** of a convex set, $C \subset \mathbb{R}^d$, is:

$$\text{relint}(C) = \{\mathbf{x} \in C : \forall \mathbf{y} \in C, \mathbf{y} \neq \mathbf{x} \quad \exists \mathbf{z} \in C : \mathbf{x} \in (\mathbf{y}, \mathbf{z})\} \tag{2.13}$$

where $(\mathbf{y}, \mathbf{z})$ is the line segment from $\mathbf{y}$ to $\mathbf{z}$ excluding the endpoints.

We illustrate everything we have introduced so far in Fig. 2.1 for $\mathbf{V} = \mathbf{I}_3$, where $\mathbf{e}_i$ is the unit vector for dimension i.

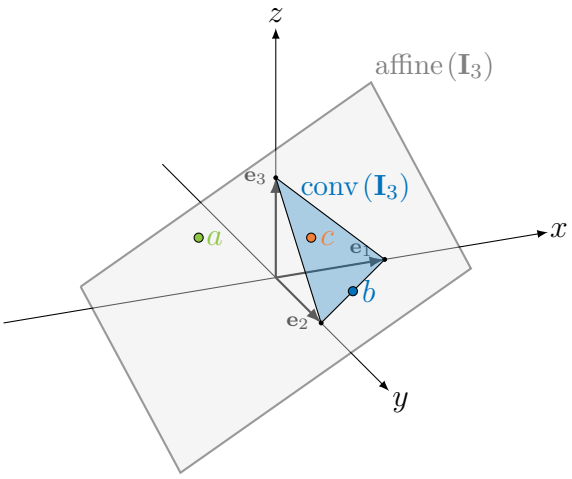


Figure 2.1: Illustration of all hulls we introduced for $\mathbf{V} = \mathbf{I}_3$ in $\mathbb{R}^3$. The linear hull, linear $(\mathbf{I}_3)$, is all of $\mathbb{R}^3$. The affine hull, affine $(\mathbf{I}_3)$, is the 2D plane (truncated for visualisation). The convex hull, conv $(\mathbf{I}_3)$, is the triangle △. Point a is on the affine hull. Point b is on the convex hull. Point c in the relative interior of the convex hull.

2.2.2 Linear Dependence, Spans and Rank

In Section 3.2 we will ask: how expressive is a bottlenecked classifier $\mathbf{W}$? In order to answer this we need concepts from linear algebra, such as linear dependence, and the span and rank of a matrix. As we will see, the culprit behind $\mathbf{W}$ are geometric constraints that arise from projecting a high dimensional space to a low dimensional one or vice-versa. These constraints have the form of linear dependencies that are introduced between the inputs and outputs and restrict which outputs $\mathbf{W}$ can express.

Linear Dependence

Def. 2.2.3. A set of d vectors $\mathbf{w}_{\cdot 1}, \mathbf{w}_{\cdot 2}, \ldots, \mathbf{w}_{\cdot d}$ is **linearly dependent** iff

$$\exists \mathbf{c} \in \mathbb{R}^d, \quad \mathbf{c} \neq \mathbf{0} : \quad \sum_{i=1}^{d} c_i \mathbf{w}_{\cdot i} = \mathbf{0} \tag{2.14}$$

The vectors are **linearly independent** if they are not linearly dependent.

We can capture the coefficients of all linear dependencies at once by introducing the nullspace of $\mathbf{W}$.

Nullspace of a Matrix

Def. 2.2.4. Given a matrix $\mathbf{W} \in \mathbb{R}^{n \times d}$, the **nullspace** is the vector space of all coefficients that form linear dependencies of the columns of $\mathbf{W}$:

$$\text{nullspace}(\mathbf{W}) = \left\{ \mathbf{x} \in \mathbb{R}^d : \mathbf{W}\mathbf{x} = \mathbf{0} \right\} \tag{2.15}$$

Span and Rank of a Matrix

On the other hand, to measure how expressive $\mathbf{W}$ is, we check the space of outputs that can be produced. We note that the span and affine span, which we introduce, are equivalent to the linear hull and affine hull (Section 2.2.1), respectively. We use the span and affine span notation since they are more widely used in the literature and are semantically more closely associated to the concept of outputs that are "reachable".

Span of a Matrix

Def. 2.2.5. The **span** of a matrix $\mathbf{W} \in \mathbb{R}^{n \times d}$ is the subspace produced by taking all possible linear combinations (see Section 2.2.1) of its columns.

$$\text{span}(\mathbf{W}) = \sum_{i=1}^{d} c_i \mathbf{w}_{\cdot i}, \quad \forall c_i \in \mathbb{R} \tag{2.16}$$

We say that a sequence of vectors spans $\mathbb{R}^n$ if any point in $\mathbb{R}^n$ can be produced as a linear combination of the vectors.

Affine Span of a Matrix

Def. 2.2.6. The **affine span** of a matrix $\mathbf{W} \in \mathbb{R}^{n \times d}$ is the subspace produced by taking all possible affine combinations (see Section 2.2.1) of its columns.

$$\text{aff}(\mathbf{W}) = \sum_{i=1}^{d} c_i \mathbf{w}_{\cdot i}, \quad \forall c_i \in \mathbb{R} : \sum_{i=1}^{d} c_i = 1 \tag{2.17}$$

Basis of a Subspace

Def. 2.2.7. A set of vectors, B, are a basis for a subspace if they span the subspace and are linearly independent. We say that a basis is **orthonormal**, if every pair of basis vectors, $(\mathbf{u}, \mathbf{v}) \in B$, is orthogonal ($\mathbf{u}^\top \mathbf{v} = 0$), and all basis vectors, $\mathbf{u} \in B$, have Euclidean norm 1, i.e. $\|\mathbf{u}\|_2 = 1$.

Example 2: The Unit Vectors Span $\mathbb{R}^n$

We usually coordinatise $\mathbb{R}^n$ via the unit vectors $\mathbf{e}_i$ in the standard basis.

$$\mathbf{e}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \mathbf{e}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \mathbf{e}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix}, \quad \ldots, \quad \mathbf{e}_n = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix} \tag{2.18}$$

It will be useful to have a name for the number of vectors in a basis for a given subspace; this is the dimension of the subspace.

Dimension of a Subspace

Def. 2.2.8. The dimension of a subspace, $\dim(\text{span}(\mathbf{W}))$, is the number of vectors in a basis for that subspace, $\text{span}(\mathbf{W})$.

Dimension of an Affine Subspace

Def. 2.2.9. The dimension of an affine subspace, $\dim(\text{aff}(\mathbf{W}))$ is the dimension of the linear subspace that is being offset by a vector.

For example, in Fig. 2.1 we saw $\text{span}(\mathbf{I}_3)$ which spanned all of $\mathbb{R}^3$, and hence had $\dim(\text{span}(\mathbf{I}_3)) = 3$, and $\text{aff}(\mathbf{I}_3)$ which was an affine plane, and so $\dim(\text{aff}(\mathbf{I}_3)) = 2$.

Rank of a Matrix

Def. 2.2.10. The rank of a matrix is the dimension of $\text{span}(\mathbf{W})$. The rank measures how expressive the linear map is.

Example 3: Square Matrix with Rank 1

Let's form a matrix $\mathbf{W} \in \mathbb{R}^{n \times n}$ using n duplicates of the same row, $\mathbf{w}_1$. You can see that this matrix is degenerate: it does not really behave like a $n \times n$ matrix, since we could replace any calculation using $\mathbf{W}$ by n copies of a single dot product using $\mathbf{w}_1$. Any two rows are linearly dependent since taking the difference between any two rows gives us the zero vector. This matrix has rank 1 —it is as expressive as having a single vector.

2.2.3 Fine Measures of Linear Dependence

We saw that the rank of a matrix captures its largest subset of linearly independent vectors. While the rank quantifies which outputs can be expressed by $\mathbf{W}$, it is not granular enough to determine how many outputs are argmaxable. This is because when $\mathbf{W}$ is a tall matrix, i.e. $\mathbf{W} \in \mathbb{R}^{n \times d}, d < n$, the rank only captures the largest number of linearly independent rows, but ignores additional information about subsets of rows that may be dependent. There are matrices with the same rank that have a different number of argmaxable outputs. In order for two matrices $\mathbf{A} \in \mathbb{R}^{n \times d}$ and $\mathbf{B} \in \mathbb{R}^{n \times d}$ to have the same number of argmaxable outputs, we require them to be in general position.

Vectors in General Position

For methods that count the number of argmaxable outputs (Cover, 1965; Cover, 1967), we will require that the rows of the matrix are in general position. This assumption is generally reasonable when discussing neural network parameters, since randomly initialised matrices are in general position and we can assume that any square block of the matrix never becomes exactly linearly dependent (see Example 5).

Since we only need d vectors to form a basis for $\mathbb{R}^d$, any $d+1$ vectors in $\mathbb{R}^d$ must be linearly dependent. We say that a set of vectors are in general position in $\mathbb{R}^d$ if any k-subset of them, $k \leq d$, is no more linearly dependent than it needs to be. That is, for $\mathbb{R}^d, d \geq k$, no $k = 2$ vectors lie on a line through the origin , no $k = 3$ vectors lie on a plane through the origin, no d vectors lie in a $d-1$ subspace. Algebraically, it means that a $d \times d$ matrix formed by stacking any d out of the n vectors together has non-zero determinant.

General Position

Def. 2.2.11. We say that n vectors are in **general position** in $\mathbb{R}^d$ if all subsets of d or fewer vectors are linearly independent (Cover, 1965).

Example 4: Matrix not in General Position

Consider the matrix in Fig. 2.2. Its rank is 2, but its rows are *not* in general position since the last two rows are linearly dependent. Perturbing $\mathbf{w}_2$ or $\mathbf{w}_3$ does not change the rank of $\mathbf{W}$, but it does change the number of regions.

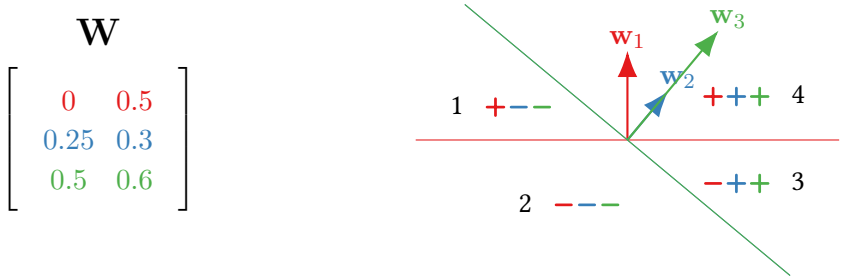


Figure 2.2: Example of 3 vectors in $\mathbb{R}^2$ which are *not* in general position. Left: The vectors stacked in the rows of $\mathbf{W}$. Right: We sketch the vectors in $\mathbb{R}^2$ to highlight that $\mathbf{w}_2$ and $\mathbf{w}_3$ are linearly dependent. We also plot the lines perpendicular to the vectors: they split $\mathbb{R}^2$ into the 4 numbered regions. Vectors in general position would produce 6 regions (see Section 2.4.4).

Example 5: Random Parameters are in General Position

We often initialise neural network classifiers by sampling a matrix $\mathbf{W} \in \mathbb{R}^{n \times d}$, $d < n$ at random as n d-dimensional samples of a normal distribution. For any $d \times d$ submatrix sampled this way to have rank $r < d$, we would need all rows or columns to fall in a $d-1$ dimensional subspace (e.g. a line in $\mathbb{R}^2$). However, this almost never happens. Therefore, any d rows of $\mathbf{W}$ are almost surely independent and the rows of $\mathbf{W}$ are almost surely in general position.

More generally, in order to understand which outputs are unargmaxable, we need to look at the linear (in)dependence of subsets of rows of $\mathbf{W}$. To access this more fine-grained information, we introduce minors: we look at determinants of square blocks within a matrix. We will need minors to introduce the Totally Positive Grassmanian, which is the structure of matrices we require to guarantee sparse outputs are argmaxable in Section 5.1.

Determinants

In order to define the determinant of a $n \times n$ matrix, we define the inversion and sign of a permutation.

Inversion of a Permutation The number of inversions of a permutation $I(\sigma)$ is the number of pairs of elements that are mapped "out of order" by the permutation

$$I(\sigma) = \sum_{i \in [n]} \sum_{j \in [n], j \neq i} [i < j, \sigma(i) > \sigma(j)] \tag{2.19}$$

Sign of a Permutation A permutation is odd if it has an odd number of inversions and even if it has an even number of inversions. The sign of a permutation is:

$$\operatorname{sgn}(\sigma) = \begin{cases} 1, & I(\sigma) = 2k \\ -1 & I(\sigma) = 2k+1 \end{cases} \qquad k \in \mathbb{Z}_{\geq 0} \tag{2.20}$$

Determinant

Def. 2.2.12. The determinant of a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, $n \in \mathbb{Z} : n \geq 1$ is:

$$\det \mathbf{A} = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} \mathbf{A}_{i,\sigma(i)} \tag{2.21}$$

where S_n is the group of permutations (Section 2.1.2).

We note that if $\det \mathbf{W} = 0$, the vectors in $\mathbf{W}$ are linearly dependent.

Minors A minor of a matrix $\mathbf{W} \in \mathbb{R}^{n \times d}$, $\Delta_{I,J}(\mathbf{W})$, is the determinant of a square submatrix of $\mathbf{W}$ formed by deleting the rows not in I and the columns not in J.[1] It is helpful to define the size of minors: a k-minor is the determinant of a $k \times k$ submatrix, e.g. $|I| = |J| = k$, $1 \leq k \leq d$. For example, for a $n \times d$ matrix with $d < n$, we compute a k minor by deleting $n - k$ rows and $d - k$ columns from the matrix and taking the determinant of this submatrix.

Maximal Minor

Def. 2.2.13. A **maximal minor** of a matrix $\mathbf{W} \in \mathbb{R}^{n \times d}$, $d < n$ is the determinant of one of its $d \times d$ submatrices:

$$\Delta_I(\mathbf{W}) = \Delta_{I,J}(\mathbf{W}), \quad |I| = d, \quad J = [d] \tag{2.22}$$

We note that if $\mathbf{W}$ is in general position, then all its maximal minors must be non-zero. We use the Vandermonde matrix in Example 6 to illustrate.

[1] This is equivalent to selecting the rows and columns based on the indices with the indices sorted in increasing order.

Example 6: $\mathcal{V}_{n,d}$ is in General Position

Consider a Vandermonde Matrix $\mathcal{V}_{n,d}$, $d < n$:

$$\mathcal{V}_{n,d} = \begin{bmatrix} 1 & t_1 & t_1^2 & \dots & t_1^d \\ 1 & t_2 & t_2^2 & \dots & t_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_n & t_n^2 & \dots & t_n^d \end{bmatrix}, \quad 0 < t_i < t_j, \quad \forall i, j \in [n] : i < j \tag{2.23}$$

Its rows are in general position: any subset of d rows that respects the total order on the t_i is $\mathcal{V}_{d,d}$, and its determinant is known (Kalman, 1984) to be:

$$\det \mathcal{V}_I = \prod_{i,j \in I : i<j} (t_j - t_i), \quad I = \binom{[n]}{d} \tag{2.24}$$

which is non-zero given that $t_j \neq t_i$.

Grassmanians

One of the main characters of this thesis is the matrix $\mathbf{W}$, which represents a bottlenecked classifier. As we will see, defining which outputs are argmaxable for $\mathbf{W}$ is invariant to the choice of a basis for $\mathbf{W}$; in fact it is invariant to any perturbation that leaves the sign of the maximal minors of $\mathbf{W}$ unchanged. [2] As such, we introduce concepts to define subspaces and families of subspaces that have the same argmaxable outputs. To this end:

1. We define families of matrices that span the same subspace (**Grassmanian**).

2. We define families of subspaces we will use in Chapter 5 to obtain our argmaxability guarantees (**Totally Positive Grassmanian**). Matrix representatives of these subspaces have maximal minors that are non-zero and have the same sign.

Grassmanian The Grassmanian, $\mathsf{Gr}_{n,d}$, is the set of all d-dimensional subspaces of $\mathbb{R}^n$. As such, any rank d matrix, $\mathbf{W} \in \mathbb{R}^{n \times d}$, with $1 \leq d \leq n$, is a representative of an element of $\mathsf{Gr}_{n,d}$.

[2] This is because the sign of the maximal minors of $\mathbf{W}$ define what is called the Chirotope of $\mathbf{W}$ (Björner et al., 1999, Definition 3.5.3). The Chirotope is an alternative axiomatisation of Oriented Matroids (Björner et al., 1999, Theorem 3.5.5), and the Oriented Matroid defined by the linear dependencies in $\mathbf{W}$ specifies which outputs are argmaxable, as we will discuss in Section 2.5.3.

Grassmanian

Def. 2.2.14.

$$\mathsf{Gr}_{n,d} = \left\{ \operatorname{span}(\mathbf{W}) : \mathbf{W} \in \mathbb{R}^{n \times d}, \dim(\operatorname{span}(\mathbf{W})) = d \right\} \tag{2.25}$$

We note that matrices that differ by a change of basis are representatives of the same element of $\mathsf{Gr}_{n,d}$. [3] For more details on the Grassmanian from an optimisation lens, see Absil et al. (2008, Section 3.4.4).

Next we subdivide the elements of the Grassmanian into subsets of subspaces by considering the sign of the maximal minors of their representatives.

Example 7: General Position and the Grassmanian

Consider the subset of the Grassmanian for which all maximal minors are non-zero:

$$\left\{ \operatorname{span}(\mathbf{W}) \in \mathsf{Gr}_{n,d} : \Delta_I(\mathbf{W}) \neq 0, \quad I \in \binom{[n]}{d} \right\} \tag{2.26}$$

Note that this is equivalent to claiming that the vectors are in general position: any $k \leq d$ rows are linearly independent.

Totally Positive Grassmanian We write $\mathsf{Gr}^+_{n,d}$ for the totally positive Grassmanian, the set of d-dimensional subspaces of $\mathbb{R}^n$ for which any representative, $\mathbf{W}$, has maximal minors that are non-zero and agree in sign (Karp, 2017, Section 1).

Totally Positive Grassmanian

Def. 2.2.15.

$$\mathsf{Gr}^+_{n,d} = \left\{ \operatorname{span}(\mathbf{W}) \in \mathsf{Gr}_{n,d} : \Delta_I(\mathbf{W})\,\Delta_J(\mathbf{W}) > 0, \quad I, J \in \binom{[n]}{d} \right\} \tag{2.27}$$

See also Postnikov (2006, Definition 3.1). [4] In Example 6, we saw that the determinants of all submatrices of the Vandermonde matrix $\mathcal{V}_{n,d}$ are positive, hence $\operatorname{span}\left(\mathcal{V}_{n,d}\right) \in \mathsf{Gr}^+_{n,d}$. In Chapter 5 we will see that the truncated Discrete Fourier Transform matrix is also a representative of an element in Gr^+.

[3] This would be clearer if we had defined the Grassmanian by introducing an equivalence relation on matrices, i.e. $\mathbf{W} \sim \mathbf{W}' \iff \operatorname{span}(\mathbf{W}) = \operatorname{span}(\mathbf{W}')$, and defined the Grassmanian as a quotient set (or even a quotient manifold), as done in Absil et al. (2008, Section 3.4.4). However, we elide these details since the notation creates a barrier to entry and does not aid our exposition.

[4] Postnikov's definition restricts maximal minors to be positive but also constrains column operations to matrices having positive determinant.

2.3 Polytopes

I call our world Flatland, not because we call it so, but to make its nature clearer to you, my happy readers, who are privileged to live in Space.

—Edwin Abbott Abbott, Flatland: A Romance of Many Dimensions

In Section 2.1, we introduced the outputs of our classifiers: categories $\triangle$, rankings $\hexagon$ and subsets $\square$. We now see where this notation stems from. We introduce polytopes, representations that will help us analyse outputs of bottlenecked classifiers. More specifically, we represent each output as a vertex of a polytope and show how the vertices behave under projections introduced by $\mathbf{W}$.

- Categories $\triangle$ arise as vertices of the Simplex $\triangle_d$ (Section 2.3.1).
- Rankings $\hexagon$ arise as vertices of the Permutohedron $\hexagon_d$ (Section 2.3.2).
- Subsets $\square$ arise as vertices of the Cube $\square_d$ (Section 2.3.3).

But first, some notation. Affine subspaces of dimension $0, 1, 2, \ldots, d-1$ in $\mathbb{R}^d$ are points, lines, planes, ... and hyperplanes, respectively. Extreme points of polytopes of corresponding dimensionality are 0-faces, 1-faces, 2-faces, ... and $(d-1)$-faces which we call vertices, edges, faces, ... and facets, see (Ziegler, 1995, Chapter 2).

We use P as a general variable for a polytope. For a polytope $P \subset \mathbb{R}^d$, we write $\operatorname{verts}(P) = \mathbf{P} \in \mathbb{R}^{n \times d}$ to extract its vertices as a matrix which contains its n vertices in the rows. We focus on the vertex representation for polytopes, i.e. the polytope is the convex hull (Definition 2.2.1) of its vertices, $P = \operatorname{conv}(\operatorname{verts}(P))$.

Some polytopes are special for this thesis since they provide representations for the outputs of interest. We highlight their connections throughout the thesis by representing them using their 2D shape and a subscript which represents their affine dimension, $\dim(\operatorname{aff}(\operatorname{verts}(P)))$ (see Definition 2.2.9). The special polytopes are the Simplex, $\triangle_d$, the Permutohedron, $\hexagon_d$, and the Cube, $\square_d$. The first two arise in the context of MCC for which the outputs are categories, and the latter for MLC where the outputs are label assignments.

2.3.1 Simplex △

Each vertex of $\triangle_d$ represents a category; one out of $d+1$ categories.

Simplex $\triangle_d$

Def. 2.3.1.

$$\triangle_d = \text{conv}\left(\mathbf{I}_{d+1}\right) \quad (2.28)$$

```python
def simplex(d):
    return np.eye(d+1)
```

Listing 1: verts $\left(\triangle_d\right)$.

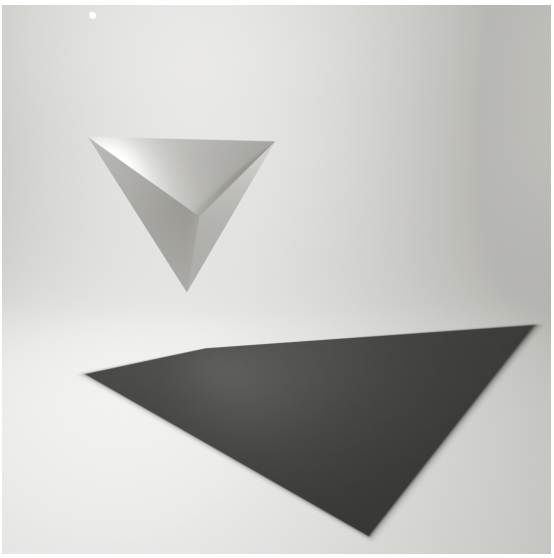


Figure 2.3: $\triangle_3$ and its shadow in 2D.

2.3.2 Permutohedron ⬡

Each vertex of $\hexagon_d$ represents a permutation/ranking of $d+1$ elements. [a]

Permutohedron $\hexagon_d$

Def. 2.3.2.

$$\hexagon_d = \text{conv}\left(\boldsymbol{\sigma} \in S_{d+1}\right) \quad (2.29)$$

[a] Both $\triangle_d$ and $\hexagon_d$ have vertices that sum to a constant, 1 and $\sum_i^d i$, respectively.

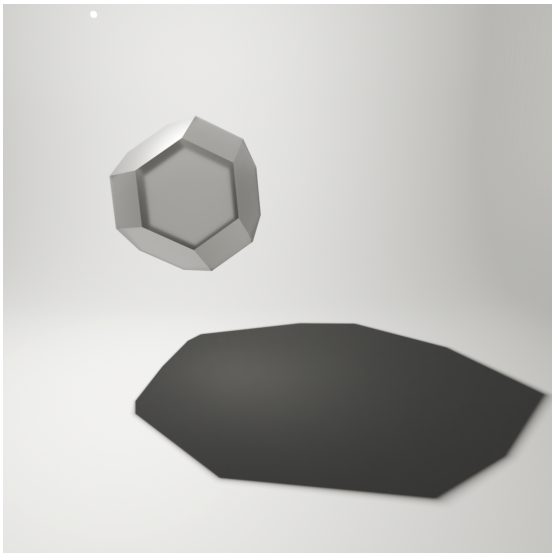


Figure 2.4: $\hexagon_3$ and its shadow in 2D.

```python
def permutohedron(d):
    P = [list(r) for r in itertools.permutations(range(d+1))]
    return P
```

Listing 2: verts $\left(\hexagon_d\right)$.

2.3.3 Cube □

Each vertex of $\square_d$ represents a subset of d elements.

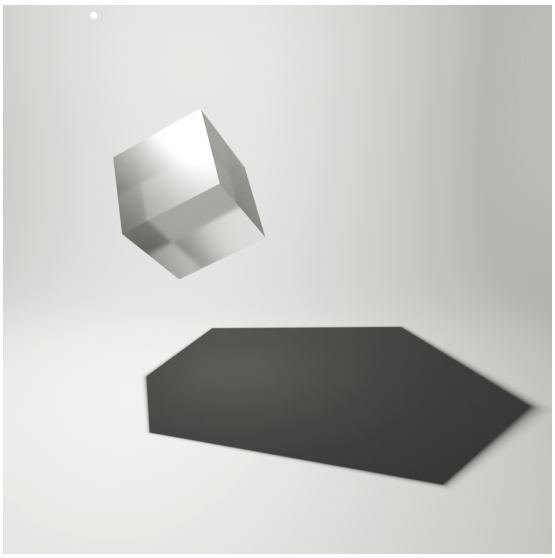


Figure 2.5: $\square_3$ and its shadow in 2D.

Cube $\square_d$

Def. 2.3.3.

$$\square_d = \text{conv}\left(\{1,-1\}^d\right) \quad (2.30)$$

```python
def cube(d):
    C = [list(r) for r in itertools.product([-1, 1], repeat=d)]
    return C
```

Listing 3: verts $(\square_d)$.

2.3.4 Shadows: Projecting Polytopes to Lower Dimensions

We now use shadows of polytopes to foreshadow how unargmaxability arises.[5] We do so to build some intuition about how $\mathbf{W}$ interacts with the output representations, namely the vertices of the polytopes.

In the figures above, we illustrated the shadows of $\triangle_3, \hexagon_3$ and $\square_3$. The shadows are 2D representations of the 3D objects, but they have lost a lot of information about their 3D source! For example, if you count the vertices of the shadow of the cube in Fig. 2.5, you will notice that only 6 of the 8 vertices of $\square_3$ surface in the shadow, 2 are killed by the projection. This is not by chance, in Fig. 2.7 we rotate the cube and always seem to only be able to represent 6/8 vertices in the shadow. Déjà vu? Remember Fig. 1.3? In Fig. 2.6 we see that for $\triangle$ there are projections that retain all vertices, but sometimes we do lose one vertex. Déjà vu again? Remember Fig. 1.2?

Shadows are a simple analogy for the constraints that arise from low dimensional projections. The vertices of the polytope represent outputs we want to predict, but

[5] Caveat: the shadow analogy is alluring but not 100% accurate. For a shadow to be a linear projection we would need a perfectly flat surface and a light source that has parallel light beams.

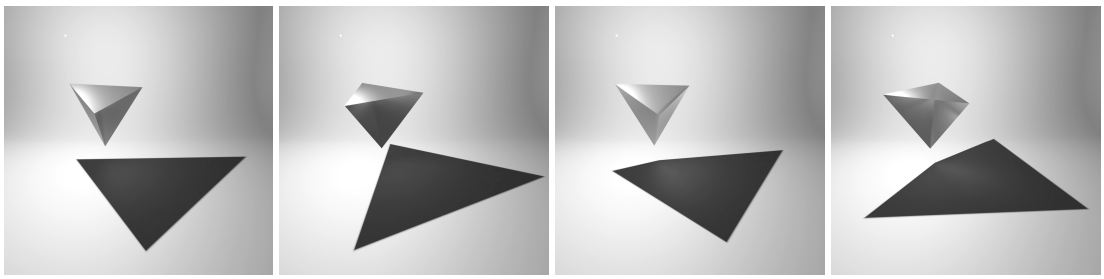


Figure 2.6: Projections of $\triangle$ *may* kill some of its vertices. We orient $\triangle_3$ in 4 different ways and plot its shadow. In the left two subfigures the projection kills a vertex of $\triangle_3$ and only $3/4$ vertices survive in the shadow, while in the other two all vertices survive.

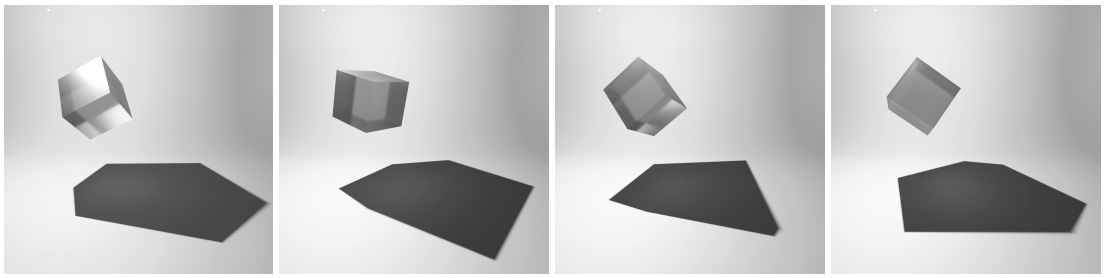


Figure 2.7: Projections of $\square$ *must* kill some of its vertices. We orient $\square_3$ in 4 different ways and plot its shadow. As can be seen, in all cases only $6/8$ vertices of $\square_3$ survive in the shadow.

$\mathbf{W}$ projects them to lower dimensions and we are forced to work with these shadows instead. As we will see in Section 3.5.2, vertices that are killed in the shadow (i.e. fall in the relative interior) are unargmaxable. We will see these shadows again, but first we need more tools.

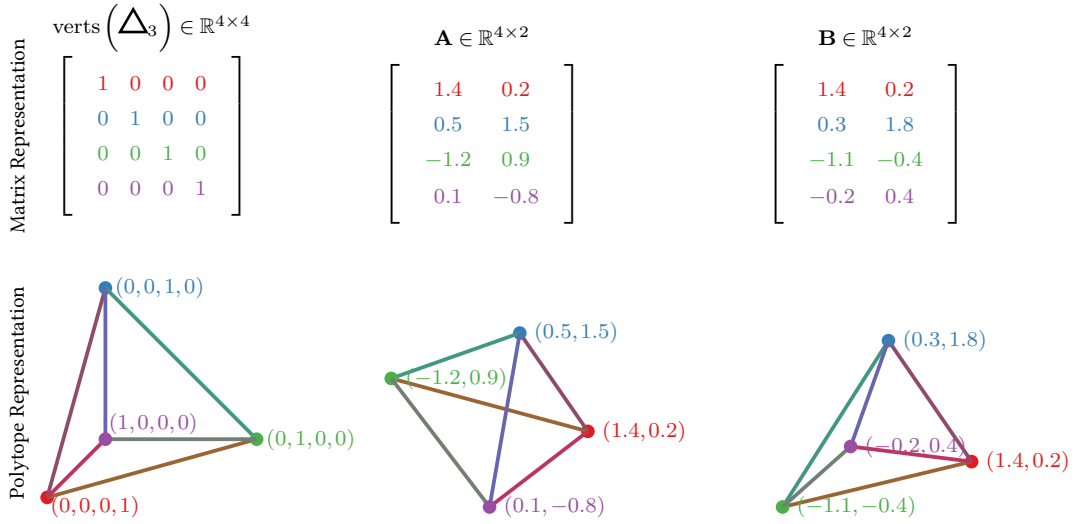
We now make projections precise in terms of the matrix, $\mathbf{W}$, so that we can easily analyse argmaxability in Section 3.5.2. We highlight that this usage of the term projection is distinct and less restricted than that often used in Linear Algebra, e.g. see (Strang, 2016, Chapter 4.2). In Linear Algebra, projection matrices are square, symmetric and idempotent. At a high-level, projection matrices project points onto a subspace, but retain the ambient dimension. In contrast, our use of the term projection follows the polytopes literature (Björner et al., 1999, p. 50, 2.2) and is less strict: a projection maps a higher dimensional space to a lower dimensional one and we "forget" about the higher dimensional space after the projection.

Polytope Projection (Shadows)

Def. 2.3.4. Consider a polytope P with n vertices in $\mathbb{R}^d$, represented as $P = \text{conv}(\text{verts}(P))$, $\text{verts}(P) = \mathbf{P} \in \mathbb{R}^{n \times z}$, and a projection $\mathbf{W}$ in $\mathbb{R}^{z \times d}$ that maps $\mathbb{R}^z \to \mathbb{R}^d$, $d \leq z$. We define the **projection** of P by $\mathbf{W}$ as:

$$\mathcal{P}(P, \mathbf{W}) = \text{conv}(\text{verts}(P))\,\mathbf{W} = \text{conv}(\mathbf{PW}) \tag{2.31}$$

We note that the last step, $\text{conv}(\text{verts}(P))\,\mathbf{W} = \text{conv}(\mathbf{PW})$, follows from the fact that a convex combination is a linear operation, so the order of application of the convex combination and the linear transformation by $\mathbf{W}$ can be swapped. Next we specifically discuss projections of $\triangle$ and $\square$.



(a) $\triangle_3$. (b) $\mathcal{P}\left(\triangle_3, \mathbf{A}\right)$. (c) $\mathcal{P}\left(\triangle_3, \mathbf{B}\right)$.

Figure 2.8: Example illustrating that polytopes are projections of the simplex. In (a) we show the simplex $\triangle_3$ as a polytope (below) and as a matrix of vertices, $\mathbf{I}_4 = \text{verts}\left(\triangle_3\right)$(above). In (b), we see that projecting $\triangle_3$ by matrix $\mathbf{A}$ (above) gives us a polygon, retaining all 4 vertices (below). However, in (c) the projection by $\mathbf{B}$ kills the purple vertex that falls in the relative interior of the triangle. As we will see in Section 3.5, the category corresponding to the purple vertex in Fig. 2.8(c) is unargmaxable.

All polytopes are projections of the simplex Consider an arbitrary polytope P and its matrix of vertices $\mathbf{P} = \text{verts}(P)$, $\mathbf{P} \in \mathbb{R}^{n \times d}$. It may seem trivial to write

$\mathbf{P} = \mathbf{I}_n\mathbf{P}$, but there is an elegant interpretation of this (Ziegler, 1995, Example 0.9).

$$\begin{aligned}\text{conv}\left(\text{verts}\left(P\right)\right) &= \text{conv}\left(\mathbf{P}\right) && (2.32)\\ &= \text{conv}\left(\mathbf{I}_n\mathbf{P}\right) && (2.33)\\ &= \text{conv}\left(\text{verts}\left(\triangle_{n-1}\right)\mathbf{P}\right) && (2.34)\\ &= \mathcal{P}\left(\triangle_{n-1}, \mathbf{P}\right) && (2.35)\end{aligned}$$

So any polytope, P, with n vertices can be thought of as a projection of $\triangle_{n-1}$, i.e. $\mathbf{P}$ maps the simplex to P. We illustrate this fact with two example projections, $\mathbf{A}$ and $\mathbf{B}$ in Fig. 2.8. For the following sections, we assume all projections map to a lower dimensional space, i.e. $\mathbf{W} \in \mathbb{R}^{z\times d}$, $d < z$.

Projections of the Simplex May Lose Vertices

What does this tell us about vertices lost in the projection of simplices? Well, the shadow does not have to lose vertices —think of a polygon, $\text{conv}\left(\mathbf{P}\right)$, $\mathbf{P} \in \mathbb{R}^{n\times 2}$ —all n vertices can survive in $\mathbb{R}^2$. However, it is possible that vertices are indeed killed by the projection. We illustrate both cases in Figs. 2.6 and 2.8. We conclude that vertices of the simplex *may* be killed by a projection. However, as we discuss next, projections of the cube *must* lose vertices.

Projections of the Cube Must Lose Vertices

In Fig. 2.7, we made a visual argument for the fact that projections of $\square_n$ to $\mathbb{R}^d$, $d < n$ must kill vertices. We illustrated the shadow for 4 different projections of $\square_3$, i.e. $\mathcal{P}\left(\square_3, \mathbf{W}\right)$ with 4 different projections $\mathbf{W} \in \mathbb{R}^{3\times 2}$. As can be seen, the shadows in $\mathbb{R}^2$ in all 4 cases only have 6 of the vertices.

While we have seen evidence of the pattern, we will not show that any projection of the cube must lose vertices, just yet. We ask the kind reader to suspend any disbelief until Section 2.5.3, where we explain that the pattern holds in general via the connection to Oriented Matroids.

Families of projections that must lose vertices are important in terms of argmaxability, so we name them and introduce them more formally in the next section.

2.3.5 Zonotopes

Projections of the cube are a subfamily of polytopes known as zonotopes (Ziegler, 1995, Section 7.2). They are so named because the faces parallel to any row of the projection forms a zone that wraps around the polytope (Eppstein, 1996).

Zonotope

Def. 2.3.5. A **zonotope**, $Z \subset \mathbb{R}^d$, is a projection of $\square_n$ defined by the projection $\mathbf{W} \in \mathbb{R}^{n \times d}$:

$$Z = \mathcal{P}\left(\square_n, \mathbf{W}\right) \tag{2.36}$$

We also introduce the shorthand $\mathcal{Z}\left(\mathbf{W}\right), \mathbf{W} \in \mathbb{R}^{n \times d}$:

$$\mathcal{Z}\left(\mathbf{W}\right) = \mathcal{P}\left(\square_n, \mathbf{W}\right) \tag{2.37}$$

Since a zonotope is a projection of the cube, we identify each of its vertices using the corresponding coordinates from the vertex that was pulled down from the cube, i.e. $\{+1, -1\}^n$. This is the sign vector representation of each vertex (Section 2.5.1).

Example 8: $\hexagon_d$ is a Zonotope

$$\hexagon_d = \mathcal{Z}\left(\mathbf{B}_d\right) \tag{2.38}$$

$$= \mathcal{P}\left(\square_{\binom{d}{2}}, \mathbf{B}_d\right) \tag{2.39}$$

where the projection $\mathbf{B}_d \in \mathbb{R}^{\binom{d}{2} \times d}$ is the Braid Matrix (see Eq. (2.52)). See also Ziegler (1995, Example 7.15).

Summary

- Outputs of MLC and MCC correspond to vertices of polytopes.
- Zonotopes are projections of cubes.
- $\mathbf{W}$ projects polytopes to lower dimensions.
- $\mathbf{W}$ *must* kill vertices of zonotopes.
- $\mathbf{W}$ *may* kill vertices of the simplex.

More on Polytopes See the Summary in Section 2.4.4 for pointers to the results on polytope projections. Excellent resources to learn more about polytopes are Ziegler

(1995) and Federico Ardila's lectures on polytopes.

2.4 Hyperplane Arrangements

In Figs. 1.2 and 1.3, we saw examples of decision boundaries of classifiers, i.e. regions in feature space where the output of a classifier changes. In this section, we learn a lot more about decision boundaries from our friends in combinatorics. Linear decision boundaries are a well studied object in combinatorics, but they come under a different name: **hyperplane arrangements**.

As with polytopes, some hyperplane arrangements have special status in this thesis. These arrangements partition Euclidean space into regions (Definition 2.4.6) that are representations for the outputs of our classifiers.

- Subsets □ arise as regions of the Boolean Arrangement ⊞ (Section 2.4.1).
- Rankings ⬡ arise as regions of the Braid Arrangement ❖ (Section 2.4.2).
- Categories △ arise as regions of the Voronoi Tesselation ⧋ (Section 2.4.3).

But before we dive into explaining those hyperplane arrangements, we introduce hyperplanes, halfspaces and intersections of halfspaces (Boyd et al., 2004, Section 2.2.1). We note that in what follows, the right-hand side is in bold because these are elements of a sign vector and not because the result is a vector. See Definition 2.5.1 for the definition of the $\text{sign}()$ function.

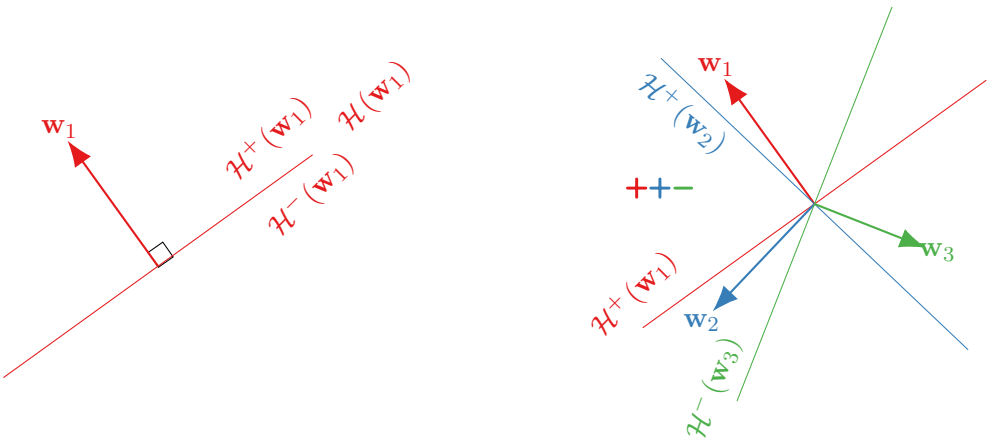
Hyperplane

Def. 2.4.1. A **hyperplane**, $\mathcal{H}$, in $\mathbb{R}^d$ is an affine subspace of dimension $d-1$. The hyperplane has one degree of freedom removed by specifying a constraint: a **normal vector** $\mathbf{w} \in \mathbb{R}^d$ to which it is perpendicular. If the hyperplane is linear, we define it in terms of its normal vector alone as $\mathcal{H}(\mathbf{w})$.

$$\mathcal{H}(\mathbf{w}) = \left\{ \mathbf{x} \in \mathbb{R}^d : \text{sign}\left(\mathbf{w}^\top \mathbf{x}\right) = \mathbf{0} \right\} \tag{2.40}$$

If the hyperplane is affine, we offset the hyperplane by b in the direction of $\mathbf{w}$:

$$\mathcal{H}(\mathbf{w}, b) = \left\{ \mathbf{x} \in \mathbb{R}^d : \text{sign}\left(\mathbf{w}^\top \mathbf{x} - b\right) = \mathbf{0} \right\} \tag{2.41}$$



(a) Hyperplane $\mathcal{H}(\mathbf{w}_1)$ splits $\mathbb{R}^2$ in two halfspaces.

(b) An intersection of halfspaces.

Figure 2.9: Left: We define a hyperplane, $\mathcal{H}(\mathbf{w}_1)$, via its normal vector, $\mathbf{w}_1$. In $\mathbb{R}^2$ a hyperplane is a line $\mathbf{x} \in \mathbb{R}^2 : \mathbf{w}_1{}^\top \mathbf{x} = 0$. $\mathcal{H}(\mathbf{w}_1)$ splits $\mathbb{R}^2$ into two halfspaces, $\mathcal{H}^+(\mathbf{w}_1)$ which is on the side $\mathbf{w}_1$ points to, and $\mathcal{H}^-(\mathbf{w}_1)$, which is the other halfspace. Right: We denote an intersection of halfspaces by using a sign vector. Here $++-$ is short for $\mathcal{H}^+(\mathbf{w}_3) \cap \mathcal{H}^+(\mathbf{w}_2) \cap \mathcal{H}^-(\mathbf{w}_3)$.

Halfspace

Def. 2.4.2. A hyperplane $\mathcal{H}(\mathbf{w}, b)$, $\mathbf{w} \in \mathbb{R}^d$, splits $\mathbb{R}^d$ into two halfspaces. The **positive halfspace**, $\mathcal{H}^+(\mathbf{w}, b)$, and the **negative halfspace**, $\mathcal{H}^-(\mathbf{w}, b)$:

$$\mathcal{H}^+(\mathbf{w}, b) = \left\{\mathbf{x} \in \mathbb{R}^d : \operatorname{sign}\left(\mathbf{w}^\top \mathbf{x} - b\right) = +\right\} \tag{2.42}$$

$$\mathcal{H}^-(\mathbf{w}, b) = \left\{\mathbf{x} \in \mathbb{R}^d : \operatorname{sign}\left(\mathbf{w}^\top \mathbf{x} - b\right) = -\right\} \tag{2.43}$$

If $b = 0$, we write $\mathcal{H}^+(\mathbf{w})$ and $\mathcal{H}^-(\mathbf{w})$, respectively.

We illustrate a hyperplane and its corresponding halfspaces in Fig. 2.9(a).

Halfspace Intersection

Def. 2.4.3.

$$\mathcal{H}^{\mathbf{s}}(\mathbf{W}, \mathbf{b}) = \bigcap_{i \in [n]} \mathcal{H}^{s_i}(\mathbf{w}_i, b_i) \tag{2.44}$$

$$= \left\{\mathbf{x} \in \mathbb{R}^n : \operatorname{sign}\left(\mathbf{w}_i^\top \mathbf{x} - b_i\right) = s_i, \quad i \in [n]\right\} \tag{2.45}$$

We illustrate an intersection of halfspaces and its sign vector representation in Fig. 2.9(b).

Orthant

Def. 2.4.4. Consider $\mathbb{R}^n$ and the n coordinate hyperplanes, $\mathcal{H}(\mathbf{e}_i), i \in [n]$. The hyperplanes partition $\mathbb{R}^n$ into 2^n orthants. Each orthant, $\mathcal{O}^{\mathbf{s}}$, is the intersection of n halfspaces: where $\mathbf{s} = \{+, -\}^n$ is a sign vector telling us whether we take $\mathcal{H}^+(\mathbf{e}_i)$ or $\mathcal{H}^-(\mathbf{e}_i)$:

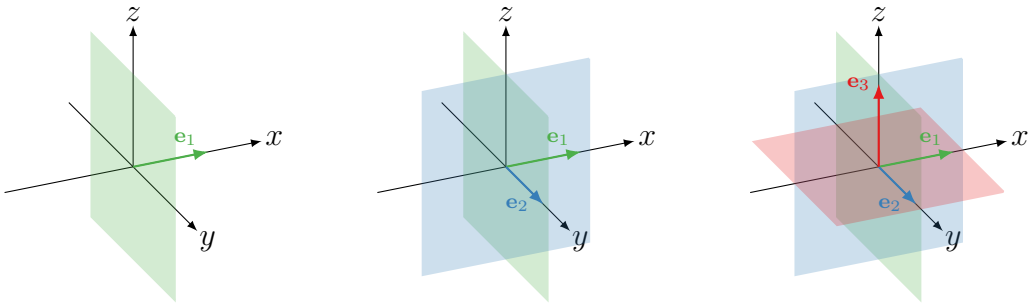
$$\mathcal{O}^{\mathbf{s}} = \left\{\mathbf{x} \in \mathbb{R}^n : \operatorname{sign}\left(\mathbf{e}_i^\top \mathbf{x}\right) = s_i, \quad i \in [n]\right\} \tag{2.46}$$

where $\mathbf{e}_i, i \in [n]$ is the standard basis for $\mathbb{R}^n$ (Example 2), i.e. $\mathbf{e}_i^\top \mathbf{x} = x_i$.

Sign quantisation gives us the orthant We note that since $\mathbf{e}_i^\top \mathbf{x} = x_i$, we have:

$$\operatorname{sign}(\mathbf{x}) = \mathbf{s} \in \{+, -\}^n \iff \mathbf{x} \in \mathcal{O}^{\mathbf{s}} \tag{2.47}$$

i.e. we can identify which orthant of $\mathbb{R}^n$ a point falls in by taking the sign of its n coordinates. If the sign vector has zero entries, it is on the boundary of an orthant, e.g. $\mathbf{x}$ is on one or more axis-aligned hyperplanes.



(a) We add $\mathcal{H}(\mathbf{e}_1)$. (b) We add $\mathcal{H}(\mathbf{e}_2)$. (c) We add $\mathcal{H}(\mathbf{e}_3)$.

Figure 2.10: How to construct a hyperplane arrangement, one hyperplane at a time. We start on the left with $\mathcal{H}(\mathbf{e}_1)$, the plane perpendicular to the normal vector $\mathbf{e}_1$. We incrementally add $\mathcal{H}(\mathbf{e}_2)$ and $\mathcal{H}(\mathbf{e}_3)$. The arrangement we have constructed on the right may seem familiar, the 8 regions are the orthants of $\mathbb{R}^3$. Note: The planes extend to infinity, we truncate them for visualisation purposes.

Hyperplane Arrangement

Def. 2.4.5. A real **hyperplane arrangement**, $\mathcal{A}$, is defined as a set of n hyperplanes in $\mathbb{R}^d$, $\mathcal{A} = \{\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_n\}$.

Following our notation above using normal vectors, we use the matrix $\text{normals}(\mathcal{A}) = \mathbf{W} \in \mathbb{R}^{n \times d}$ to define a specific hyperplane arrangement:

$$\mathcal{H}(\mathbf{W}) = \{\mathcal{H}(\mathbf{w}_1), \mathcal{H}(\mathbf{w}_2), \ldots, \mathcal{H}(\mathbf{w}_n)\} \tag{2.48}$$

The set of *regions* $\mathcal{R}$ defined by a hyperplane arrangement $\mathcal{A}$ are the connected components X of Euclidean space $\mathbb{R}^d$ left when we remove the hyperplanes of $\mathcal{A}$, namely $X = \mathbb{R}^d - \bigcup_{\mathcal{H} \in \mathcal{A}} \mathcal{H}$. Each region is defined by an intersection of halfspaces. Given an ordering of the hyperplanes that define the halfspaces, we can represent the region using its sign vector. The sign vector has as many elements as the number of hyperplanes in the arrangement, and each sign denotes which side of the hyperplane the region is on (see also Aguiar et al. (2017, Section 1.4.1)).

Hyperplane Arrangement Region

Def. 2.4.6. We define a **region** of a hyperplane arrangement $\mathcal{H}(\mathbf{W})$ as the intersection of halfspaces with normal vectors defined by $\mathbf{W} \in \mathbb{R}^{n \times d}$ and the orientation of the halfspaces by the sign vector $\mathbf{s} \in \{+, -\}^n$.

$$\mathcal{R}^{\mathbf{s}}(\mathbf{W}) = \mathcal{H}^{\mathbf{s}}(\mathbf{W}) \tag{2.49}$$

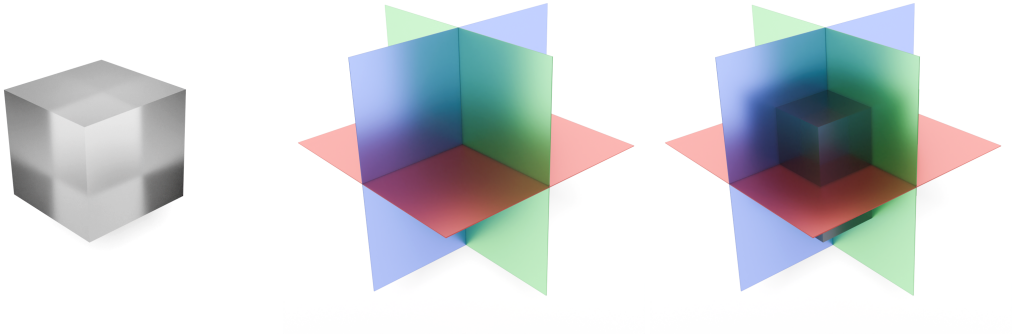
We represent the set of all regions of an arrangement as $\mathcal{R}(\mathbf{W})$, i.e. we omit $\mathbf{s}$:

$$\mathcal{R}(\mathbf{W}) = \left\{ \mathcal{R}^{\mathbf{s}}(\mathbf{W}) \quad \forall \mathbf{s} \in \{+, -\}^n : \mathcal{H}^{\mathbf{s}}(\mathbf{W}) \neq \emptyset \right\} \tag{2.50}$$

As we alluded to earlier, the decision boundaries of linear classifiers give rise to hyperplane arrangements. We now introduce two hyperplane arrangements. The Boolean Arrangement, ⊞, which arises when we consider decision boundaries in MLC. And the Braid Arrangement, ⊛, which arises when we consider the decision boundaries in MCC.

2.4.1 Boolean Arrangement ⊞

The Boolean Arrangement, $\boxplus_n$, consists of the n coordinate hyperplanes. Namely, the normal vectors which define it are the unit vectors $\mathcal{H}(\mathbf{I}_n)$. $\boxplus_n$ and splits $\mathbb{R}^n$ into



(a) $\square_3$. (b) $\boxplus_3$. (c) $\square_3$ inside $\boxplus_3$.

Figure 2.11: The Boolean Hyperplane Arrangement $\boxplus_3$. We superimpose $\square_3$ over $\boxplus_3$ to illustrate the correspondence between subsets, vertices of $\square_3$ and regions of $\boxplus_3$. As can be seen, $\boxplus_3$ has $2^3 = 8$ regions, since each vertex of $\square_3$ falls in one region.

2^n regions, the orthants of $\mathbb{R}^n$. It is of interest to us since it gives us the decision boundaries for MLC: each region corresponds to a label assignment.

Boolean Arrangement $\boxplus_n$

Def. 2.4.7.

$$\boxplus_n = \mathcal{H}(\mathbf{I}_n), \quad \mathbf{I}_n \in \mathbb{R}^{n \times n} \tag{2.51}$$

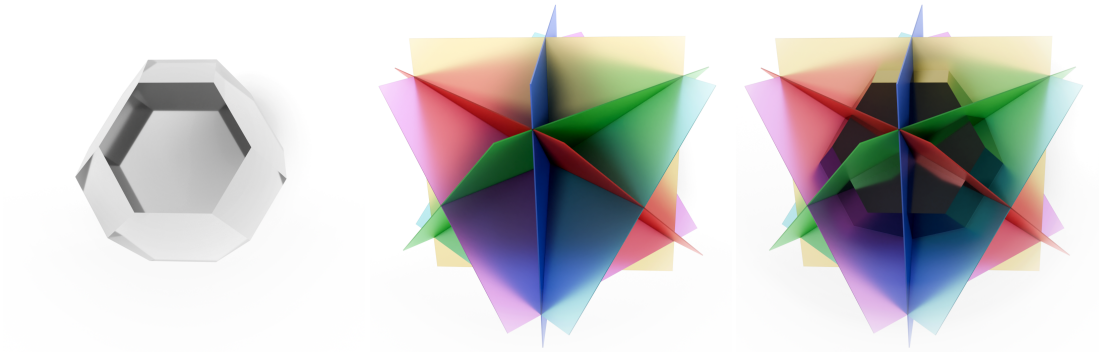
where $\mathbf{I}_n$ is the $n \times n$ identity matrix.

2.4.2 Braid Arrangement

Before we define the Braid Hyperplane arrangement, we introduce the braid matrix, the matrix of normal vectors that define it.

The Braid Matrix $\mathbf{B}_n$

The braid matrix is the incidence matrix of a directed fully connected graph.



(a) $\hexagon_3$. (b) $\permutohedron_3$. (c) $\hexagon_3$ inside $\permutohedron_3$.

Figure 2.12: The Braid Hyperplane Arrangement $\permutohedron_3$. We superimpose $\hexagon_3$ over $\permutohedron_3$ to illustrate the correspondence between rankings, vertices of $\hexagon_3$ and regions of $\permutohedron_3$. As can be seen, $\permutohedron_3$ has $4! = 24$ regions, since each vertex of $\hexagon_3$ falls in one region.

The **braid matrix**, $\mathbf{B}_n \in \mathbb{R}^{\binom{n}{2} \times n}$, is:

$$\mathbf{B}_n = \begin{bmatrix} \mathbf{e}_1 - \mathbf{e}_2 \\ \mathbf{e}_1 - \mathbf{e}_3 \\ \vdots \\ \mathbf{e}_2 - \mathbf{e}_3 \\ \vdots \\ \mathbf{e}_i - \mathbf{e}_j \end{bmatrix}, \quad i,j \in [n], \quad i < j \tag{2.52}$$

where $\mathbf{e}_i$ is the standard unit vector.

```python
def braid(n):
    W = np.eye(n)
    N = math.comb(n, 2)
    B = np.zeros((N, n))
    combs = combinations(range(n), 2)
    for a, (i,j) in enumerate(combs):
        B[a] = W[i] - W[j]
    return B
```

Listing 4: Building the Braid Matrix $\mathbf{B}_n$.

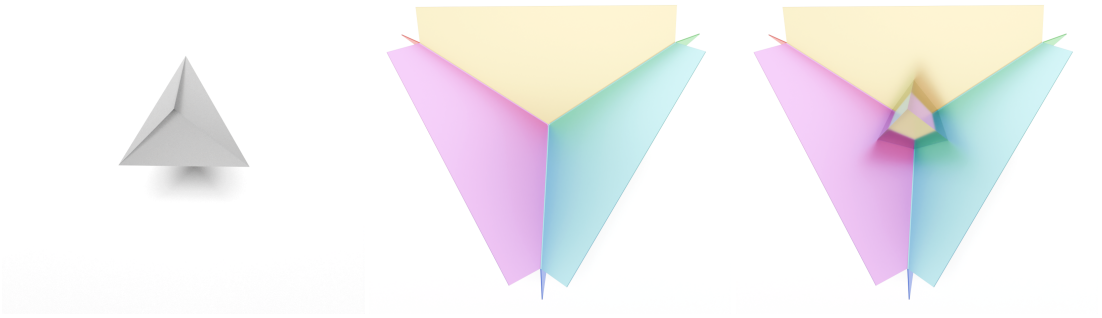
Braid Arrangement $\permutohedron_n$

Def. 2.4.8.

$$\permutohedron_n = \mathcal{H}(\mathbf{B}_n), \quad \mathbf{B}_n \in \mathbb{R}^{\binom{n}{2} \times n} \tag{2.53}$$

where $\mathbf{B}_n$ is the braid matrix Eq. (2.52).

The Braid Arrangement (Aguiar et al., 2017, Section 6.3), $\permutohedron_n$, is a hyperplane arrangement that arises when we consider decision boundaries in multi-class classification. This is because it partitions space into $n!$ regions which correspond to rankings of $n+1$ elements (see Fig. 2.12).



(a) $\triangle_3$ (b) $\triangle_3$ (c) $\triangle_3$ inside $\triangle_3$.

Figure 2.13: The Voronoi tesselation $\triangle_3$. We superimpose $\triangle_3$ over $\triangle_3$ to illustrate the correspondence between categories, vertices of $\triangle_3$ and regions of $\triangle_3$. As can be seen, $\triangle_3$ has 4 regions; each vertex of $\triangle_3$ falls in one region.

2.4.3 Voronoi Tesselation $\triangle$

The Voronoi tesselation, $\triangle$, arises as the decision boundaries in MCC (Hess et al., 2020, Theorem 1).[6] Each region of $\triangle_n$ corresponds to the set of inputs in $\mathbb{R}^n$ for which a target category is assigned the largest score. As such, it is straightforward to see that each region of $\triangle_n$ is a union of regions of $\circledast_n$: the regions that rank a given class above all others.

Voronoi Tesselation $\triangle_n$

Def. 2.4.9.

$$\triangle_n = \bigcup_{i\in[n]} \left(\bigcap_{j\in[n], j\neq i} \mathcal{H}^+(\mathbf{e}_i - \mathbf{e}_j) \right) \tag{2.54}$$

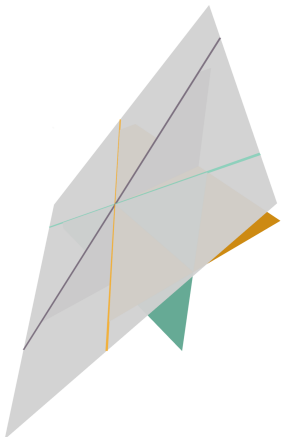
We note that similar to how $\triangle$ is not a zonotope, the Voronoi tesselation is not a hyperplane arrangement: the regions cannot be formed by hyperplanes without truncating some of them.

2.4.4 Slicing: Restricting Arrangements to Lower Dimensions

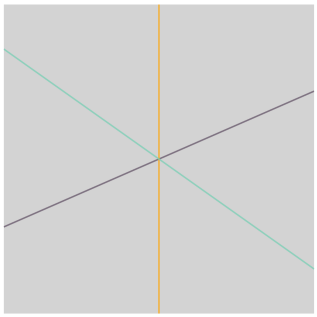
We now build intuition about how $\mathbf{W}$ interacts with the output representations, namely the regions of the hyperplane arrangements. We saw that $\boxplus_n$ gives us the regions for MLC with n labels and $\triangle_n$ gives us the regions for MCC with n categories via $\circledast_n$. However, in ML we do not use $\mathbf{I}_n$ as an output layer, we usually learn a matrix

[6] This is true when $\mathbf{W} \in \mathbb{R}^{n\times n}$ is square and full rank.

a) $\boxplus_3$

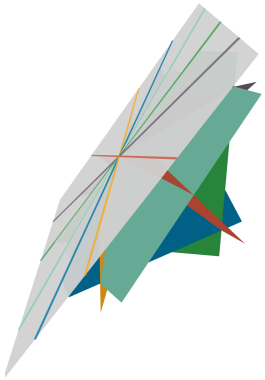


b) Intersect by $\text{span}(\mathbf{W})$

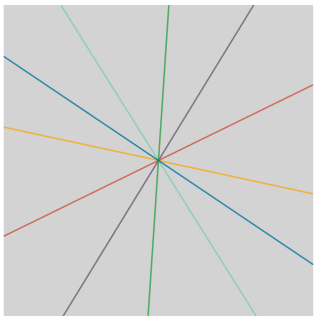


c) $\mathcal{P}\left(\boxplus_3, \mathbf{W}\right), \in \mathbb{R}^2$

a) $\circledast_3$



b) Intersect by $\text{span}(\mathbf{W})$



c) $\mathcal{P}\left(\circledast_3, \mathbf{W}\right), \in \mathbb{R}^2$

$\mathbf{W} \in \mathbb{R}^{n \times d}$, $d < n$. We now explain how this matrix $\mathbf{W}$ affects the decision boundaries: it restricts the corresponding hyperplane arrangement to a subspace. As a result of the restriction, some of the regions of the hyperplane arrangement are lost.

The restriction operation intersects a hyperplane arrangement with $\text{span}(\mathbf{W})$. Restriction is equivalent to projecting the normals of the hyperplanes, as we did for vertices in Definition 2.3.4, see also Björner et al. (1999, p. 12, d).

Hyperplane Arrangement Restriction

Def. 2.4.10. Consider a hyperplane arrangement $\mathcal{A}$ represented as $\text{normals}(\mathcal{A}) = \mathbf{A} \in \mathbb{R}^{n \times z}$, and a matrix $\mathbf{W}$ in $\mathbb{R}^{z \times d}$ that maps $\mathbb{R}^z \to \mathbb{R}^d$, $d \leq z$. We define the **restriction** of $\mathcal{A}$ by $\mathbf{W}$ as:

$$\mathcal{P}(\mathcal{A}, \mathbf{W}) = \text{normals}(\mathcal{A})\,\mathbf{W} = \mathbf{A}\mathbf{W} \tag{2.55}$$

Intuitively, if $\text{span}(\mathbf{W})$ is a hyperplane and we restrict $\boxplus_d$ to it, the hyperplane "slices" the orthants, as in Section 2.4.4. If the subspace is lower dimensional, $\text{span}(\mathbf{W})$

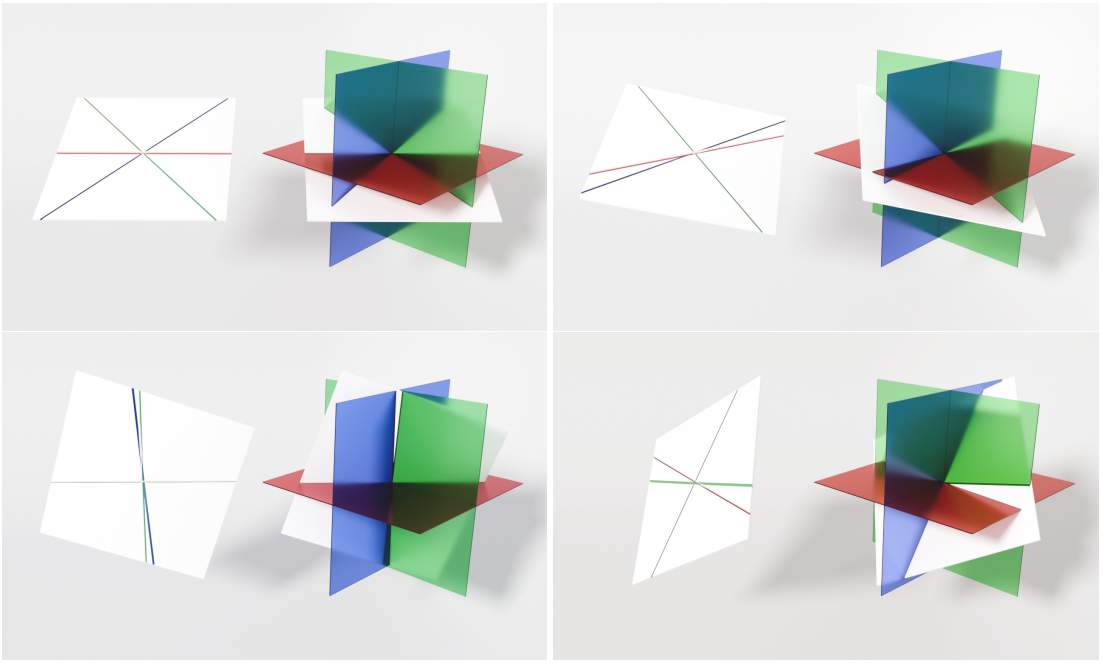


Figure 2.15: Restricting $\boxplus$ *must* kill regions $\boxdot$. We restrict $\boxplus_3$ to $\operatorname{span}(\mathbf{W})$ for four different choices of $\mathbf{W}$ and show the intersection. In all four subfigures only $6/8$ regions survive.

"stabs" the orthants (imagine a 1d ray stabbing the orthants in $\mathbb{R}^3$). Only the intersected orthants arise as regions in the restricted arrangement.

As we can see from Example 9 and 10, both $\mathcal{P}\left(\boxplus_n, \mathbf{W}\right)$ and $\mathcal{P}\left(\circledast_n, \mathbf{W}\right)$ can be interpreted as intersecting orthants, of $\mathbb{R}^n$ and $\mathbb{R}^{\binom{n}{2}}$, respectively.

Example 9: $\mathcal{H}(\mathbf{W})$, $\mathbf{W} \in \mathbb{R}^{n \times d}$ is a Restriction of $\boxplus_n$

$$\begin{aligned} \mathcal{H}(\mathbf{W}) &= \mathcal{H}(\mathbf{I}_n \mathbf{W}) && (2.56) \\ &= \mathcal{P}\left(\boxplus_n, \mathbf{W}\right) && (2.57) \end{aligned}$$

Example 10: $\circledast_d$ is a Restriction of $\boxplus_{\binom{n}{2}}$

$$\begin{aligned} \circledast_n &= \mathcal{H}(\mathbf{B}_n) && (2.58) \\ &= \mathcal{H}\left(\mathbf{I}_{\binom{n}{2}} \mathbf{B}_n\right) && (2.59) \\ &= \mathcal{P}\left(\boxplus_{\binom{n}{2}}, \mathbf{B}_n\right) && (2.60) \end{aligned}$$

Therefore, $\mathcal{P}\left(\circledast_n, \mathbf{W}\right) = \mathcal{P}\left(\boxplus_{\binom{n}{2}}, \mathbf{B}_n \mathbf{W}\right)$.

Restrictions of Hyperplane Arrangements Must Lose Regions

This statement is analogous to the one about projections of Zonotopes: $\mathbf{W}$ must kill vertices of Zonotopes, and it must also kill regions of Hyperplane Arrangements. We see this in Fig. 2.15, where we see a reincarnation of Fig. 1.3, and once again we can only represent 6/8 regions.

Restrictions of the Voronoi Tesselation May Lose Regions

As we mentioned earlier, $\triangle_n$ is not a hyperplane arrangement, but its regions are formed as a union of regions from $\hexagram_n$. Therefore, for a region of $\triangle$ to be killed by $\mathbf{W}$, all corresponding regions of $\hexagram$ must be killed. This does not always happen, as we show in Fig. 2.16, which is a reincarnation of Fig. 1.2. Therefore, once again analogously to the polytope case, we say that $\mathbf{W}$ may kill regions of $\triangle$.

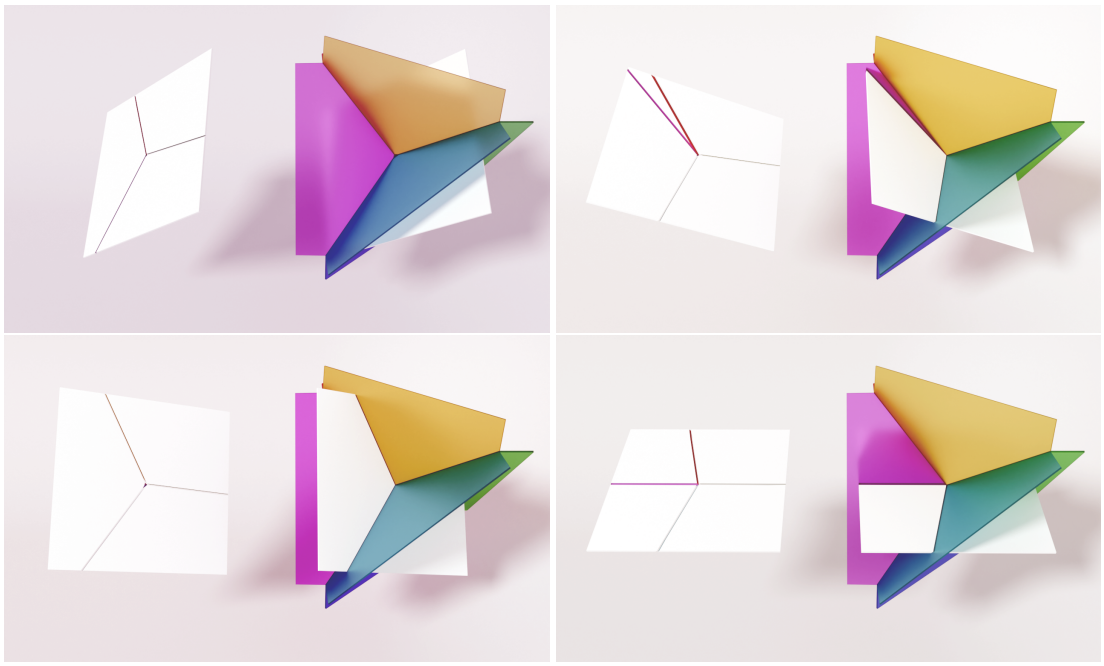


Figure 2.16: Restricting $\triangle$ *may* kill regions . We restrict $\triangle_4$ to $\mathrm{span}(\mathbf{W})$ for four different choices of $\mathbf{W}$ and show the intersection. In the left two subfigures the restriction kills a region and only $3/4$ regions survive, while in the subfigures on the right all regions survive.

Number of Regions

As we have seen, restricted arrangements lose regions when compared to unrestricted ones. As we will explain in Section 2.5.1, these lost regions correspond to unargmaxable

outputs. It is therefore natural to ask: how many of the outputs can we represent when we have an arrangement restricted by $\mathbf{W}$? The answer to this question corresponds to counting the number of regions of $\mathcal{P}(\mathcal{A}, \mathbf{W})$.

We discuss the number of regions of $\mathcal{P}\left(\boxplus, \mathbf{W}\right)$ and $\mathcal{P}\left(\circledast, \mathbf{W}\right)$ under the assumption that $\mathbf{W} \in \mathbb{R}^{n \times d}$ is in general position (see Definition 2.2.11). For a more general solution for counting regions in arrangements we point to Zaslavsky's PhD thesis (Zaslavsky, 1975).

Counting the Regions of the Restricted Boolean Arrangement

We are specifically interested in the number of regions of $\mathcal{P}\left(\boxplus_n, \mathbf{W}\right)$, since as we will see in Chapter 5, these correspond to the number of argmaxable label assignments (Definition 3.4.5) in MLC. For $\boxplus_n$, we have the following result (Cover, 1965), see MacKay (2003, Chapter 40) for a clear exposition of this result.

Number of Regions of $\mathcal{H}(\mathbf{W})$

Thm. 2.1. (Cover, 1965, Thm 2) If $\mathbf{W} \in \mathbb{R}^{n \times d}$ is in general position, the number of regions of $\mathcal{H}(\mathbf{W})$ is:

$$|\mathcal{R}(\mathbf{W})| = 2\sum_{d'=0}^{d-1} \binom{n-1}{d'} \tag{2.61}$$

We note that the number of regions above depends only on n and d, and as long as $\mathbf{W}$ is in general position, $\mathcal{R}(\mathbf{W})$ is agnostic to the exact value of $\mathbf{W}$. There is also a corresponding theorem for the number of vertices of a projected zonotope (Z. Zhang et al., 2019, Theorem 3.3).

Counting the Regions of the Restricted Braid Arrangement

The number of regions of $\mathcal{P}\left(\circledast_n, \mathbf{W}\right)$ correspond to the number of argmaxable rankings of categories in MCC. We will not need this result in this thesis, but there is also a closed form solution for computing this (Cover, 1967), see Appendix A.1 for more details.

Summary

Summary

- Decision boundaries of linear classifiers correspond to hyperplane arrangements.
- Outputs of MLC and MCC correspond to regions in hyperplane arrangements.
- $\mathbf{W}$ restricts hyperplane arrangements and kills some regions.
- $\mathbf{W}$ *must* kill regions of hyperplane arrangements.
- $\mathbf{W}$ *may* kill regions of Voronoi tesselations.
- Regions of hyperplane arrangements correspond to intersected orthants.

More Details For the correspondence between hyperplane arrangements and zonotopes, see Ziegler (1995, Theorem 7.16). Restricting hyperplane arrangements to low-rank $\mathbf{W}$ (Ziegler, 1995, Lemma 7.11) must kill regions of the hyperplane arrangement (Cover, 1965). Correspondingly, projecting a zonotope by $\mathbf{W}$ must kill some of its vertices (L. Zhang et al., 2018, Theorem 3.3). While a Voronoi Tesselation is not a hyperplane arrangement, the Voronoi Tesselation is a coarsening of the fan of the braid hyperplane arrangement: combine the regions which correspond to rankings which rank the target category above all others. The region of the Voronoi Tesselation is killed only if all corresponding regions of the braid arrangement are killed, therefore $\mathbf{W}$ may kill regions of the Voronoi Tesselation. The corresponding statement for the simplex comes from identifying the normal fan of the standard simplex as the Voronoi Tesselation.

More on Hyperplane Arrangements Excellent resources to learn more about hyperplane arrangements are Stanley (2004) and Federico Ardila's lectures on polytopes (see Lecture 34 onwards). Connections of the Boolean Arrangement to ML can be found in MacKay (2003, Chapter 40). For those who prefer a more gentle introduction via a hands on approach, Sagemath (Stein et al., 2019) contains implementations of many hyperplane arrangements and functions that we found useful when learning this material.

2.5 Sign Vectors: The Atoms of Argmaxability

To distill the "combinatorial essence", we use the sign function.

—Günter Ziegler

We discussed two representations of outputs, vertices of polytopes (Section 2.3) and regions of hyperplane arrangements (Section 2.4). As we saw, $\mathbf{W}$ comes into the picture as a projection of the vertices (Definition 2.3.4) or the normal vectors (Definition 2.4.10), respectively. As a result of the projection, zonotopes lose vertices and hyperplane arrangements lose regions. We now use a bit of abstraction to distil what matters for argmaxability from both representations: sign vectors. As we will see, the fine-grained linear dependence structure of $\mathbf{W}$ (Section 2.2.3) tells us which sign vectors are lost. By mapping sign vectors onto output representations, we can then explain which outputs are argmaxable for a given bottlenecked classifier, $\mathbf{W}$.

2.5.1 Sign Vectors

In order to understand which outputs become unargmaxable due to a bottlenecked classifier, $\mathbf{W}$, we need to represent outputs in a way that is compatible. The correct granularity for understanding argmaxability is at the level of atoms called **sign vectors**. For our current purposes, we think of sign vectors as a discrete representation for a vector $\mathbf{x} \in \mathbb{R}^n$. We discretise the vector $\mathbf{x}$ by summarising each coordinate using its sign, as defined below.

Sign Vector

Def. 2.5.1. A **sign vector** of length n is: $(\,+\,,\,-\,,\,\mathbf{0}\,)^n$, where we assume the outputs are ordered (we will encode the ordering using a fixed order of colours, e.g. $++++++$, $-0+-+$). We construct sign vectors from vectors in $\mathbb{R}^n$ by quantising them via the $\operatorname{sign}()$ function:

$$\operatorname{sign}(x) = \begin{cases} +\,, & \text{if } x > 0 \\ -\,, & \text{if } x < 0 \\ \mathbf{0}\,, & \text{if } x = 0 \end{cases} \tag{2.62}$$

We will slightly abuse notation and apply the sign function to vectors $\mathbf{x}$, by which we mean that we apply the function element-wise.

Example 11: Sign Vector

$$\mathbf{z} = \begin{bmatrix} 1.5 & -2.1 & 0 & .2 \end{bmatrix}^\top \quad \text{sign}(\mathbf{z}) = \begin{bmatrix} + & - & 0 & + \end{bmatrix}^\top$$

Sign Vectors in Computations We note that an entry of zero in a sign vector, $\mathbf{0}$, is not to be confused with a vector of zeroes - this will be clear from context. We abuse the notation of sign vectors so that we can use them directly in computations when the context is clear. The numerical interpretation is the obvious one, $\mathbf{0}$ is 0 and we interpret $+$ as $+1$ and $-$ as -1. For example, $+-0+$ is equivalent to $(1,-1,0,1)^\top$.

2.5.2 Partial Order on Sign Vectors

By introducing sign vectors we have defined the atoms, i.e. the vocabulary of our discrete space. However, we have not yet defined the structures we can build with these sign vectors. To assist with this, we define a partial order on sign vectors; i.e. we define what the operators $<$ and $>$ mean when applied to sign vectors. The partial order will allow us to refer to groups of sign vectors by specifying a given inequality, e.g. as we will see, $\mathbf{x} \leq 0++$ implies $\mathbf{x} \in \{\, 0++\,,\ +++\,,\ -++\,\}$.

Partial Order on Sign Vectors

Def. 2.5.2. For each element of the sign vector we define $+ > \mathbf{0}$ and $- > \mathbf{0}$ with $+$ and $-$ being incomparable. We break comparisons between sign vectors into element-wise comparisons.

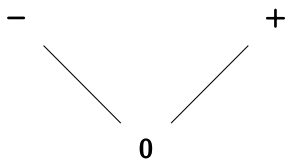


Figure 2.17: Partial Order on Sign Vectors.

For example, given the partial order, we have $0++ < +++$ and $0++ < -++$, but $+++$ and $-++$ are incomparable.

While the above comparisons may seem abstract, they capture the semantics of face inclusion in zonotopes and hyperplane arrangements.

- For zonotopes, the faces consist of vertices, edges, ..., facets and the whole zonotope. The partial order tells us which faces are included in which other faces. E.g. the vertices (0-faces) of an edge (1-face) are included in the edge.
- For hyperplane arrangements, the faces consist of regions (i.e. intersections of halfspaces), intersections of halfspaces restricted to a single hyperplane, ..., the origin [7]. E.g. the regions (d-faces) include rays ($(d-1)$-faces), which are restrictions of halfspaces to a hyperplane.

We illustrate the partial order of sign vectors as interpreted for zonotopes and hyperplane arrangements in Fig. 2.18. As can be seen in Fig. 2.18(a), for the zonotope example we mentioned above, $\mathbf{0}++$ represents the edge that joins the vertices represented by $-++$ and $+++$. As such, our comparison $\mathbf{x} \leq \mathbf{0}++$ encodes that $\mathbf{x}$ matches either the edge or its vertices. The partial order encodes *reverse inclusion* for the faces of the zonotope.

On the other hand, for hyperplane arrangements, as can be seen in Fig. 2.18(b), $\mathbf{0}++$ corresponds to $\mathcal{H}(\mathbf{w}_1) \cap \mathcal{H}^+(\mathbf{w}_2) \cap \mathcal{H}^+(\mathbf{w}_3)$, i.e. the intersection the hyperplane and two halfspaces. As such, our comparison $\mathbf{x} \leq \mathbf{0}++$ encodes $+++ \cup -++ \cup \mathbf{0}++$, i.e. the union of the two regions and their shared boundary. The partial order encodes *inclusion* of the faces of the hyperplane arrangement.

We will use the partial order to define when certain outputs are unargmaxable. In Section 3.5, the sign vectors we will be thinking of will correspond to a) orthants of $\mathbb{R}^n$ or b) vertices of $\square_n$. The partial order will allow us to talk about composing such primitives. In particular, we will use the partial order to define the argmaxable regions in MCC, since they are unions of regions of $\circledast$.

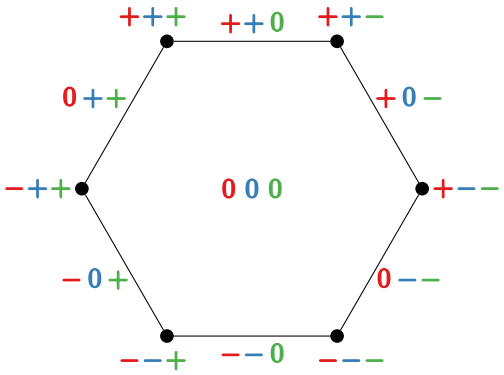
2.5.3 Representable Sign Vectors

Although oriented matroids need some amount of new notation and terminology, there is little magic involved: just don't be scared of names.

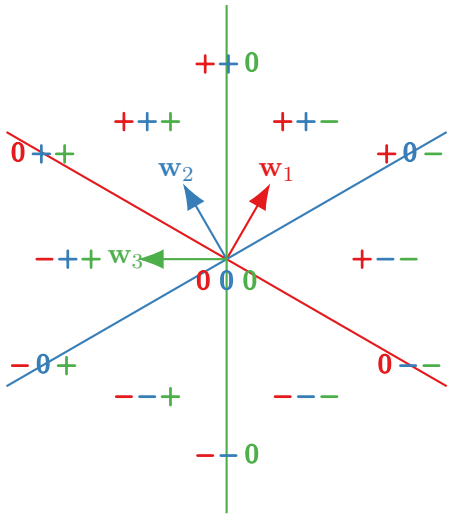
—Günter M. Ziegler

We now get to the core question regarding argmaxability. Given a matrix $\mathbf{W} \in \mathbb{R}^{n \times d}$, $d < n$, which sign vectors can we represent and which ones do we lose? Oriented Matroids (Björner et al., 1999) give us the answer, it is a beautiful theory that distils

[7] We assume the hyperplane arrangement is central, i.e. that all hyperplanes intersect at the origin. This is true for the output layers we discuss when we do not use a bias term.



(a) **Zonotope** defined by $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$. The sign at position i tells us how $\mathbf{w}_i$ contributes to the result, where $+$ means only $\mathbf{w}_i$ contributes, $-$ means only $-\mathbf{w}_i$ contributes, and $\mathbf{0}$ means the whole line segment $(\mathbf{w}_i, -\mathbf{w}_i)$ contributes. Each maximal sign vector is a vertex with sign vectors lower in the partial order representing edges and 0 0 0 representing the whole zonotope.



(b) **Hyperplane Arrangement** defined by the normal vectors $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$. The sign at position i specifies whether we are above, below or on the i^{th} hyperplane in the direction of the normal vector. The maximal sign vectors are intersections of halfspaces, with sign vectors lower in the partial order being halfspace intersections restricted to hyperplanes and 0 0 0 representing the origin.

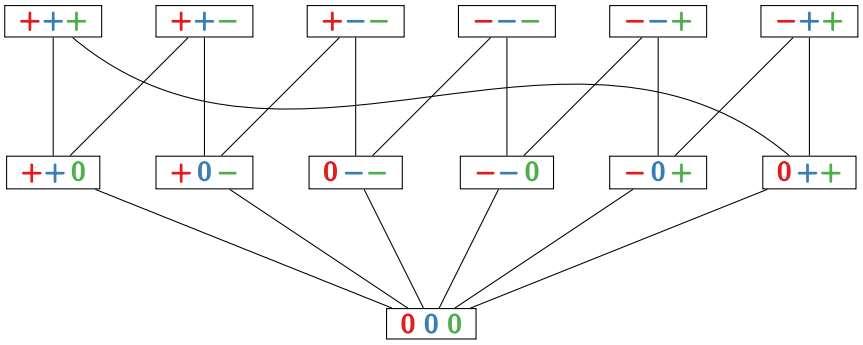


Figure 2.18: Example of partial order on sign vectors constructed by considering vectors $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$. The partial order captures face inclusion for the Zonotope (Fig. 2.18(a)) and the corresponding Hyperplane Arrangement (Fig. 2.18(b)), but for the Zonotope the order is reversed, i.e. higher-dimensional faces are lower in the partial order. E.g. for the Zonotope, the vertices +++ and ++− are included in the edge ++0, and therefore ++0 is lower than the vertices +++ and ++− in the partial order. For the Hyperplane Arrangement, the region +++ includes both ++0 and 0++ (faces of Hyperplane Arrangements include their boundary, see Ziegler (1995, Chapter 7.1)), so +++ is higher in the partial order.

the essence of the linear dependencies in $\mathbf{W}$. It partitions sign vectors into two sets, the vectors and the covectors. For our purposes, it tells us that when we can represent the covectors, we cannot represent the vectors, and vice-versa. Introducing Oriented Matroids in detail is out of scope for this thesis, but we briefly introduce vectors and covectors to answer which outputs are argmaxable/unargmaxable for a given matrix $\mathbf{W}$. In terms of the previous sections, this makes precise our points about zonotopes losing vertices under projections and hyperplane arrangements losing regions under restriction.

Covectors of Oriented Matroids

The **covectors** are the sign vectors which distil the possible output values of $f(\mathbf{x}) = \mathbf{W}\mathbf{x}$, $\mathbf{W} \in \mathbb{R}^{n \times d}$.

Covectors

Def. 2.5.3. The Covectors, $\mathcal{V}^*(\mathbf{W})$, of the Oriented Matroid associated with $\mathbf{W}$ are:

$$\mathcal{V}^*(\mathbf{W}) = \left\{ \text{sign}(\mathbf{z}), \quad \mathbf{z} = \mathbf{W}\mathbf{x} \quad \forall \mathbf{x} \in \mathbb{R}^d \right\} \tag{2.63}$$

As we will discuss in Section 3.3, a bottlenecked classifier computes $\mathbf{z} = \mathbf{W}\mathbf{x}$. As such, the covectors correspond to the atoms that construct argmaxable outputs (see Section 3.4).

Vectors of Oriented Matroids

The **vectors** are the sign vectors that distil the linear dependencies (Definition 2.2.3) in $\mathbf{W} \in \mathbb{R}^{n \times d}$.

Vectors

Def. 2.5.4. The Vectors, $\mathcal{V}(\mathbf{W})$, of the Oriented Matroid associated with $\mathbf{W}$ are:

$$\mathcal{V}(\mathbf{W}) = \left\{ \text{sign}(\mathbf{z}), \quad \mathbf{z} \in \mathbb{R}^n : \mathbf{W}^\top \mathbf{z} = \mathbf{0} \right\} \tag{2.64}$$

These correspond to atoms that are unargmaxable. They are the regions of the hyperplane arrangement that we lost under restriction and the vertices of the zonotopes that we lost under projection.

To summarise, Oriented Matroids provide a framework for answering one of the main questions of our thesis: Which outputs are (un)argmaxable for $\mathbf{W}$? The answer

is: the (vectors) covectors of the Oriented Matroid defined by $\mathbf{W}$.

Repercussions of Connection to Oriented Matroids

What are the repercussions of this connection of argmaxability to Oriented Matroids?

By linking argmaxability to Oriented Matroids we can take advantage of any algorithm developments in Oriented Matroids and apply them to our problem. Morever, we inherit hardness results from Oriented Matroids which we can translate to the problem of argmaxability. For example, an important problem in Neurosymbolic AI is to guarantee that outputs of interest can be predicted (or, conversely, that invalid outputs cannot be predicted) (Ahmed et al., 2022; Giunchiglia et al., 2022; Krieken et al., 2024; Manhaeve et al., 2018). We could express such outputs as covectors of an Oriented Matroid and ask: Given the set of covectors, what is a $\mathbf{W}$ for which these are indeed covectors of the underlying Oriented Matroid? Through the connection to Oriented Matroids, we validate our suspicion that this problem is hard in general. This problem is known as the realisability problem for Oriented Matroids, and we know it is NP-Hard (Mnev, 1988; Shor, 1990).

Another insight obtained from the connection, is that we can detect unargmaxable outputs by computing linear dependencies in $\mathbf{W} \in \mathbb{R}^{n \times d}$, $d < n$. However, even if $\mathbf{W}$ is in general position, for large n and d there are too many linear dependencies (i.e. $\binom{n}{d+1}$ of them) for us to exhaustively enumerate them. Moreover, we are not aware of a way to guide such a search for outputs we may want to check.[8] In Chapter 4 we will use a targeted approach. For a target output, we check whether it is argmaxable by detecting whether the corresponding region in the hyperplane arrangement survives the restriction by $\mathbf{W}$. Maybe unsurprisingly, as we will see, we can solve this problem via Linear Programming.

So, given $\mathbf{W}$, we can check whether a sign vector is a covector. But how do the covectors map to outputs? In other words, how do we know which outputs are argmaxable? We now show how to decompose output representations into **sign vectors** so that we can decide whether an output is argmaxable based on whether certain sign vectors are covectors. We start from subsets since they are the easiest to explain.

[8] Although, if we do try to guide such a search, we may be reinventing the simplex algorithm from Linear Programming.

2.5.4 Subsets as Sign Vectors $\mathbf{y}^{\square}$

Consider a set of n labels, Y and form a subset $S \subseteq Y$. To encode a subset as a sign vector, set $y_i^{\square} = \mathbf{+}$ if the element is in the subset, and $y_i^{\square} = \mathbf{-}$ if it is not.

$$\mathbf{y}^{\square} = \begin{bmatrix} s(1) \\ s(2) \\ \vdots \\ s(n) \end{bmatrix}, \quad i \in [n], \quad s(i) = \begin{cases} \mathbf{+} & \text{if } Y_i \in S \\ \mathbf{-} & \text{if } Y_i \notin S \end{cases} \tag{2.65}$$

2.5.5 Rankings as Sign Vectors $\mathbf{y}^{\hexagon}$

We encode a ranking as a sign vector by considering all pairwise comparisons of elements in a particular order. Consider a set of n categories. Let $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ be the permutation that gives our target ranking, r. The sign vector is $\mathbf{y}^{\hexagon}$ is:

$$\mathbf{y}^{\hexagon} = \begin{bmatrix} s(1,2) \\ s(1,3) \\ \vdots \\ s(2,3) \\ \vdots \\ s(i,j) \end{bmatrix}, \quad i,j \in [n], \quad i < j, \quad s(i,j) = \begin{cases} \mathbf{+} & \text{if } \sigma_i > \sigma_j \\ \mathbf{-} & \text{else} \end{cases} \tag{2.66}$$

2.5.6 Categories as Sign Vectors $\mathbf{y}^{\triangle}$

Consider a set of n categories and a target category c. The sign vector for a category is given by all rankings which rank the category with index c above all other categories:

$$\mathbf{y}^{\triangle} = \begin{bmatrix} s(1,2) \\ s(1,3) \\ \vdots \\ s(2,3) \\ \vdots \\ s(i,j) \end{bmatrix}, \quad i,j \in [n], \quad i < j, \quad s(i,j) = \begin{cases} \mathbf{+} & \text{if } i = c \\ \mathbf{-} & \text{if } j = c \\ \mathbf{0} & \text{else} \end{cases} \tag{2.67}$$

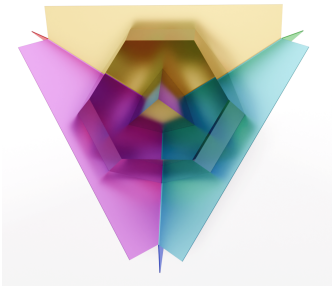


Figure 2.19: Correspondence between the 6 rankings (vertices of $\hexagon_3$) and the category (represented by the vertex of $\triangle_3$).

2.5.7 Atomic and Compound Outputs

In Section 3.5, we show that a consequence of a bottlenecked classifier is that some outputs *may* be unargmaxable while others *must* be unargmaxable. To explain this difference, we introduce two families of outputs.

Atomic Outputs We use atomic here to mean in the indivisible sense: sign vectors cannot be further broken down.

Atomic Output

Def. 2.5.5. An **atomic output** corresponds to a single sign vector.

Since for any bottlenecked classifier some sign vectors are unargmaxable, some atomic outputs *must* be unargmaxable. As such, label assignments and rankings are atomic outputs.

Compound Outputs On the other hand, compound outputs are built from multiple sign vectors.

Compound Output

Def. 2.5.6. A **compound output** comprises multiple sign vectors.

For example, a category in MCC consists of all rankings that rank the target category above all others. This means that compound outputs have redundancy when it comes to unargmaxability: we would need all sign vectors comprising this output to be unargmaxable for the compound output to be unargmaxable. We therefore say that for a low-rank classifier some compound outputs *may* be unargmaxable.

2.6 Probability Distributions

Ok, so we have an answer of 1000 ± 30 *thanks to Mr. Binomial, the inventor of the distribution that bears his name.*

— David J. C. Mackay, Introduction to Information Theory

Here we define the discrete probability distributions we will need in order to model MCC and MLC. A more explicit mapping can be seen below:

- Binary Classification (MLC, $n = 1$ or MCC, $n = 2$) $\rightarrow$ Bernoulli distribution
- MLC with n labels $\rightarrow$ Joint distribution of n Bernoulli distributions
- MCC with $n > 2$ categories $\rightarrow$ Categorical distribution over n categories

In Chapter 4 we will use DNNs to estimate the parameters from data.

We assume the reader is comfortable with random variables and probability theory and refer the reader to Bertsekas et al. (2008) for a refresher. To define a discrete random variable we need two things: a) the set of possible values this random variable can take, i.e. the **support** of the distribution and b) the probability of each value, i.e. the **Probability Mass Function (PMF)**.

2.6.1 Bernoulli Distribution

Support We use a Bernoulli random variable to model an event that has two possible outcomes. As such, the support is $\{+, -\}$, say success ($+$) and failure ($-$).

PMF In order to define the PMF for the Bernoulli, we need to define a single parameter, θ, the probability of success.

$$P(y;\theta) = \begin{cases} \theta & \text{if } y = + \\ 1-\theta & \text{if } y = - \end{cases} \tag{2.68}$$

2.6.2 Distribution of Multiple Bernoulli

Support For the support we combine n Bernoulli random variables $y_1, y_2, \ldots, y_n$. Since each y_i takes a value from $\{+, -\}$, the support is $\{+, -\}^n$.

PMF For our analysis in Section 3.3.3, we will assume the y_i are independent [9]. We therefore only need n parameters. We concatenate the individual parameters into a vector, i.e. $\boldsymbol{\theta} = [\theta_1; \theta_2; \ldots; \theta_n]$. Given that the y_i are independent, we get the following joint PMF:

$$P(\mathbf{y}; \boldsymbol{\theta}) = \prod_i \theta_i [y_i = \text{+}] + (1 - \theta_i) [y_i = \text{−}] \tag{2.69}$$

where [] is the Iverson bracket (see Section 0.2).

Geometric Interpretation The parameters $\boldsymbol{\theta}$ of n Bernoullis live in $\square_n$.[10]

2.6.3 Categorical Distribution

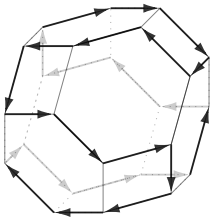
Support In MCC, our goal is to choose from a set of categories $\{c_1, c_2, \ldots, c_n\}$.

PMF We define the PMF of a categorical distribution over n outcomes via $n-1$ parameters, $\boldsymbol{\theta}$:

$$P(y; \boldsymbol{\theta}) = \begin{cases} \theta_1 & \text{if } y = c_1 \\ \theta_2 & \text{if } y = c_2 \\ \vdots & \\ \theta_{n-1} & \text{if } y = c_{n-1} \\ 1 - \sum_{i=1}^{n-1} \theta_i & \text{if } y = c_n \end{cases} \tag{2.70}$$

Geometric Interpretation The parameters $\boldsymbol{\theta}$ of a categorical distribution live in $\triangle_{n-1}$, also known as the probability simplex.

[9] In the neural network models, this will be conditional independence given the input features.
[10] The cube is offset by $\mathbf{1}$ and scaled by $\frac{1}{2}$



3 Argmaxability in Neural Networks

In this chapter, we reframe argmaxability in the context of Deep Neural Networks (DNNs). To see the connection, we decompose DNNs into an encoder and output layer (Section 3.1) and introduce the output layers used for MCC and MLC (Section 3.2). We identify the culprit of unargmaxable outputs, the matrix $\mathbf{W}$, as the parameters of the output layer of DNNs. As we will see, $\mathbf{W}$ arises when we make the output layer a bottlenecked classifier (Section 3.3). However, the constraints imposed by $\mathbf{W}$ differ depending on whether $\mathbf{W}$ is used for MCC or MLC. We therefore formalise the (un)argmaxability problem for MCC and MLC (Section 3.4) and explain under which scenarios we *may* have unargmaxable outputs and when we *must* have unargmaxable outputs (Section 3.5).

3.1 Deep Neural Networks (DNNs)

3.1.1 Encoder and Output Layer Perspective

In this thesis, we break down DNNs conceptually into two parts: i) a **feature encoder** and ii) an **output layer** (see Fig. 3.1). We focus on models that have log-linear output layers. We think of these as log-linear classifiers applied to dense feature vectors produced by DNNs. This view unifies 2000s ML with DNNs: DNNs benefit from replacing feature engineering with an expressive feature encoder (Goldberg, 2015) that is learned in tandem with the output layer.

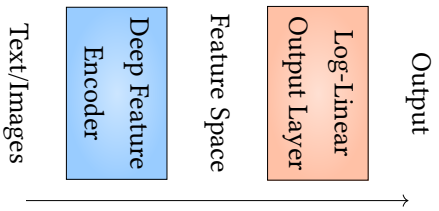


Figure 3.1: DNNs comprise an expressive encoder followed by a log-linear classifier.

Encoder

When we say **encoder** in this thesis, we mean feature encoder. The feature encoder is a function, typically a DNN, that maps any input data, e.g. some text or an image, to a feature vector. For example, consider BERT (Devlin et al., 2019) and ResNet (He et al., 2016) as feature encoders for text and images, respectively. Feature vectors are outputs of the encoder and inputs to the output layer. In what follows, *we think of feature vectors as inputs, since we will focus on the output layer.*

Idealised Encoder Assumption

In this thesis we do not analyse the feature encoder; for our purposes we assume it is fully expressive; i.e. if the encoder produces d features, we assume it can yield any feature vector in $\mathbb{R}^d$. We study constraints on predictions that arise solely from the output layer. As such, *the outputs that we find to be unargmaxable in our work are unargmaxable irrespective of the choice of feature encoder.*

It is easy to miss that DNNs can be shackled by constraints present in log-linear layers. This is because DNNs are associated with highly expressive non-linear models. Feature encoders now comprise multiple layers interspersed with non-linearities, they have a huge number of parameters and in the case of sequence prediction, can encode very long contexts. However, *even for LLMs, the output layer is still a bottlenecked log-linear classifier* (see Table 1.1). It is important to analyse the output layer, since any constraints it imposes apply to the whole model, irrespective of the complexity of the feature encoder. From now on we focus on this log-linear output layer.

3.2 Log-Linear Classifiers

Linear classifiers score each output as a linear function of their inputs. Herein, we focus on log-linear classifiers. These are linear classifiers in log space, i.e. linear scoring functions followed by an activation function which has an exponential form.

A log-linear classifier comprises two parts:

1. **a parametrisation** (Section 3.2.1) and

2. **an activation function** (Sections 3.2.2 and 3.2.3).

3.2.1 Parametrisation

Consider a log-linear classifier with d inputs and n outputs, i.e. one that maps input feature vectors $\mathbf{x} \in \mathbb{R}^d$ to outputs $\mathbf{z} \in \mathbb{R}^n$. We represent the parameters of the log-linear classifier compactly by using a matrix $\mathbf{W} \in \mathbb{R}^{n \times d}$. As such, $\mathbf{W}$ defines the linear map $f : \mathbb{R}^d \to \mathbb{R}^n$, $\mathbf{z} = \mathbf{W}\mathbf{x}$. We say that a log-linear classifier is **fully-parametrised** if it has $n = d$, i.e. $\mathbf{W}$ is a square matrix. We focus on fully-parametrised classifiers for now and postpone discussion of **bottlenecked** parametrisations that have $d < n$ to Section 3.3.1.

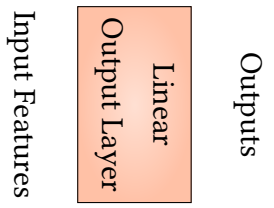


Figure 3.2: We focus on the output layer.

Now that we have elaborated on the parametrisation of log-linear classifiers, we introduce the activation functions that follow the linear map to complete the classifier. These activations differ depending on the type of classifier. For MCC, the outputs are mutually exclusive, so we use a softmax activation function to form a softmax layer. For MLC, each label can be active independently, so we apply the sigmoid activation function element-wise to form a sigmoid layer. [1]

3.2.2 Softmax Layer: Multi-Class Classification

A linear layer with a softmax activation is used when we want to do MCC, i.e. we want to score n mutually exclusive categories. The softmax[2] operation squashes the activations of an unconstrained neural network such that they form a categorical probability distribution (Section 2.6.3). We call the inputs to softmax, i.e. the vector $\mathbf{z} \in \mathbb{R}^n$, the **logits**. In MCC, we want to estimate the parameters of a categorical distribution over n outputs. Softmax does this by mapping the logits $\mathbf{z} \in \mathbb{R}^n$ to the parameters of the categorical. As we saw in (see Section 2.6.3), this is the simplex Δ_{n-1}, so in keeping with our mnemonic shape notation, we define softmax as $\sigma_{\Delta}()$.

[1] We apply the function to each element of the input vector.

[2] We note that softmax is a misnomer. Softargmax would be semantically correct, since the operator corresponds to an argmax which is smoothed via the entropy (Blondel et al., 2019).

Softmax, $\sigma_\triangle()$

Def. 3.2.1. We can describe the output at element i of softmax as a function:

$$\sigma_\triangle(\mathbf{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{3.1}$$

where e is the base of the natural logarithm.

Partition Function $\mathcal{Z}$ The sum in the denominator of softmax is the normalisation constant needed to scale the outputs into a probability distribution. We call the denominator the **partition function** and represent it as $\mathcal{Z} = \sum_j e^{z_j}$.

Geometric Interpretation Geometrically, we can think of softmax as mapping the logits $\mathbb{R}^n \to \operatorname{relint}(\triangle_{n-1})$, where $\triangle_{n-1}$ is the probability simplex (Definition 2.3.1), see also Amos (2019, Section 2.4.4). As we saw in Section 2.6.3, the resulting categorical distributions live in the simplex, we illustrate it in Fig. 3.3.

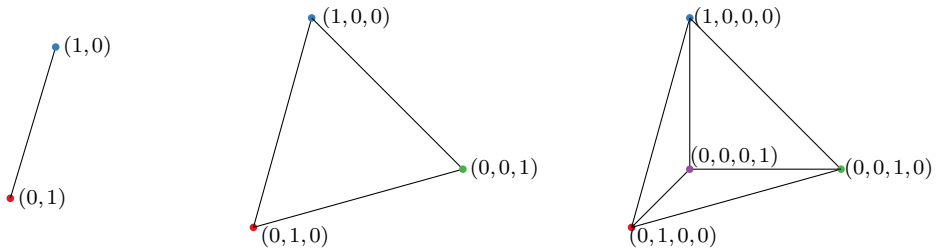


Figure 3.3: Softmax, $\sigma_\triangle()$, maps logits in $\mathbb{R}^n$ to the interior of $\triangle_{n-1}$. Above: $\triangle_n$ for $n = 1, 2, 3$: from left to right. Each vertex of the simplex represents a category. We colour-code the vertices to distinguish between the categories they represent.

We now complete the description of our softmax layer by introducing notation for it in terms of its parameters $\mathbf{W} \in \mathbb{R}^{n \times d}$ and an optional bias term $\mathbf{b} \in \mathbb{R}^n$.

Softmax Layer $\mathrm{BSL}_\triangle$

Def. 3.2.2. A **softmax layer** with n categories and d input features is a map, $\mathrm{BSL}_\triangle() : \mathbb{R}^d \to \triangle_{n-1}$, parametrised by $\mathbf{W} \in \mathbb{R}^{n \times d}$ and optionally a bias term $\mathbf{b} \in \mathbb{R}^n$. We write:

$$P(\mathbf{y} \mid \mathbf{x}) = \mathrm{BSL}_\triangle(\mathbf{x}; \mathbf{W}, \mathbf{b}) \tag{3.2}$$

$$= \sigma_\triangle(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{3.3}$$

We use BSL as short for bottlenecked classifiers, since in all parts of the thesis after (Section 3.3) we will be discussing either Bottlenecked Softmax Layers or Bottlenecked Sigmoid Layers; both BSLs. When we need to distinguish between the two, we will include a subscript, e.g. we use $\mathrm{BSL}_{\triangle}$ for a bottlenecked softmax.

Next, we define properties of softmax we will need in order to

1. elaborate on the expressivity of a softmax layer.
2. define (un)argmaxable outputs and derive how to detect them for a $\mathrm{BSL}_{\triangle}$.

Softmax Properties

Softmax is Invariant to Constant Offset

Property 3.1. Adding any constant to the logits of softmax does not change the obtained probabilities. That is, for any constant c, we have:

$$\sigma_{\triangle}(\mathbf{W}\mathbf{x}+c\mathbf{1}) = \sigma_{\triangle}(\mathbf{W}\mathbf{x}) \tag{3.4}$$

Derivation. Property 3.1 holds because the constant offset cancels, as we elaborate below for $i \in [n]$:

$$\sigma_{\triangle}(\mathbf{W}\mathbf{x}+c\mathbf{1})_i = \frac{e^{\mathbf{w}_i^\top \mathbf{x}+c}}{\sum_j e^{\mathbf{w}_j^\top \mathbf{x}+c}} \tag{3.5}$$

$$= \frac{\cancel{e^{c}} e^{\mathbf{w}_i^\top \mathbf{x}}}{\cancel{e^{c}} \sum_j e^{\mathbf{w}_j^\top \mathbf{x}}} \tag{3.6}$$

$$= \sigma_{\triangle}(\mathbf{W}\mathbf{x})_i \tag{3.7}$$

□

Softmax has $n-1$ Free Variables

Property 3.2. Consider a fully-parametrised softmax classifier over n categories. Intuitively, although a softmax classifier has n outputs, the fact that the probabilities must sum to one means that we only need $n-1$ parameters. Parametrisation-wise, we only need $n-1$ columns in $\mathbf{W}$, i.e. $\mathbf{W} \in \mathbb{R}^{n\times(n-1)}$.

Derivation. We can show the above as a consequence of Property 3.1. To do so, our goal is to isolate the constant offset contribution which gets cancelled by softmax. We

can do so if we pick an orthonormal basis for $\mathbb{R}^n$ (Definition 2.2.7) for which one of the columns is constant, i.e. $\frac{1}{\sqrt{n}}\mathbf{1}_n$, as below:

$$\mathbf{B} = \left[\frac{1}{\sqrt{n}}\mathbf{1}_n \; \mathbf{B}_{:,-1}\right] \in \mathbb{R}^{n\times n} \tag{3.8}$$

where the brackets denote matrix concatenation and $\mathbf{B}_{:,-1}$ is the remainder of matrix $\mathbf{B}$ after dropping the first column (see Section 0.2 for details on notation). We have:

$$\sigma_\triangle(\mathbf{Bx}) = \sigma_\triangle\left(\mathbf{B}_{:,-1}\mathbf{x}_{2:} + \frac{x_1}{\sqrt{n}}\mathbf{1}\right) = \sigma_\triangle\left(\mathbf{B}_{:,-1}\mathbf{x}_{2:}\right) \tag{3.9}$$

where the last step follows from Property 3.1 and tells us that dropping the constant column in $\mathbf{B}$ does not change the result. We note that the term $\mathbf{B}_{:,-1}\mathbf{x}_{2:}$ cannot have any contribution to the constant, because the columns of $\mathbf{B}_{:,-1}$ are orthogonal to $\mathbf{1}_n$. By a change of basis argument, we see that $\mathbf{W}$ need not have more than $n-1$ columns. □

Softmax is Invariant to Column Mean Centering of $\mathbf{W}$

Property 3.3.

$$\sigma_\triangle\left(\left(\mathbf{W} - \overline{\mathbf{W}}_{\text{col}}\right)\mathbf{x}\right) = \sigma_\triangle(\mathbf{Wx}) \tag{3.10}$$

where $\overline{\mathbf{W}}_{\text{col}} = \frac{1}{n}\mathbf{1}_n\mathbf{1}_n^\top\mathbf{W}$ is the matrix of column means of $\mathbf{W}$.

We will need this property to define the softmax bottleneck (Definition 3.3.2).

Derivation. To see why Property 3.3 holds, let us first introduce three results we will need for our main point. We have:

$$\overline{\mathbf{W}}_{\text{col}} = \frac{1}{n}\mathbf{1}_n\mathbf{1}_n^\top\mathbf{W} \tag{3.11}$$

We can therefore write the column mean centered version of $\mathbf{W}$, $\mathbf{W}_c$, as:

$$\mathbf{W}_c = \mathbf{W} - \overline{\mathbf{W}}_{\text{col}} \quad \iff \tag{3.12}$$

$$\mathbf{W} = \overline{\mathbf{W}}_{\text{col}} + \mathbf{W}_c \tag{3.13}$$

We can therefore decompose $\mathbf{Wx}$ as:

$$\mathbf{Wx} = \left(\overline{\mathbf{W}}_{\text{col}} + \mathbf{W}_c\right)\mathbf{x} \tag{3.14}$$

$$= \overline{\mathbf{W}}_{\text{col}}\mathbf{x} + \mathbf{W}_c\mathbf{x} \tag{3.15}$$

Now, the idea is to isolate the contribution of $\mathbf{Wx}$ to the constant offset that will get cancelled by the softmax operation, as per Property 3.1. To this end, we reformulate our softmax layer, $\sigma_{\triangle}(\mathbf{Wx})$, in terms of an orthonormal basis, $\mathbf{B}$, which includes the constant offset direction as the first column and where $\mathbf{B}_{:,-1}$ is the remainder of matrix $\mathbf{B}$ after dropping the first column:

$$\mathbf{B} = \left[\frac{1}{\sqrt{n}}\mathbf{1}_n \; \mathbf{B}_{:,-1}\right] \in \mathbb{R}^{n\times n} \tag{3.16}$$

We have:

$$\begin{aligned}
\mathbf{Wx} &= \mathbf{B}\mathbf{B}^\top\mathbf{Wx} && \mathbf{B} \text{ is orthonormal: } \mathbf{B}\mathbf{B}^\top = \mathbf{I}_n && (3.17)\\
&= \left(\frac{1}{\sqrt{n}}\mathbf{1}_n\frac{1}{\sqrt{n}}\mathbf{1}_n^\top + \mathbf{B}_{:,-1}\mathbf{B}_{:,-1}^\top\right)\mathbf{Wx} && \mathbf{B}\mathbf{B}^\top = \sum_{i=1}^{n}\mathbf{b}_{\cdot i}\mathbf{b}_i^\top && (3.18)\\
&= \left(\frac{1}{n}\mathbf{1}_n\mathbf{1}_n^\top\mathbf{W}\right)\mathbf{x} + \mathbf{B}_{:,-1}\mathbf{B}_{:,-1}^\top\mathbf{Wx} && \text{from Eq. (3.11)} && (3.19)\\
&= \overline{\mathbf{W}}_{\text{col}}\mathbf{x} + \mathbf{B}_{:,-1}\mathbf{B}_{:,-1}^\top\mathbf{Wx} && && (3.20)
\end{aligned}$$

Therefore, by combining Eq. (3.15) and Eq. (3.20), we have:

$$\begin{aligned}
\overline{\mathbf{W}}_{\text{col}}\mathbf{x} + \mathbf{W}_c\mathbf{x} &= \overline{\mathbf{W}}_{\text{col}}\mathbf{x} + \mathbf{B}_{:,-1}\mathbf{B}_{:,-1}^\top\mathbf{Wx} \quad \Longleftrightarrow && (3.21)\\
\mathbf{W}_c\mathbf{x} &= \mathbf{B}_{:,-1}\mathbf{B}_{:,-1}^\top\mathbf{Wx} && (3.22)
\end{aligned}$$

We can now show our result:

$$\begin{aligned}
\sigma_{\triangle}(\mathbf{Wx}) &= \sigma_{\triangle}\left(\frac{1}{n}\mathbf{1}_n\underbrace{\mathbf{1}_n^\top\mathbf{Wx}}_{\text{scalar}} + \mathbf{B}_{:,-1}\mathbf{B}_{:,-1}^\top\mathbf{Wx}\right) && \text{from Eq. (3.19)} && (3.23)\\
&= \sigma_{\triangle}\left(\mathbf{B}_{:,-1}\mathbf{B}_{:,-1}^\top\mathbf{Wx}\right) && \text{apply Property 3.1} && (3.24)\\
&= \sigma_{\triangle}(\mathbf{W}_c\mathbf{x}) && \text{from Eq. (3.22)} && (3.25)\\
&= \sigma_{\triangle}\left(\left(\mathbf{W} - \overline{\mathbf{W}}_{\text{col}}\right)\mathbf{x}\right) && \text{from Eq. (3.12)} && (3.26)
\end{aligned}$$

□

Softmax is an Order Preserving Map As we have seen, the output probabilities are formed by exponentiating the logits and renormalising. Therefore, softmax is an order preserving map: if its arguments are ranked in order of magnitude, the softmax probabilities also have this ranking. We will use this property in Section 3.4.

Order Preserving Map

Def. 3.2.3. A map $f : \mathbf{z} \in \mathbb{R}^n \to \mathbb{R}^n$ is **order preserving** if:

$$z_1 \leq z_2 \leq \ldots \leq z_n \iff f(\mathbf{z})_1 \leq f(\mathbf{z})_2 \leq \ldots \leq f(\mathbf{z})_n \tag{3.27}$$

Softmax is an Order Preserving Map

Property 3.4.

$$\sigma_{\triangle}(\mathbf{z})_a \leq \sigma_{\triangle}(\mathbf{z})_b \leq \ldots \leq \sigma_{\triangle}(\mathbf{z})_n \iff \text{Eq. (3.1)} \tag{3.28}$$

$$\frac{e^{z_a}}{\mathcal{Z}} \leq \frac{e^{z_b}}{\mathcal{Z}} \leq \ldots \leq \frac{e^{z_n}}{\mathcal{Z}} \iff \mathcal{Z} > 0 \tag{3.29}$$

$$e^{z_a} \leq e^{z_b} \leq \ldots \leq e^{z_n} \iff \log \text{ is increasing } f \tag{3.30}$$

$$z_a \leq z_b \leq \ldots \leq z_n \tag{3.31}$$

For a more general derivation, see Blondel et al. (2019, Proposition 1, Point 2).

Propagating Argmax through Softmax We now use the above property to propagate argmax (Definition 3.4.1) through softmax to its arguments.

$$\operatorname*{argmax}_{y \in \mathcal{Y}} \sigma_{\triangle}(\mathbf{z}) = \operatorname*{argmax}_{y \in \mathcal{Y}} \mathbf{z} \tag{3.32}$$

3.2.3 Sigmoid Layer: Multi-Label Classification

A linear layer with a sigmoid activation is used for binary or MLC, i.e. when we want to score n labels which can be active independently. The sigmoid function squashes each individual output of a neural network such that it forms a probability distribution over two possible outcomes.

Sigmoid, $\sigma_{\square}()$

Def. 3.2.4. The **sigmoid** function is:

$$\sigma_{\square}(z) = \frac{1}{1 + e^{-z}} \tag{3.33}$$

In this case, the output for each label is the parameter of a Bernoulli distribution. We will think of the whole layer as predicting the parameters of n independent Bernoulli distributions (see Section 2.6.2).

Geometric Interpretation Geometrically, we can think of the sigmoid functions of the sigmoid classifier as mapping the logits $\mathbb{R}^n \to \text{relint}\,(\square_n)$, see also Amos (2019, Section 2.4.4). We illustrate this mapping in Fig. 3.4. We note that $\square_n$ is offset by $\mathbf{1}$ and scaled by $\frac{1}{2}$.

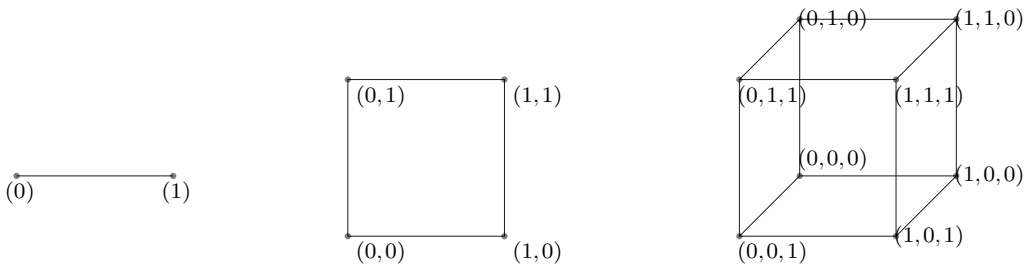


Figure 3.4: Applying the sigmoid function, $\sigma_{\square}()$, element-wise to logits in $\mathbb{R}^n$ projects them to the interior of $\square_n$, as we saw in Section 2.6.2. We illustrate $\square_n$ for $n = 1, 2, 3$.

We now complete the description of our sigmoid layer by introducing notation for it in terms of its parameters $\mathbf{W} \in \mathbb{R}^{n \times d}$ and an optional bias term $\mathbf{b} \in \mathbb{R}^n$.

Sigmoid Layer $\text{BSL}_{\square}$

Def. 3.2.5. A **sigmoid layer** with n labels and d input features is a map, $\text{BSL}_{\square}() : \mathbb{R}^d \to \square_n$, parametrised by $\mathbf{W} \in \mathbb{R}^{n \times d}$ and optionally a bias term $\mathbf{b} \in \mathbb{R}^n$. We write:

$$P(\mathbf{y} \mid \mathbf{x}) = \text{BSL}_{\square}(\mathbf{x}; \mathbf{W}, \mathbf{b}) \tag{3.34}$$

$$= \sigma_{\square}(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{3.35}$$

When we want to talk about a general bottlenecked classifier, we will say BSL, without specifying the subscript.

$\text{BSL}_{\square}$s are used as output layers in many neural MLC models. Examples include MLC problems such as fine-grained entity typing (Choi et al., 2018), protein function prediction (Kulmanov et al., 2019), clinical coding (Mullenbach et al., 2018) and multi-label image classification (Baruch et al., 2020).

Sigmoid Properties

Sigmoid is an Order Preserving Map

Property 3.5. A sigmoid layer is also an order preserving map, like softmax (see Eq. (3.28)). This is because the sigmoid function is strictly increasing, i.e.

$$x_1 \leq x_2 \iff \sigma_{\square}(x_1) \leq \sigma_{\square}(x_2) \tag{3.36}$$

and a sigmoid layer applies a sigmoid activation elementwise to each output.

To summarise, we have discussed the activation functions that together with the parametrisation, form complete MCC and MLC classifiers. We note that despite these activation functions being non-linear, the decision boundaries of the classifier, i.e. the locations in feature space where the decision changes, are linear. As such, their decisions can be analysed as we saw in Section 2.4. We now briefly discuss the expressivity of the parametrisation of these classifiers.

3.2.4 Expressivity of Fully-Parametrised Classifiers

As we will see in Section 3.3.1, some parametrisations lead to classifiers that are less expressive. But less expressive compared to what? To make this precise, we define the concept of a fully-expressive parametrisation. When we say that a classifier has reduced expressivity, we are implicitly comparing it to a fully-expressive parametrisation.

The ideas we need build on linear algebra concepts we defined in Section 2.2. We begin with the idea of a reachable output.

Reachable Output

Def. 3.2.6. An output $\mathbf{z}$ is reachable for a matrix $\mathbf{W}$ if:

$$\exists \mathbf{x} : \mathbf{z} = \mathbf{W}\mathbf{x} \tag{3.37}$$

i.e. $\mathbf{z}$ is reachable if the system of linear equations $\mathbf{W}\mathbf{x} = \mathbf{z}$ has at least one solution. This is true when $\mathbf{z} \in \text{span}(\mathbf{W})$ (Definition 2.2.5).

We use reachability to define what we mean by a fully-expressive parametrisation.

Fully-Expressive Parametrisation

Def. 3.2.7. We say that a parametrisation with n outputs is fully-expressive if all outputs in $\mathbb{R}^n$ are reachable, i.e. $\text{rank}(\mathbf{W}) = n$ (Definition 2.2.10).

We note that it is insufficient for a classifier to be fully-parametrised for it to be fully-expressive, as the matrix we learn via gradient descent may have rank less than n. While the tools we develop in this thesis can be used to study the low-rank case, we focus on bottlenecked parametrisations which are guaranteed to reduce the expressivity of our classifier.

3.3 Bottlenecked Classifiers (BSLs)

Bottlenecked Classifiers (BSLs) are log-linear classifiers that have more outputs than inputs (Definition 1.1.1). We call such classifiers bottlenecked because if we illustrate the map from fewer inputs to more outputs we get a bottleneck shape, as shown in Fig. 3.5.

Motivation BSLs are generally desirable, since they have a reduced number of trainable parameters and are computationally more efficient than fully-parametrised log-linear classifiers. However, while desirable, as we make the bottleneck narrower, we restrict the class of functions the classifier can represent, thus reducing its expressivity, as was highlighted in (Yang et al., 2018). In this thesis, we quantify this loss of expressivity from a discrete perspective by defining unargmaxable outputs: outputs that cannot be produced by this restricted family of functions. Understanding what outputs are argmaxable for BSLs is important, not only because such classifiers are widely used in ML, but more importantly because they are the last layer of many of the latest models, such as LLMs.

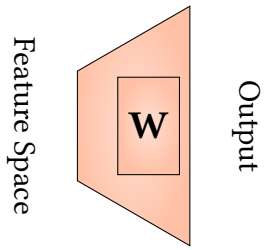


Figure 3.5: We focus on the output layer parametrisation, $\mathbf{W}$, which is often bottlenecked (a tall matrix).

In what follows we will see two examples of BSLs: the $\mathrm{BSL}_{\triangle}$, which is used in MCC tasks and is the output layer of LLMs (Carlini et al., 2024; Devlin et al., 2019; Finlayson et al., 2024; Groeneveld et al., 2024; Touvron et al., 2023), and the $\mathrm{BSL}_{\square}$, which is used in MLC tasks such as Clinical Coding (Mullenbach et al., 2018). Below we elaborate on the parametrisation of BSLs (Section 3.3.1).

3.3.1 Bottlenecked Parametrisation

We say that a parametrisation of a log-linear layer is bottlenecked when $\mathbf{W}$ has more rows that columns, i.e. the classifier has more outputs than inputs. More concretely, bottlenecked output layers compute $\mathbf{z} = \mathbf{W}\mathbf{x}$ where $\mathbf{x} \in \mathbb{R}^d$ is the input, $\mathbf{z} \in \mathbb{R}^n$ is the output and $\mathbf{W} \in \mathbb{R}^{n \times d}$, $d < n$. Importantly, bottlenecked parametrisations necessarily restrict the rank of the matrix $\mathbf{W}$ to be at most d. As a result, a bottlenecked classifier has reduced expressivity compared to a fully-expressive classifier.

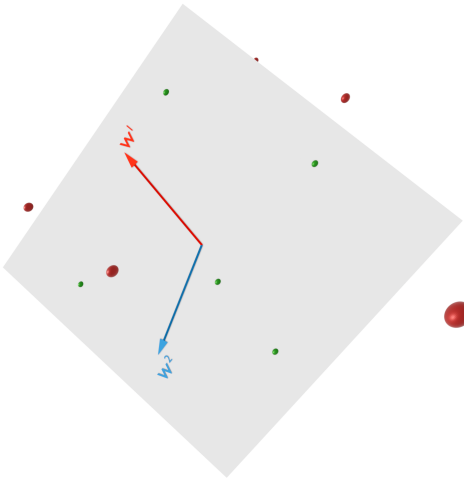


Figure 3.6: Output space of a bottlenecked classifier with $d = 2$ and $n = 3$. Outputs $\mathbf{z}$ are restricted to be on $\text{span}(\mathbf{W})$, the grey 2D subspace in 3D. The example green points on the subspace are reachable, but the red ones are not. We make points large and truncate the subspace to a box to aid visualisation.

We illustrate the reduced expressivity of a bottlenecked parametrisation in Fig. 3.6. A fully-expressive parametrisation would be able to represent any point in $\mathbb{R}^n$. On the other hand, the outputs $\mathbf{z}$ are constrained to a d-dimensional subspace of $\mathbb{R}^n$, i.e. $\mathbf{z} \in \text{span}(\mathbf{W})$. This makes most outputs unreachable (Definition 3.2.6) (see example points in red). The loss in expressivity increases as we make d smaller, e.g. if we had $d = 1$ in Fig. 3.6, $\mathbf{z}$ would be constrained to a line in 3D. We can quantify the loss in expressivity to a first degree by looking at the rank of $\mathbf{W}$. Next, we briefly introduce the more general case of low-rank parametrisations to make the idea of a bottlenecked parametrisation sharper.

Low-Rank Matrices For our work, when we say that a matrix with n rows and d columns is **low-rank**, we mean that it has rank r, $r \leq n - 1$. [3] We use this definition such that bottlenecked parametrisations are considered low-rank even when they have rank d. The motivation for this is that the rank of the output layer matrix may have been larger if the classifier was fully-parametrised: it is a design choice to use a bottlenecked parametrisation in neural network architectures. Rectangular matrices, e.g. $\mathbf{W} \in \mathbb{R}^{n \times d}$, $d < n$, must have rank at most d by construction.

Low-Rank Parametrisation

We say that a parametrisation is **low-rank** if its parametrisation is a low-rank matrix. As such, all bottlenecked parametrisations are low-rank parametrisations, but not vice-versa: a fully-parametrised classifier may be low-rank due to initialisation or as a result of training. We frame the thesis in terms of bottlenecked parametrisations and not low-rank parametrisations specifically because we will not look into such initialisation or

[3] This is in contrast to linear algebra, where a rectangular matrix is considered low-rank if its rank is strictly smaller than $\min(n, d)$.

training details. We focus on limitations that arise due to the architectural decision of introducing a bottleneck, i.e. we will not explicitly look at the rank of $\mathbf{W}$. Nevertheless, everything we will discuss in terms of reachable outputs and (un)argmaxable outputs for bottlenecked parametrisations, also apply for low-rank parametrisations. We call a classifier that has a low-rank parametrisation a low-rank classifier.

3.3.2 Pros and Cons of Low-Rank Parametrisations

Linear classifiers with Low-Rank Parametrisations (LRPs) are extremely common in DNNs. Their ubiquity is motivated by conventional wisdom which suggests that LRPs provide a desirable trade-off between expressivity and computational efficiency. In fact, LRPs are sometimes deemed unavoidable, due to computational limitations. Moreover, LRPs match our assumptions and empirical observations about real-world data: while data is high dimensional, it is often highly compressible to low-dimensional representations, e.g. word embeddings (Mikolov et al., 2013). Before diving into details, we survey the perspectives on LRPs and argue for their pros and cons.

Arguments for Low-Rank Parametrisations (LRPs)

LRPs are Computationally Appealing This is because low-rank layers are *computationally appealing*, both because they require less memory CPU/GPU cycles as well as because they reduce the number of trainable parameters in models. For this reason, low-rank parametrisations have been used to approximate full-rank log-linear layers in neural networks (Savostianova et al., 2023) and are ubiquitous for model compression/distillation (Kim et al., 2016a).

Low-rank constraints commonly exist as bottlenecks in neural network hidden layers, e.g. autoencoders (Hinton et al., 1994) and projection heads in multi-head transformers (Bhojanapalli et al., 2020) among others. While bottlenecks make a model less expressive by restricting the functions it can represent, they are desirable both computationally (Papadimitriou et al., 2021), since they require less memory and computation than full-rank layers, and as a form of inductive bias, since data is assumed to approximately lie in a low dimensional manifold (McInnes et al., 2018).

For example, Sainath et al. (2013) used a low-rank factorisation of the softmax layer to reduce the number of parameters in their speech recognition system by $30-50\%$ with no increase in word-error-rate, evidencing that the loss in expressivity does not always impact metrics that are not particularly sensitive to long tail effects.

LRPs match our Assumptions For many applications the data is assumed to be on or close to a low-dimensional manifold (McInnes et al., 2018) and thus low-rank representations suffice to approximate the data well. We highlight a caveat here, that a low-rank manifold need not be linear, so this assumption does not motivate LRPs of output layers we use in practice, but it does motivate LRPs deeper in the network.

LRPs Work Excellently in Practice Perhaps the most important argument, is that low-rank parametrisations have served us well and work well in practice. In Table 1.1 we saw how all the latest LLMs that have enabled applications thought impossible until now are all constrained by LRPs.

While it is easy to show that LRPs are problematic in theory, it is much harder to delineate the consequences of this in practice. LRPs have many side-effects which are not yet well understood. With this caveat in mind, we discuss some limitations of LRPs and some potential consequences.

Arguments Against LRPs

LRPs are Less Expressive LRPs are not as expressive as full-rank models (Ganea et al., 2019; Yang et al., 2018). This means that the approximations can sometimes be far off. For example, DNNs struggle at approximating examples in the long tail (Horn et al., 2017; Kang et al., 2020; Menon et al., 2021b). More importantly, *we cannot easily quantify the loss in expressivity.* For example, for the softmax bottleneck case, we can set Singular Value Decomposition (SVD)-style bounds on Mean Squared Error (Ganea et al., 2019) but we are not aware of similar bounds in terms of KL-divergence.

LRPs Produce Badly Calibrated Classifiers Neural network models that output probabilities are generally badly calibrated (Guo et al., 2017; Jiang et al., 2021). A model is calibrated when it is not over or under confident, i.e. the probability it assigns to a given output is consistent with the empirical counts of that output. In Section 3.3.3 we will see how LRPs constrain probability distributions: apart from prediction it is reasonable to assume that such constraints hinder calibration.

LRPs have Side-Effects They have other obscure side-effects: in LLMs, low-rank output layers oversmooth the probabilities of tokens (Demeter et al., 2020). Finlayson et al. (2023) argue that this is the reason truncation sampling (Fan et al., 2018; Hewitt et al., 2022; Holtzman et al., 2020) is effective for generating tokens from LLMs:

truncating the distribution drops artefacts introduced by the BSL. Kuian et al. (2019) has previously argued that this is due to issues with calibration, which can be partly attributed to the output layer being low-rank.

LRPs can create Unargmaxable Outputs This is our core focus and contribution.

3.3.3 Expressivity of BSLs

In Section 3.2.1, we discussed the expressivity of a parametrisation. We now make this concept sharper for softmax and sigmoid classifiers; we ask: *what does it mean for the corresponding classifier to be fully expressive*?

Intuitively, a fully expressive classifier should be able to produce any target probability distribution over the outputs. This requirement leads to a slightly different result for softmax and sigmoid classifiers, because softmax classifiers have one fewer free variable due to Property 3.2. As we will see, softmax layers with n categories can be fully expressive with a $\mathbf{W}$ having rank $n-1$, while sigmoid layers require rank n. While this difference will not be important for our experiments, as we will consider BSLs having $d \ll n$, we clarify this difference here to avoid confusion when we discuss argmaxability in low-dimensional examples in Section 3.5.

Fully-Expressive Softmax Classifiers

Fully-Expressive Softmax Classifier

Def. 3.3.1. We say that a softmax classifier is **fully-expressive** if it can represent any categorical distribution over n outputs, i.e. it can represent any point in the $(n-1)$-simplex:

$$\forall \mathbf{z} \in \operatorname{relint}(\Delta_{n-1}) \quad \exists \mathbf{x} \in \mathbb{R}^d : \mathbf{z} = \sigma_{\Delta}(\mathbf{W}\mathbf{x}) \tag{3.38}$$

We note that due to Property 3.2, a softmax classifier having $d = (n-1)$ can still be fully expressive, as long as $\mathbf{1}_n \notin \operatorname{span}(\mathbf{W})$.

Bottlenecked Softmax Layer When a softmax classifier is not fully-expressive, we say that we have a **Bottlenecked Softmax Layer** (BSL_{Δ}) (Yang et al., 2018). We note that Yang et al. (2018) defined the softmax bottleneck in terms of the rank of the matrix of target log-probabilities. In contrast, our definition assumes that the target

log-probabilities can be any point in the simplex and our softmax layer is bottlenecked if there are any categorical distributions that cannot be represented.

Bottlenecked Softmax Layer $\mathrm{BSL}_{\triangle}$

Def. 3.3.2. Consider a softmax classifier parametrised by $\mathbf{W} \in \mathbb{R}^{n \times d}$. We say that the softmax classifier is a **softmax bottleneck** if $\operatorname{rank}\left(\mathbf{W} - \overline{\mathbf{W}}_{\text{col}}\right) < n-1$, i.e. if the rank of $\mathbf{W}$ after column mean centering is less than $n-1$.

A softmax classifier with $d < n-1$ must be a softmax bottleneck.

Example 12: Why Column Mean Centering Matters

In Definition 3.3.2, we relied on the rank of $\mathbf{W}$ after column mean centering. To justify why we apply column mean centering, consider the softmax layer:

$$\sigma_{\triangle}(\mathbf{W}\mathbf{x}), \quad \mathbf{W} = \left[\frac{1}{\sqrt{n}}\mathbf{1}_n \; \mathbf{W}_{:,-1}\right] \in \mathbb{R}^{n \times d}, d = n-1 \tag{3.39}$$

Given the above specification for $\mathbf{W}$, it is possible that $\operatorname{rank}(\mathbf{W}) = n-1$, so if we build our criterion based on the rank of $\mathbf{W}$, this softmax layer would be fully expressive. However, we know that any constant offset component of $\mathbf{W}$ is cancelled by softmax (Property 3.1). For this reason, we apply column mean centering to remove the constant component that gets cancelled by softmax. By doing so, in our case, we have that:

$$\operatorname{rank}\left(\overline{\mathbf{W}}_{\text{col}}\right) = n-2 \tag{3.40}$$

since after column mean centering, the constant column becomes $\mathbf{0}_n$. Therefore, for this choice of $\mathbf{W}$, $\sigma_{\triangle}(\mathbf{W}\mathbf{x})$ is a softmax bottleneck.

As we saw earlier, BSLs constrain the logits to be in a d-dimensional subspace, this means that the categorical distributions produced by BSLs are also constrained. In Fig. 3.7, we illustrate the categorical distributions that can be produced for a particular $\mathrm{BSL}(\mathbf{x}; \mathbf{W})$ having $\mathbf{W} \in \mathbb{R}^{4 \times 2}$. As can be seen, we can only represent a 2D surface within $\triangle_3$. In Section 3.5, we highlight the discrete version of this constraint: due to the BSL some rankings of the categories must be unargmaxable and some categories may be unargmaxable. We now discuss the sigmoid case.

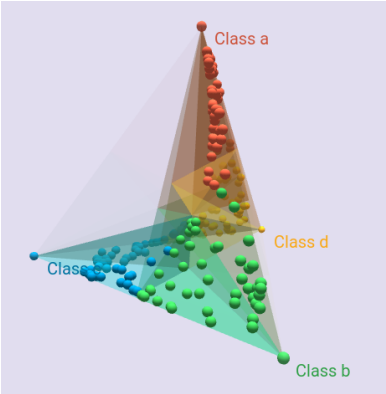


Figure 3.7: $\mathrm{BSL}_{\triangle}$ has categorical distributions that cannot be represented. We plot some of the softmax activations of reachable logits for a $\mathrm{BSL}_{\triangle}$ with $n = 4$ and $d = 2$. As can be seen, we can only represent a 2D surface within the probability simplex: most parameters of the categorical distribution cannot be represented. For a full interactive demo see https://viz.unargmaxable.ai/softmax/.

Fully-Expressive Sigmoid Classifiers

Fully-Expressive Sigmoid Classifier

Def. 3.3.3. We say that a sigmoid classifier is **fully-expressive** if it can represent the parameters of any joint distribution of n *independent* Bernoulli random variables. As we saw in Section 3.2.3, geometrically this means we can represent any point in $\square_n$.

$$\forall \mathbf{z} \in \operatorname{relint}(\square_n) \quad \exists \mathbf{x} \in \mathbb{R}^d : \mathbf{z} = \sigma_{\square}(\mathbf{W}\mathbf{x}) \tag{3.41}$$

Bottlenecked Sigmoid Layer When a sigmoid classifier is not fully-expressive, we say we have a $\mathrm{BSL}_{\square}$.

Bottlenecked Sigmoid Layer ($\mathrm{BSL}_{\square}$)

Def. 3.3.4. Consider a sigmoid classifier parametrised by $\mathbf{W} \in \mathbb{R}^{n \times d}$. We say that the sigmoid classifier is a **sigmoid bottleneck** if $\operatorname{rank}(\mathbf{W}) < n$.

A sigmoid classifier with $d < n$ must be a sigmoid bottleneck.

As a consequence of using a $\mathrm{BSL}_{\square}$, as we will see in the next section, some label assignments must be unargmaxable.

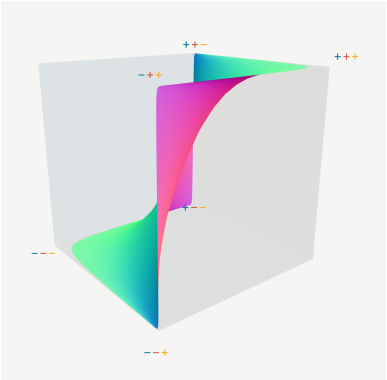


Figure 3.8: $\mathrm{BSL}_{\square}$ has independent Bernoulli distributions that cannot be represented. We plot the sigmoid activations of the reachable logits for a $\mathrm{BSL}_{\square}$ with $n = 3$ and $d = 2$. For the vertices of $\square_3$, we represent 0 with $-$ and 1 with $+$. As can be seen, we can only represent a 2D surface within the cube: most parameters for a joint distribution over 3 independent Bernoulli variables cannot be represented.

3.4 Argmaxability

> *Perhaps, instead of the most efficient paths, we should be looking for the most pleasant, or the greenest, or the most surprising, or the most beautiful.*
>
> —Federico Ardila, "CAT(0) Geometry, Robots, and Society".

We now precisely define argmaxability and elaborate on why bottlenecked classifiers have outputs that cannot be predicted for any input despite having non-linear activation functions. We begin by introducing the general case of unargmaxable outputs for a probabilistic classifier and consider more specifically the case of log-linear probabilistic classifiers. We introduce MCC and MLC as more specific cases of such log-linear classifiers and define what outputs are (un)argmaxable for them.

Notation for Argmax Prediction Before we dive deeper, we introduce some notation. We will need to define two components for a classifier: 1. a scoring function to score outputs of interest; 2. a definition of the outputs of interest.

Score Functions We think of classifiers as functions f that score a set of outputs $y \in \mathcal{Y}$. In general, the family of score functions can be as general as we like. However, in this work we will focus on functions that assign probabilities to outputs, and in particular log-linear classifiers.

Outputs The score function is applied to a set of outputs of interest, $\mathcal{Y}$. We focus here on categories, rankings and subsets (Section 2.1). We assume the outputs are countable; we can therefore assign each output a unique integer index. In what follows, we use the index to succinctly define the argmax using the output's index i without necessarily referring to the complete output y_i. Later on we will also refer to an output using its sign vector, as we introduced in Section 2.5.1.

Argmax Prediction Given a score function f and a set of outputs of interest, how do we make decisions? A common decision rule is to choose the highest scoring output, i.e. the **argmax**, which we define next.

Argmax

Def. 3.4.1. Consider a score function f that scores a set of candidate outputs $\mathcal{Y} = \{y_1, y_2, \ldots, y_n\}$. The **argmax** of f is:

$$\underset{y \in \mathcal{Y}}{\operatorname{argmax}} f(y) = \{i \in [n] : \quad f(y_i) \geq f(y_j), \quad \forall j \in [n], \quad i \neq j\}. \tag{3.42}$$

Example 13: Argmax

$f(y)$	1	**5**	3	0
y	0	**1**	2	3

The maximum is 5, the argmax is {1}.

Example 14: Argmax with a Tie

$f(y)$	1	**5**	3	**5**
y	0	**1**	2	**3**

The maximum is 5, the argmax is {1, 3}.

Argmaxability Informally, an output is **argmaxable** for a score function if we can find an input for which the output is assigned the largest score. We will also talk about sets of outputs being argmaxable, in which case we mean that each output of the set is argmaxable.

3.4.1 Argmaxability for a Probabilistic Model

We have so far introduced argmaxability for a general family of score functions. We now focus on probabilistic models.

For probabilistic models, the scores are probabilities, i.e. we have $f(y) = P(y \mid \mathbf{x})$. In this thesis we ask: which outputs are argmaxable for a given probabilistic model? We will answer this question specifically for log-linear models, but we first need to define what it means for an output to be argmaxable.

Since we have already defined the argmax function, you may think we have already defined (un)argmaxable outputs, i.e. they are simply the outputs that can be produced as the result of the argmax operation. However, things are slightly trickier. We need to give special consideration to ties, i.e. to the case where multiple outputs are assigned the maximum score and are tied as the argmax.

For a probabilistic classifier which conditions on an input, $\mathbf{x}$, and outputs a probability for each target output, y, we define **(un)argmaxable outputs** as follows:

Argmaxability

Def. 3.4.2. An output y^* is **(un)argmaxable** if:

$$(\nexists)\, \exists \mathbf{x} : \operatorname*{argmax}_{y \in \mathcal{Y}} P(y \mid \mathbf{x}) = \{y^*\} \tag{3.43}$$

We note that in the above definition we do not just check whether there exists an $\mathbf{x}$ such that y^* is a member of the argmaxable set. Instead, we take the target set to be the single element set, $\{y^*\}$, in order to explicitly avoid the case of ties.

To see why we need to avoid ties, consider the case where the output is a uniform distribution, i.e. $P(y \mid \mathbf{x}) = \frac{1}{|\mathcal{Y}|}$. Note that according to Definition 3.4.1, all outputs are the argmax for the uniform distribution. Therefore, if we allowed ties in the definition of argmaxable outputs, we would end up with the problem that all outputs are argmaxable if our model can produce the uniform distribution. This would be a problem for two reasons:

1. our definition of argmaxable outputs would become vacuous for many models of interest, because most models we consider can represent the uniform distribution over all outputs.

2. our definition would not be able to distinguish between outputs that are always tied and outputs that may be tied for some inputs but can also be assigned the largest score independently.

We will elaborate further on the implications of our definition with concrete examples in Section 3.4.2.

We now have a definition of argmaxability that is precise. However, we cannot say much about which outputs are argmaxable unless we elaborate on our model's parametrisation. We therefore follow our feature encoder / classifier decomposition from Section 3.1.1. We now see that outputs can be unargmaxable for two reasons:

1. The classifier $\mathbf{W}$ is such that there is no $\mathbf{x}$ such that we can predict y^*.
2. The feature encoder imposes restrictions on $\mathbf{x}$ making it impossible to produce the $\mathbf{x}$ needed such that y^* can be predicted.

In Section 3.1.1 we explained that we are making the fully-expressive encoder assumption. As such, we drop the second case and focus on constraints imposed by the classifier alone.

3.4.2 Argmaxability for a Linear Classifier

In order to understand argmaxability for a log-linear classifier, we need to understand argmaxability for a linear classifier, which we now introduce. Consider a linear function $f(\mathbf{y}) = \mathbf{W}\mathbf{x}$ which scores each output y as $f(y) = \mathbf{w}_y^\top \mathbf{x}$, where $\mathbf{w}_y$ is the classification vector for output $\mathbf{y}$.

When considering argmaxability, it is helpful to think of the above as a function of $\mathbf{x}$. From this perspective, we are asking if there exists a linear functional, $\mathbf{x}$, such that $\mathbf{w}_y$ is the argmax. We plot this interpretation in Fig. 3.9.

Connection to Linear Programming We note that in its new role, $\mathbf{x}$ is reminiscent of the cost vector of a LP:

$$\text{maximise} \quad \mathbf{x}^\top \mathbf{w} \tag{3.44}$$

$$\text{subject to} \quad \mathbf{A}\mathbf{w} \leq \mathbf{b} \tag{3.45}$$

where $\mathbf{A}$ and $\mathbf{b}$ are the halfspace constraints defining the valid $\mathbf{w}$. The difference is that our matrix $\mathbf{W}$ is a vertex representation of a polytope of valid $\mathbf{w}$, as opposed to a halfspace representation, as is common in LPs. If the $\mathbf{w}_y$ are vertices of $P = \text{conv}(\mathbf{W})$, then we can construct $\mathbf{A}$ and $\mathbf{b}$ such that they are the halfspace description of P. But note that if some $\mathbf{w}_y$ is interior to P, it cannot be a solution of the LP. In other words, we know from Linear Programming (Dantzig et al., 1954), that if a solution to the LP exists, it will be found at a vertex/face of P; interior points are unargmaxable. In fact,

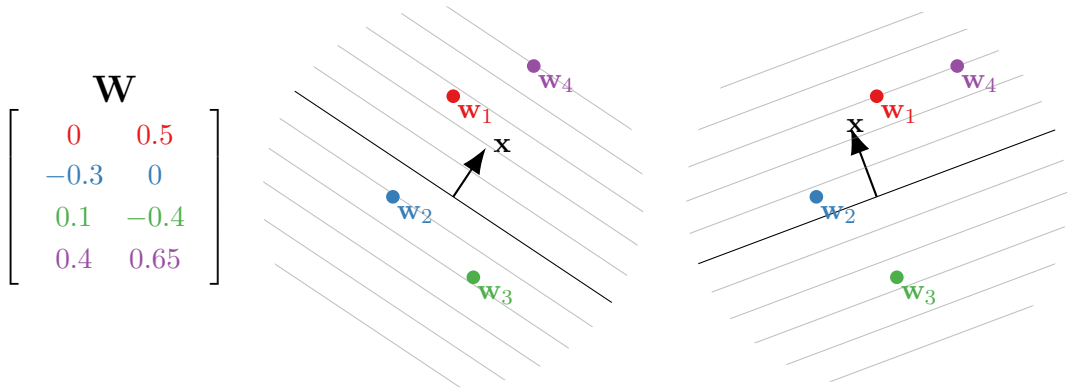


Figure 3.9: Interpretation of output activations as a function of $\mathbf{x}$. On the left we have the matrix $\mathbf{W}$ which defines our linear model $f(\mathbf{y}) = \mathbf{W}\mathbf{x}$. On the right we plot how the function $f(\mathbf{x}) = \mathbf{w}_y^\top \mathbf{x}$ changes for a given $\mathbf{x}$. Each gray line perpendicular to $\mathbf{x}$ is a level set of the linear function, i.e. any $\mathbf{w}$ on the same line produces the same output, with lines further in the direction of $\mathbf{x}$ producing a larger output. As can be seen, for $\mathbf{x}$ in the middle figure, $\mathbf{w}_4$ produces the largest activation, so $\operatorname{argmax}_{y\in\mathcal{Y}} \mathbf{w}_y^\top \mathbf{x} = \{4\}$. On the other hand, for the $\mathbf{x}$ on the right, we have a tie, i.e. $\operatorname{argmax}_{y\in\mathcal{Y}} \mathbf{w}_y^\top \mathbf{x} = \{1, 4\}$.

if we group together the $\mathbf{x} \in \mathbb{R}^d$ that have the same argmax, we partition $\mathbb{R}^d$ into regions depending on which vertex/face of P is the argmax.[4]

The regions where a vertex is the argmax is an intersection of halfspaces (see also Section 3.4.3). Therefore, our definition of (un)argmaxable outputs directly corresponds to detecting non-empty intersections of halfspaces. We will show how to detect such non-empty intersections of halfspaces using LPs, e.g. see Eqs. (4.2) and (4.9).

Examples for Linear Classifiers

In this section we provide a few examples of (un)argmaxable outputs. We use them to build intuition and justify why we require the argmax to be a single element set in our definition of (un)argmaxable outputs.

Example 15: Setting $\mathbf{W} = \mathbf{0}$

If we set $\mathbf{W} = \mathbf{0}$, all outputs are unargmaxable. This is because $\operatorname{argmax}_{y\in\mathcal{Y}} \mathbf{W}\mathbf{x} = \operatorname{argmax}_{y\in\mathcal{Y}} \mathbf{0} = \{y : y \in \mathcal{Y}\}$ for all $\mathbf{x}$. As such, there is no $\mathbf{x}$ such that the argmax is the set containing an individual output.

[4] This is known as the normal fan of a polytope (see Ziegler (1995, Example 7.3)).

Example 16: The Problem with Tied Outputs

Consider the cases where there exists an $\mathbf{x}$ for which all scores of the linear function are tied. In equations, $\exists \mathbf{x} : f(\mathbf{y}) = \mathbf{W}\mathbf{x} = \mathbf{c}$. For such an $\mathbf{x}$, we have that $\operatorname{argmax}_{y \in \mathcal{Y}} \mathbf{W}\mathbf{x} = \operatorname{argmax}_{y \in \mathcal{Y}} \mathbf{c} = \{y : y \in \mathcal{Y}\}$. Examples where tied outputs can be produced include:

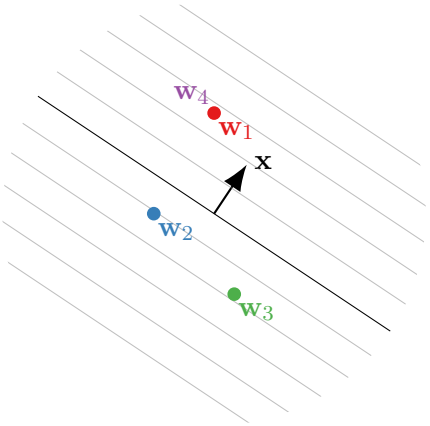
- Choosing $\mathbf{x} = \mathbf{0}$.
- Setting $\mathbf{W} : W_{i,j} = c \quad \forall i, j, \quad c \in \mathbb{R}$.
- Setting $\mathbf{W}$ such that the columns of $\mathbf{W}$ are linearly dependent.

If we allowed for ties in the definition of (un)argmaxable outputs, all outputs would be argmaxable for the above situations. Moreover, allowing the case $\mathbf{x} = \mathbf{0}$ would imply that all outputs are argmaxable irrespective of $\mathbf{W}$.

Example 17: Example with Subset of Outputs Tied

Another way of getting tied outputs is when we have the same representation for different outputs. Consider $\mathbf{W}$ with a subset of the rows repeated. For example, consider the case where rows 1 and 4 are identical, like in the figure below. The outputs that have the same parametrisation are unargmaxable.

$$\mathbf{W} = \begin{bmatrix} 0 & 0.5 \\ -0.3 & 0 \\ 0.1 & -0.4 \\ 0 & 0.5 \end{bmatrix}$$



Note that because $\mathbf{w}_1 = \mathbf{w}_4$, the scores for outputs $y = 1$ and $y = 4$ are tied for all $\mathbf{x}$. Therefore, outputs $y = 1$ and $y = 4$ are unargmaxable, because the resulting argmax can never be the single element set $\{1\}$ or $\{4\}$.

3.4.3 Argmaxability for a Log-Linear Classifier

Although log-linear classifiers are not linear, their argmaxable outputs depend only on $\mathbf{W}$. [5] This is because, as we show next, we can ignore the non-linear activation functions because of their strictly increasing property. This means that we can analyse log-linear classifiers using our understanding from the previous section.

Passing Argmax Through Non-Linearity Recall that for log-linear classifiers we have $f(y) = \sigma(g(y))$, where σ is a logistic type function like softmax ($\sigma_{\triangle}()$, Definition 3.2.1) or sigmoid ($\sigma_{\square}()$, Definition 3.2.4), and $g(y) = \mathbf{w}_y^\top \mathbf{x}$ is the linear function that scores output y given input features $\mathbf{x}$. As we saw in Eqs. (3.28) and (3.36), both softmax and element-wise sigmoid are order preserving. Since order preserving maps do not change the locations of the maxima of their arguments, they propagate argmax to their arguments. That is, since f is order preserving, we have:

$$\operatorname*{argmax}_{y \in \mathcal{Y}} f\left(g(y)\right) = \operatorname*{argmax}_{y \in \mathcal{Y}} g(y) \tag{3.46}$$

For example, for softmax, using Eq. (3.32), we have:

$$\begin{aligned}
\{y^*\} &= \operatorname*{argmax}_{y \in \mathcal{Y}} P(y \mid \mathbf{x}) && (3.47)\\
&= \operatorname*{argmax}_{y \in \mathcal{Y}} \sigma_{\triangle}(\mathbf{W}\mathbf{x})_y && (3.48)\\
&= \operatorname*{argmax}_{y \in \mathcal{Y}} \left(\mathbf{W}\mathbf{x}\right)_y && (3.49)\\
&= \operatorname*{argmax}_{y \in \mathcal{Y}} \mathbf{w}_y^\top \mathbf{x} && (3.50)
\end{aligned}$$

As such, if we want to compute the argmax of a log-linear classifier, f, for an input, $\mathbf{x}$, we are effectively taking an argmax over the linear functionals defined by $\mathbf{w}_y$.

Polytope View: Interior Outputs are Unargmaxable

Argmaxability (Polytope)

Proposition 3.1. *Consider a softmax classifier $P(y \mid \mathbf{x}) = \sigma_{\triangle}(\mathbf{W}\mathbf{x})_y$. An output y^* is **(un)argmaxable** if:*

$$\mathbf{w}_{y^*}(\in) \notin \operatorname{conv}\left(\left\{\mathbf{w}_y, y \in \mathcal{Y} \setminus \{y^*\}\right\}\right) \tag{3.51}$$

[5] More precisely, as we saw in Section 3.5.1, we need not know the basis; knowing $\operatorname{span}(\mathbf{W})$ suffices.

Given the matrix $\mathbf{W}$, form a polytope $P_{\mathcal{Y}\setminus\{y^*\}}$ by computing the convex hull (Definition 2.2.1) of $\mathbf{W}_{-y^*}$, i.e. the convex hull of the rows of $\mathbf{W}$, excluding the row which corresponds to y^*. If $\mathbf{w}_{y^*}$ is in $P_{\mathcal{Y}\setminus\{y^*\}}$, y^* is unargmaxable (Dantzig et al., 1954).

Hyperplanes View: Unintersected Orthants are Unargmaxable

Argmaxability (Hyperplanes)

Proposition 3.2. *Consider a softmax classifier* $P(y \mid \mathbf{x}) = \sigma_{\triangle}(\mathbf{W}\mathbf{x})_y$. *An output represented by sign vector* $\mathbf{y}^*$ *is* ***(un)argmaxable*** *if:*

$$(\nexists)\exists \mathbf{x} : \operatorname{sign}(\mathbf{B}\mathbf{W}\mathbf{x}) \geq \mathbf{y}^* \tag{3.52}$$

As we showed in Section 2.4.3, while for MCC the region where an output is assigned the largest probability is not an orthant, we can always form the region by joining orthants. The orthants correspond to rankings of the outputs; applying matrix $\mathbf{B}$ to $\mathbf{W}$ subdivides △ into the regions of ⬡ which correspond to rankings (see Section 2.4.2). We then compare sign vectors using Definition 2.5.2 to join the needed orthants together.

Criteria From the examples above, we extrapolate to the bigger picture in Table 3.1. In the third column, we encode the fact that if the projection of a polytope by $\mathbf{W}$ kills a vertex, the corresponding output is unargmaxable. In the fourth column, we encode the fact that if the restriction of ⊞ by $\mathbf{W}$ kills all needed orthants, the corresponding output is unargmaxable. We only need one set of criteria to proceed, and we focus on the hyperplane criteria, as we explain next.

Hyperplane Criteria

In what follows, we elaborate on the hyperplane criteria in Table 3.1 . We focus on the hyperplanes perspective, because:

1. The polytope criteria, while intuitive, are not efficient to work with (other than in the case of multi-class classification), since for the rankings and subsets there are $n!$ and 2^n vertices correspondingly.

2. If we have an affine layer that has a bias vector, it is more easily interpretable

Task	Output	Unargmaxability Criterion		Exists
		Polytope	Hyperplanes	
Multi-Class	category	$\mathbf{w}_{\mathbf{y}^{\triangle}} \in \operatorname{conv}\left((\mathbf{IW})_{-\mathbf{y}^{\triangle}}\right)$	$\nexists \mathbf{x} : \operatorname{sign}(\mathbf{BWx}) \geq \mathbf{y}^{\triangle}$	*may*
	ranking	$\mathbf{w}_{\mathbf{y}^{\hexagon}} \in \operatorname{conv}\left((\mathbf{PW})_{-\mathbf{y}^{\hexagon}}\right)$	$\nexists \mathbf{x} : \operatorname{sign}(\mathbf{BWx}) = \mathbf{y}^{\hexagon}$	*must*
Multi-Label	subset	$\mathbf{w}_{\mathbf{y}^{\square}} \in \operatorname{conv}\left((\mathbf{CW})_{-\mathbf{y}^{\square}}\right)$	$\nexists \mathbf{x} : \operatorname{sign}(\mathbf{Wx}) = \mathbf{y}^{\square}$	*must*

Table 3.1: Criteria for unargmaxable categories, $\mathbf{y}^{\triangle}$, rankings, $\mathbf{y}^{\hexagon}$, and label assignments, $\mathbf{y}^{\square}$, for a BSL, $\mathbf{W}$. $\mathbf{I} = \operatorname{verts}\left(\triangle\right)$, $\mathbf{C} = \operatorname{verts}\left(\square\right)$ and $\mathbf{P} = \operatorname{verts}\left(\hexagon\right)$ (Section 2.3) and $\mathbf{B}$ is the braid matrix (Section 2.4.2) which defines $\circledast$ (Definition 2.4.8). We highlight that the $\geq$ comparison is the partial order between sign vectors (Definition 2.5.2). We use equality for $\mathbf{y}^{\square}$ and $\mathbf{y}^{\hexagon}$, because these are maximal elements.

in the hyperplanes perspective, since the bias term acts like an offset to the hyperplanes, as we discuss in Section 3.5.3.

3. Moreover, with the hyperplane criteria we get insights into the feature space. For example, we can get an estimate of how robust to noise a specific class is by detecting how large the argmaxable region is.

We now use the hyperplane criteria and define (un)argmaxability for MCC and MLC.

3.4.4 Argmaxability for Multi-class Classification

For MCC with n categories, our outputs are $\mathcal{Y} = [n]$. We compute $P(\mathbf{y} \mid \mathbf{x}) = \sigma_{\triangle}(\mathbf{Wx})$ (Definition 3.2.2). By making Definition 3.4.2 specific to MCC we get:

Argmaxability (Category)

Def. 3.4.3. A **category** y^* is **(un)argmaxable** if:

$$(\nexists)\, \exists \mathbf{x} : \operatorname*{argmax}_{y \in \mathcal{Y}} \sigma_{\triangle}(\mathbf{Wx})_y = \{y^*\} \tag{3.53}$$

As we showed in Section 2.4.3, the region of $\triangle$ where the target category is argmaxable is the union of all regions of $\circledast$ that rank the target category above all others. The region exists if:

$$\exists \mathbf{x} : \operatorname{sign}(\mathbf{BWx}) \geq \mathbf{y}^{\triangle} \tag{3.54}$$

where $\mathbf{y}^{\triangle}$ is the sign vector given in Eq. (2.67). [6] Next we define when a ranking of the categories is argmaxable.

Argmaxable Rankings

In this case, our outputs are all rankings on n elements, $\mathcal{Y} = S_n$ (see Section 2.1.2). In this case, the sign vector $\mathbf{y}^{\hexagon}$ (Eq. (2.66)) defines the region that $\mathbf{W}$ needs to intersect. The region exists when:

$$\exists \mathbf{x} : \operatorname{sign}(\mathbf{B}\mathbf{W}\mathbf{x}) = \mathbf{y}^{\hexagon} \tag{3.55}$$

where $\mathbf{y}^{\hexagon}$ is given in Eq. (2.66).

3.4.5 Argmaxability for Multi-label Classification

For MLC with n labels, our outputs are label assignments, $\mathcal{Y} = \{-, +\}^n$. We assume conditional independence for each binary classifier given the input; i.e. this is a $\text{BSL}_{\square}$ parametrised by $\mathbf{W}$, as we described in Definition 3.2.5. By making Definition 3.4.2 specific to MLC, we obtain:

Argmaxability (Multi-Label)

Def. 3.4.4. A **label assignment** y^* is **(un)argmaxable** if:

$$(\nexists)\, \exists \mathbf{x} : \operatorname*{argmax}_{y \in \mathcal{Y}} \prod_{i=1}^{n} \sigma_{\square}\left(y_i \mathbf{w}_i^{\top} \mathbf{x}\right) = \{y^*\} \tag{3.56}$$

For MLC with independence assumptions, argmax is equivalent to taking the sign of the activation.

Argmaxability (Multi-Label, Independence Assumptions)

Def. 3.4.5. A **label assignment** $\mathbf{y}^*$ is **argmaxable** for a classifier $\mathbf{W}$ if there exists an input $\mathbf{x}$ for which thresholding the output probabilities using the decision rule $\operatorname{sign}\left(P(y_i = + \mid \mathbf{x}) - \frac{1}{2}\right)$ produces y_i.[a].

$$\mathbf{y}^* \text{ is argmaxable} \iff \exists \mathbf{x} : \operatorname{sign}(\mathbf{W}\mathbf{x}) = \mathbf{y}^* \tag{3.57}$$

[a] Due to monotonicity of sigmoid $\sigma_{\square}(a) > \frac{1}{2} \iff a > 0$.

In terms of sign vectors, $\mathbf{y}^{\square}$ (Eq. (2.65)) defines the region that $\mathbf{W}$ needs to intersect. The region exists when:

$$\exists \mathbf{x} : \operatorname{sign}(\mathbf{W}\mathbf{x}) = \mathbf{y}^{\square} \tag{3.58}$$

[6] In the LP (Eq. (4.9)) we drop the constraints that have a $\mathbf{0}$ sign vector coefficient.

Summary

- We defined (un)argmaxability.
- We introduced the (un)argmaxability criteria for unargmaxable categories, rankings and subsets.

Halfspace intersection problems can be solved via LPs. We will revisit the above in Chapter 4, where we will implement the LPs and show that BSLs cause DNNs used in practice to have unargmaxable outputs. In the next section, we revisit when there may and when there must be unargmaxable outputs due to BSLs in MCC and MLC.

3.5 Low-Rank Constraints

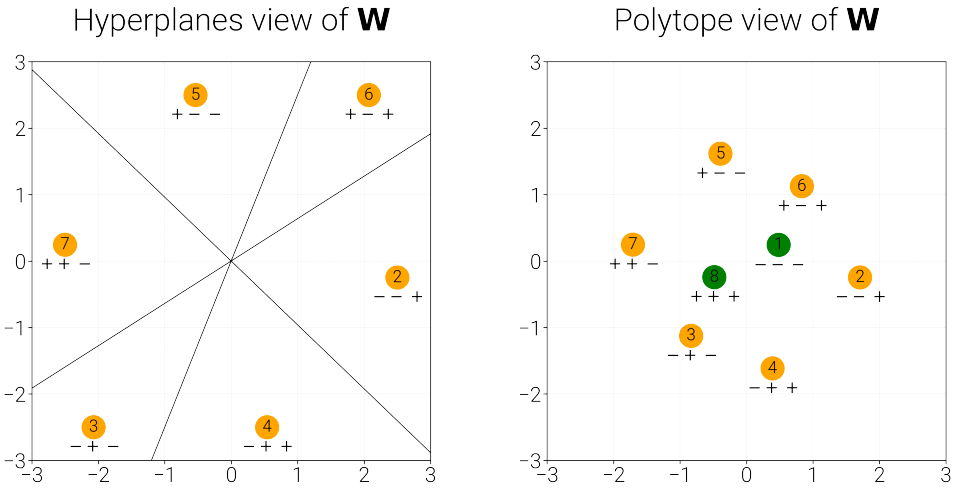


Figure 3.10: The Hyperplane Arrangement view and Polytope view for a $\mathrm{BSL}_{\square}(\mathbf{x};\mathbf{W})$. We look at $\mathbf{W} \in \mathbb{R}^{3\times 2}$ from two perspectives. **On the left**, we illustrate $\mathcal{P}\left(\boxplus_3, \mathbf{W}\right)$: $\mathbf{W}$ restricts the outputs to a 2D subspace; this subspace intersects 6/8 orthants, the regions of $\boxplus_3$ in $\mathbb{R}^3$. **On the right**, we illustrate $\mathcal{P}\left(\square_3, \mathbf{W}\right)$: $\mathbf{W}$ projects $\square_3$ to $\mathbb{R}^2$. We number the vertices $\square_3$ to identify them in the projection and colour the ones that survive on the boundary of the projection yellow. As we saw in Section 2.5, the vertices of the cube that survive on the boundary of the projection are the same as the intersected orthants, and correspond to the argmaxable label assignments.

In this section, we consolidate our findings from Chapter 2 and Section 3.4 and apply them to BSLs which we will use for MLC and MCC. We revisit Figs. 1.2 and 1.3 to visualise unargmaxability from both views:

1. the **hyperplane arrangement view** $\mathcal{H}(\mathbf{W})$

2. the **polytope view** $\mathcal{Z}(\mathbf{W})$

Importantly, we further highlight the distinction between whether the classifier *may* have unargmaxable outputs and whether the classifier *must* have unargmaxable outputs.

	View	
	Hyperplanes $\mathcal{H}(\mathbf{W})$ $\mathbb{R}^d \to \mathbb{R}^n$	Polytope $\mathcal{Z}(\mathbf{W})$ $\mathbb{R}^n \to \mathbb{R}^d$
$\mathbf{w}_i$	normal vector of hyperplane	generator of zonotope
Operation	restriction	projection
Constraint	unreachable outputs	linear dependencies
Sign vectors	orthants $\boxplus_n$	vertices of $\square_n$
Argmaxable	orthant intersected	vertex on convex hull
Unargmaxable	orthant not intersected	vertex interior to convex hull

Table 3.2: Connections between the two views for a BSL, $\mathbf{W} \in \mathbb{R}^{n \times d}$, $d < n$.

3.5.1 Hyperplane Arrangement View $\mathcal{H}(\mathbf{W})$

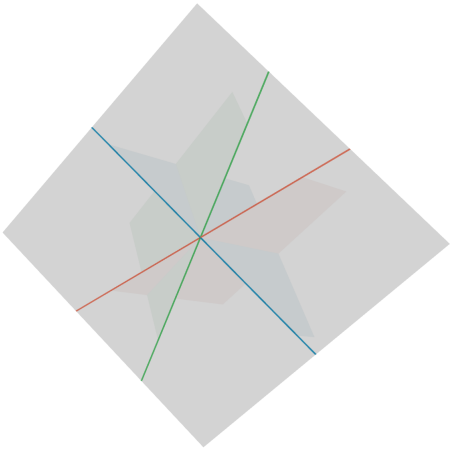
A BSL imposes constraints because it can only represent a d-dimensional subspace of $\mathbb{R}^n$, making some outputs **unreachable**. More concretely, we think of the mapping as $\mathbb{R}^d \to \mathbb{R}^n$. The layer computes:

$$\mathbf{z} = \mathbf{W}\mathbf{x}, \quad \mathbf{W} \in \mathbb{R}^{n \times d} \tag{3.59}$$

where $\mathbf{x} \in \mathbb{R}^d$ is the input, i.e. the feature vector, and $\mathbf{z} \in \mathbb{R}^n$ is the output, i.e. the logits. As we saw in Fig. 3.6, since $\mathbf{x} \in \mathbb{R}^d$, $\mathbf{z}$ is constrained to be in $\operatorname{span}(\mathbf{W})$.

Intersected Orthants As we saw in Section 2.4.4, $\mathbf{W}$ restricts $\boxplus$ to a subspace, and therefore only a small number of the orthants $\boxplus$ in $\mathbb{R}^n$ can be intersected. We illustrate this in Fig. 3.11: each intersected orthant in $\mathbb{R}^n$ appears as a region in $\mathbb{R}^d$. We identify the intersected orthants by their corresponding sign vectors.

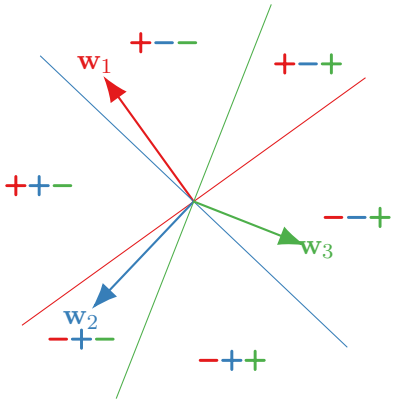
a) Orthants of $\mathbb{R}^3$



b) $\mathrm{span}\,(\mathbf{W})$ Intersects Orthants

$$\mathbf{W} = \begin{bmatrix} -0.49 & 0.68 \\ -0.66 & -0.69 \\ 0.61 & -0.24 \end{bmatrix}$$

c) Parametrisation



d) Sign Vectors

Figure 3.11: a) The label assignments as defined by $\boxplus_3$. b) Only the grey 2D subspace, $\mathrm{span}\,(\mathbf{W})$, is reachable. The regions formed in the subspace correspond to the orthants that are intersected; we can only intersect 6/8 orthants of $\mathbb{R}^3$. c) The matrix $\mathbf{W}$ which defines $\mathrm{span}\,(\mathbf{W})$. d) The argmaxable label assignments correspond to the intersected orthants; we represent them by sign vectors. Note that $++{+}$ and $-{-}-$ are not intersected and are therefore unargmaxable, similarly to the case of ◎. See also https://viz.unargmaxable.ai/braid-slice.

Sign Vectors: Identifying Intersected Orthants We can identify the orthant's sign vector by checking which side of each hyperplane in $\mathbb{R}^d$ the region is (as we did in Fig. 2.9(a)). More concretely, in Fig. 3.11 we have colour coded the rows of $\mathbf{W}$. Each $\mathbf{w}_i$ defines a hyperplane separating $\mathbb{R}^2$ into regions, which we identify via sign vectors. An entry of the sign vector is $\mathbf{0}$ if the point is on the hyperplane, $+$ if the point is in the halfspace on the side of the normal vector and $-$ if it is in the opposite halfspace. For example, $+{-}-$ is on the side of $\mathbf{w}_1$ and opposite sides of $\mathbf{w}_2$ and $\mathbf{w}_3$,

while $+0-$ is the section of the blue hyperplane separating $++-$ from $+--$. More specifically, it is the section of the blue hyperplane in the positive halfspace of the red and the negative halfspace of the green hyperplane. In general, for any point $\mathbf{x}$ we can obtain the orthant by computing $\mathbf{y} = \text{sign}(\mathbf{W}\mathbf{x})$.

Multi-Label Classification

$\text{BSLs}_{\square}$ *must* have Unargmaxable Label Assignments As we saw in Section 2.5.7, label assignments are atomic outputs. Therefore, since there are always orthants that cannot be intersected, $\text{BSLs}_{\square}$ *must* have unargmaxable label assignments.

Multi-Class Classification

We now revisit the $\text{BSL}_{\triangle}$ example from Fig. 1.2. The region where each class has the largest probability is a region of $\triangle$ (Section 2.4.3). Without loss of generality, we can decompose any such region into rankings, i.e. regions of $\hexagon$, as shown in Fig. 3.12. The region where a target category is argmaxable is the union of regions that rank it above all others.

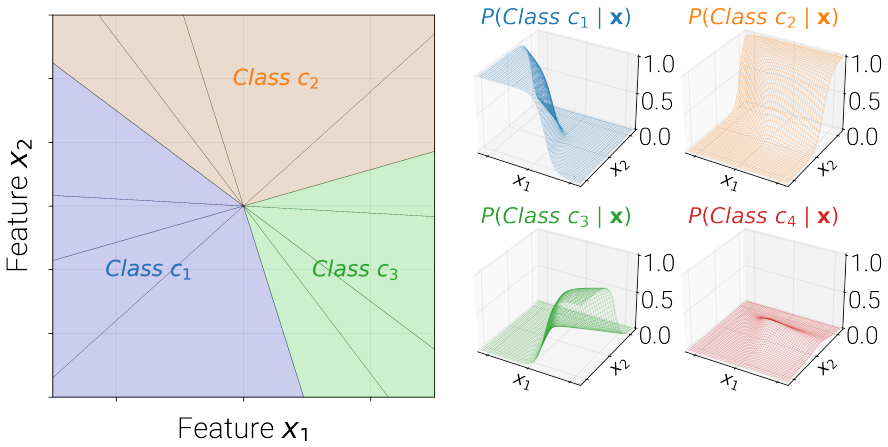


Figure 3.12: We revisit the $\text{BSL}_{\triangle}$ from Fig. 1.2. The regions of $\triangle$ are a union of the regions of $\hexagon$ which rank the target category above all others. We illustrate this more clearly in Fig. 3.13. c_4 is unargmaxable because all regions of $\hexagon$ ranking c_4 above c_1, c_2 and c_3 are unargmaxable.

$\text{BSLs}_{\triangle}$ *must* have Unargmaxable Rankings We illustrate the rankings $\hexagon$ in-Fig. 3.13. We represent each ranking by a stack of coloured disks having the largest

value at the top and the smallest at the bottom. [7] Since ⛹ is a hyperplane arrangement, restricting it to $\mathbf{W}$ must kill regions. Since rankings are atomic outputs, $\mathrm{BSLs}_{\triangle}$ must have unargmaxable rankings.[8] We show this in Fig. 3.13, where only $12/24$ rankings are argmaxable.

$\mathrm{BSLs}_{\triangle}$ *may* have Unargmaxable Categories In order for a target category to be unargmaxable, all rankings that rank that target category above all other categories must be unargmaxable. This is possible, as we illustrated in Fig. 3.13, but it is not necessary. Categories are compound outputs, so $\mathrm{BSLs}_{\triangle}$ *may* have unargmaxable categories.

3.5.2 Polytope View $\mathcal{Z}(\mathbf{W})$

We now switch to the polytope perspective: BSLs introduce linear dependencies by projecting a high dimensional space to a lower one, i.e. $\mathbb{R}^n \to \mathbb{R}^d$. We have:

$$\mathbf{x} = \mathbf{W}^\top \mathbf{z}, \quad \mathbf{W} \in \mathbb{R}^{n \times d} \tag{3.60}$$

Here $\mathbf{z} \in \mathbb{R}^n$ is the input and $\mathbf{x} \in \mathbb{R}^d$ is the output, i.e. the projection.

Sign Vectors: Identifying Vertices of the Cube We previously introduced the correspondence between the orthants of $\mathbb{R}^n$ and sign vectors. We now use sign vectors to identify vertices of $\square_n$ in $\mathbb{R}^n$. To see the correspondence, center $\square_n$ at the origin and orient it such that each vertex falls in the relative interior of an orthant. Read off the sign vector of each vertex from its corresponding orthant. [9] We now revisit our earlier claims about $\mathrm{BSLs}_{\triangle}$ and $\mathrm{BSLs}_{\square}$ with polytope visualisations of the same facts.

Multi-Label Classification

$\mathrm{BSL}_{\square}$ *must* have Unargmaxable Label Assignments In Fig. 3.14, we see the polytope version of Fig. 3.11. Note that the orthants $+++$ and $---$ which could

[7] The sign vectors $\mathbf{y}^{\hexagon}$ are the signs of $\binom{4}{2} = 6$ pairwise comparisons which define the ranking; we replace the sign vectors with the corresponding rankings for ease of exposition.

[8] We note that we are abusing the term "unargmaxable ranking" here to mean that a ranking cannot be realised (e.g. the ranking cannot be produced by argsort). We make this choice to avoid introducing more jargon, in addition to the fact that we are not considering probabilistic models over permutations, where the concept of rankings that cannot be scored above all others would be useful.

[9] This suffices for our current discussion, the more general situation arises by building a zonotope from a set of basis vectors using the Minkowski sum with positive and negative weights.

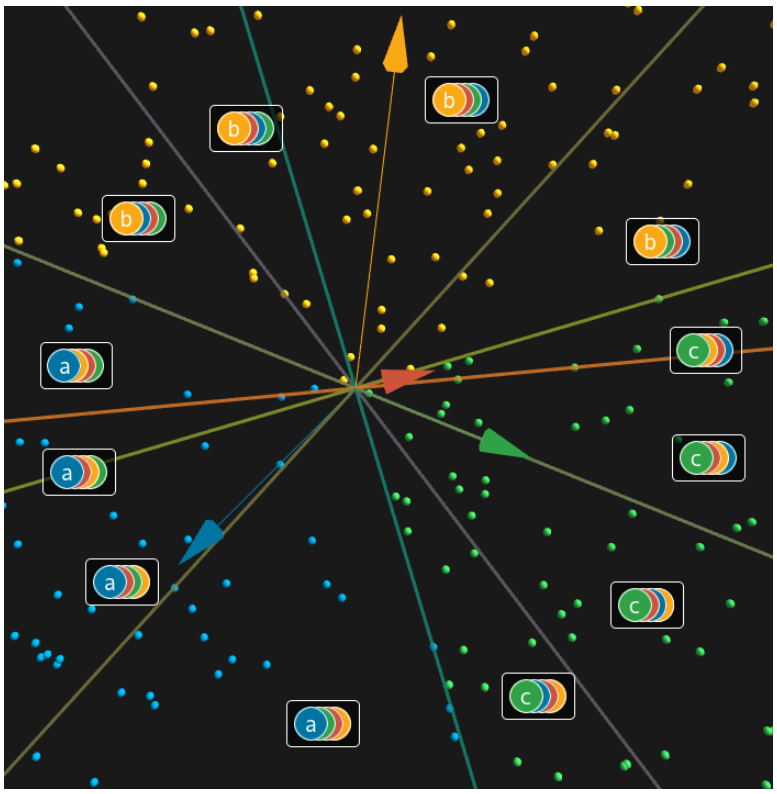
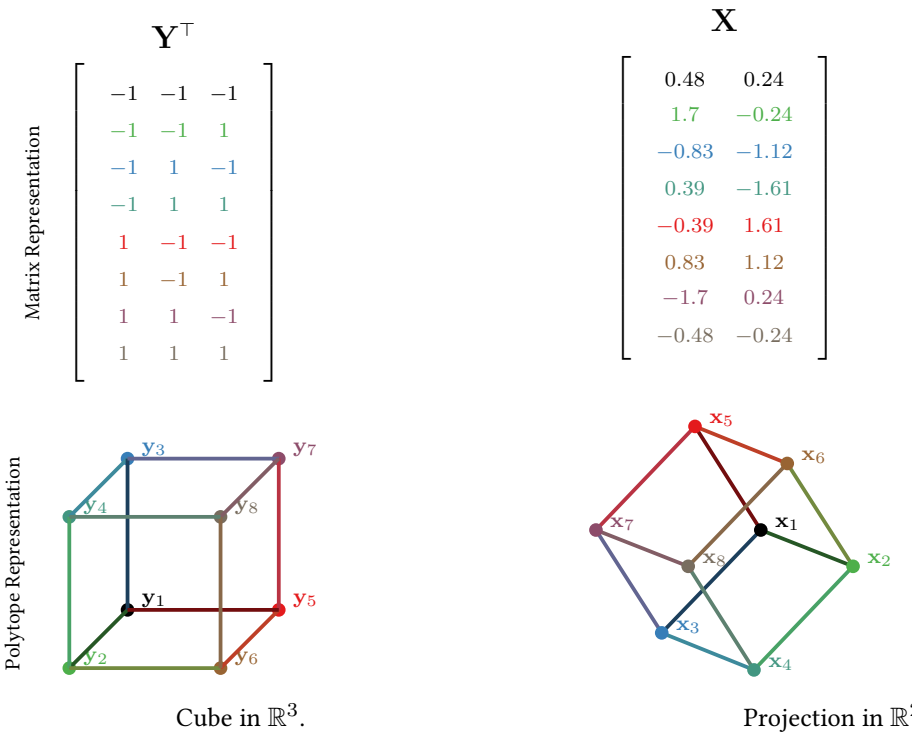


Figure 3.13: $\mathrm{BSL}_{\triangle}$ for $d = 2$ and $n = 4$ classes. In this plot, every point in 2D is an example feature vector and is coloured according to which class is assigned the largest probability. The feature vectors that fall in each region produce class probabilities that have a particular ranking. We represent each ranking by a stack of disks of colours, one disk per class, with the argmax being at the top of the stack. As can be seen, only 12/24 rankings of the 4 classes are argmaxable. In this case, all rankings that rank class d above the remaining classes are unargmaxable, so class d is unargmaxable. See https://viz.unargmaxable.ai/softmax for the interactive plot, where you can drag and drop the vectors to change the decision boundaries.

not be intersected in Fig. 3.11 are now vertices of the cube that are in the relative interior of the hexagon. Hence they are unargmaxable, as we saw in Section 3.4.

Multi-Class Classification

Without loss of generality, we can decompose the representation of each category into a more granular representation. Instead of having a vertex per category, we introduce a vertex per ranking of the categories. We form ⎔, which is a projection of □ from $\mathbb{R}^{\binom{n}{2}}$ to $\mathbb{R}^n$ (Example 8). As such, we can analyse all cases as sign vectors: we first project □ to ⎔ and then project ⎔ to $\mathbb{R}^d$ using $\mathbf{W}$, and check which rankings of the categories survive on the boundary.



Cube in $\mathbb{R}^3$. Projection in $\mathbb{R}^2$.

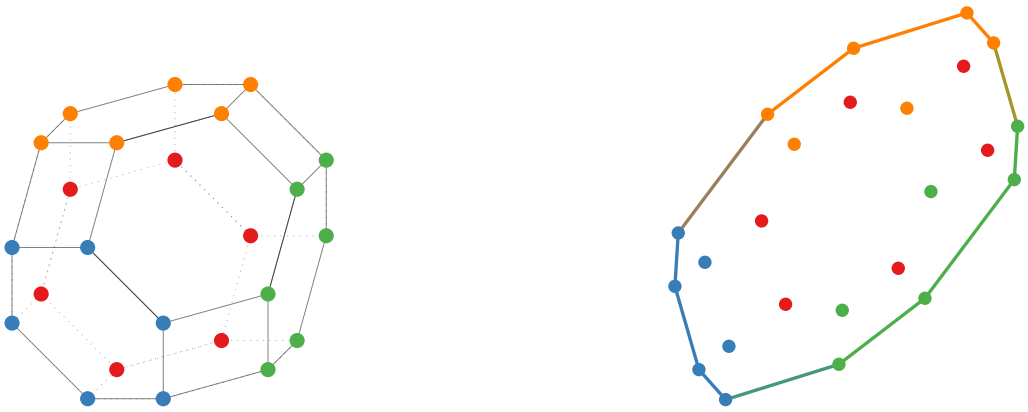
Figure 3.14: $\mathbf{W}$ (not drawn in the figure, see Section 3.5.1) projects $\mathbf{Y} = \text{verts}(\square_3)$ to $\mathbb{R}^2$, introducing linear dependencies. $\mathbf{y}_1$ and $\mathbf{y}_8$ are interior to the convex hull, and hence unargmaxable.

BSLs$_\triangle$ *must* have Unargmaxable Rankings In Fig. 3.15 we see the projection of $\hexagon$, which has a vertex per ranking of the 4 categories. As can be seen, only 12/24 vertices remain on the convex hull of the projection, analogously to Fig. 3.12.

BSLs$_\triangle$ may have Unargmaxable Categories We illustrate this in two cases: a) in Fig. 3.16 we see this directly, i.e. the vertex for the category is interior to the convex hull of the remaining categories, hence it is unargmaxable. b) in Fig. 3.15, this corresponds to the fact that all the vertices of the permutohedron that rank the target category above all other categories are interior to the convex hull.

Summary

- Multi-class classifiers *may* have unargmaxable categories.
- Multi-class classifiers *must* have unargmaxable rankings.
- Multi-label classifiers *must* have unargmaxable label assignments



(a) Permutohedron in $\mathbb{R}^3$.

(b) Convex Hull of projected vertices in $\mathbb{R}^2$.

Figure 3.15: Projecting $\hexagon_3$ to $\mathbb{R}^2$. The vertices are coloured according to the argmax. The vertices that rank c_4 above all others are all internal to the convex hull, and hence those rankings and c_4 are unargmaxable (see Proposition 3.1).

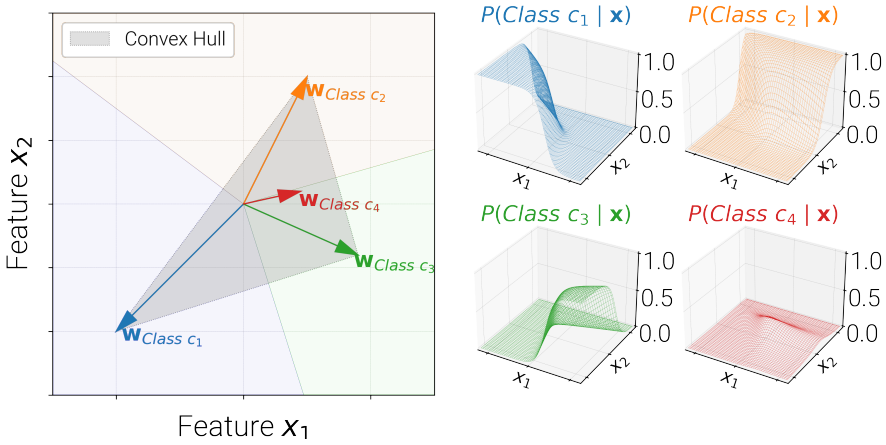


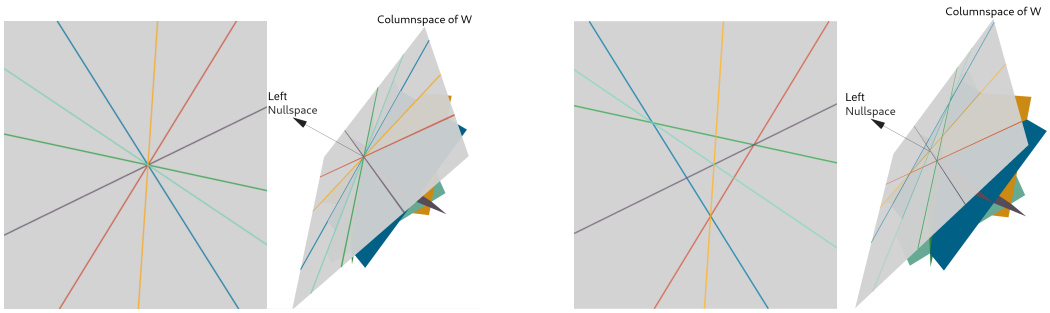
Figure 3.16: We revisit the $\mathrm{BSL}_\triangle$ from Fig. 1.2. The representation for c_4 is interior to the convex hull of the remaining category representations. Therefore c_4 is unargmaxable.

3.5.3 Affine Parametrisation

So far we have focussed on $\mathrm{BSL}(\mathbf{x}; \mathbf{W})$, i.e. BSLs that do not have a bias term. But what happens if we have an affine parametrisation, $\mathrm{BSL}(\mathbf{x}; \mathbf{W}, \mathbf{b})$?

Visually, the bias term offsets the $\mathrm{span}(\mathbf{W})$ in the direction of $\mathbf{b}$. Instead of restricting to a subspace, $\mathcal{P}\left(\circledast_n, \mathbf{W}\right)$, we restrict to an affine subspace. We illustrate the restriction of $\circledast$ for $\mathrm{BSL}_\triangle(\mathbf{x}; \mathbf{W})$ (left) and $\mathrm{BSL}_\triangle(\mathbf{x}; \mathbf{W}, \mathbf{b})$ (right) in Fig. 3.17.

Argmaxability-wise, not much changes: adding a bias term is less expressive than increasing the number of features by one. This is because we can embed d-affine space



(a) Input space $\mathbf{b} = \mathbf{0}$ (b) Output space $\mathbf{b} = \mathbf{0}$ (c) Input space $\mathbf{b} \neq \mathbf{0}$ (d) Output space $\mathbf{b} \neq \mathbf{0}$

Figure 3.17: Effect of bias term $\mathbf{b}$ on argmaxable permutations. Left: $\mathrm{BSL}_{\triangle}(\mathbf{x}; \mathbf{W})$. Right: $\mathrm{BSL}_{\triangle}(\mathbf{x}; \mathbf{W}, \mathbf{b})$. Having a bias term offsets the grey plane and allows it to avoid the origin. This increases the number of regions by creating bounded regions seen in subfigures c and d.

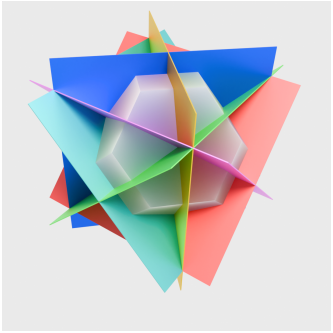
in $d+1$ linear space by fixing one of the input features to 1. For $\mathbf{W} \in \mathbb{R}^{n \times d}$, we have:

$$\mathbf{W}\mathbf{x} + \mathbf{b} = \begin{bmatrix} \mathbf{W} & \mathbf{b} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} \tag{3.61}$$

$$= \mathbf{W}'\mathbf{x}' \qquad \mathbf{W}' \in \mathbb{R}^{n \times (d+1)} \tag{3.62}$$

In terms of the LPs, we note that the change is straightforward, we can either implement the above constraint, or add the coefficients of the bias term to the inequalities. We will explicitly do the latter for MCC, see Eq. (4.9). For MLC, we will not use a bias term in Eq. (4.2), but we include it in the derivation in Appendix A.3.

In the next chapter we put our tools to use and detect unargmaxable outputs in classifiers used for MLC and MCC.



4 Detecting Argmaxability

In Section 3.5 we showed that:

- Bottlenecked Multi-label Classifiers, $\text{BSLs}_{\square}$
  - *must* have unargmaxable label assignments.
- Bottlenecked Multi-class Classifiers, $\text{BSLs}_{\triangle}$
  - *may* have unargmaxable categories.
  - *must* have unargmaxable rankings.

If our DNNs have unargmaxable outputs which are needed for our applications, this is an important problem. The presence of unargmaxable outputs would imply that the last layer of our expressive DNNs is hindering their generalisation capability. Importantly, unargmaxable outputs have further ramifications for the fairness, safety and reliability of our DNNs, as we explained in Section 1.1.2. *But do unargmaxable outputs impact DNNs in practice?* Are there any outputs which we need for our applications which end up being unargmaxable?

To answer this question, we fill a gap in the literature and build exact tools to detect unargmaxable outputs and use them to verify BSLs used for MLC (Section 4.1) and MCC (Section 4.2).

While we saw that unargmaxable outputs are related to LPs in Section 3.4, we have not yet provided a concrete implementation. In this chapter,

1. we construct detection algorithms by implementing these LPs.
2. we apply the detection algorithms to models to show that unargmaxable outputs exist in practice.

Two versions of the LP We introduce two LPs to detect:

- unargmaxable label assignments in MLC (Eq. (4.2)) and
- unargmaxable categories in MCC (Eq. (4.9))

As we argued in Section 3.4.3, we will rely on the hyperplane view, since it allows us to get more insights about the feature space. More specifically, we use the Chebyshev LP, which also estimates how large the argmaxable region is, if it exists.

Framing Experiments for MLC and MCC In what follows, we detect unargmaxable outputs for MLC and MCC models. Due to differences in the tasks and models, our approaches differ methodologically.

For MLC (Section 4.1), unargmaxable label assignments are only a problem if our application requires those label assignments. As such, we check whether the train, validation and test sets of well known MLC tasks can be predicted. Since there are fewer released models for MLC, and it is common that bespoke models are needed for each problem, we fine-tune MLC models on three widely used datasets. More specifically, we train models for varied bottleneck widths, in order to assess the impact of the bottleneck dimension on argmaxability.

On the other hand, for MCC (Section 4.2), we scrutinise text generation models (e.g. MT models and LLMs). Since text generation models can be used in a variety of downstream tasks where all vocabulary tokens may be needed, it is vital to show that all tokens can be generated. We will therefore limit our analysis to investigating if the tokens in the vocabulary can be produced for publicly released models when using argmax prediction, without considering specific test sets.

4.1 Multi-Label Classification

Sigmoid output layers are widely used in MLC tasks, in which multiple labels can be assigned to any input. Such sigmoid classifiers (Section 3.2.3) are simple to implement: just append a linear layer with sigmoid activations to your neural feature encoder of

choice. They are widely used in neural MLC with thousands of output labels; applications include clinical coding (Mullenbach et al., 2018), image classification (Baruch et al., 2020), fine-grained entity typing (Choi et al., 2018) and protein function prediction (Kulmanov et al., 2019). Moreover, they are the default for MLC in frameworks such as Scikit-learn (Pedregosa et al., 2011) and Keras (Paul et al., 2014).

However, there has been little analysis on the effect of using bottlenecked parametrisations for such models. While previous work has shown that BSLs are restricted in expressivity, they focused only on MCC (Ganea et al., 2019; Yang et al., 2018) and not MLC. But for MCC, as we will see in Section 4.2, the consequences are relatively minor: classes can be unargmaxable in theory (Demeter et al., 2020) but rarely are in practice (Grivas et al., 2022).

Contributions Our contribution here is to highlight that for $\text{BSLs}_{\square}$ the consequences are much more pronounced. As we saw in Section 2.4.4, $\text{BSLs}_{\square}$ must have an exponentially large number of unargmaxable label assignments, and as we will show here, unargmaxable label assignments do exist in practice.

To this end, we check the consequences of a $\text{BSL}_{\square}$ empirically. We train text classification and image classification models where we vary the bottleneck dimensionality in order to observe at what dimensionality we get unargmaxable validation and test set outputs. Since we build the output layer from scratch, we avoid the use of a bias term to avoid the nuisance of adjusting the LP to the affine case.

Notation We consider a MLC model that predicts a complete label assignment $\mathbf{y} \in \{+, -\}^n$ for a label vocabulary of size n, and where each $y_i \in \{+, -\}$ denotes if a single label is active ($+$) or inactive ($-$).

Bottlenecked Sigmoid Layer (BSL)

A linear sigmoid layer takes as input a feature vector $\mathbf{x} \in \mathbb{R}^d$ and predicts $\mathbf{y} \in \{+, -\}^n$ by assuming that all labels are independent given $\mathbf{x}$.[1] The idea is that a powerful feature encoder (Section 3.1.1) does the "heavy lifting" and projects inputs to meaningful embeddings in $\mathbb{R}^d$ such that they can be easily separated by n different hyperplanes. When n is large, due to computational constraints it is popular to realise such a layer

[1] To be more precise, our MLC loss function need not be motivated from an independence assumption, it would hold even if we chose to ignore the dependencies and fit the marginal label probabilities. However, we maintain this phrasing here for backwards compatibility with the paper.

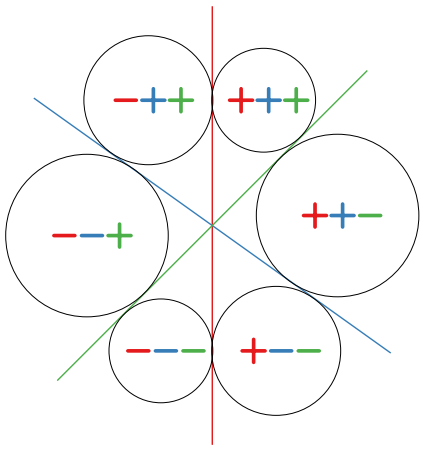


Figure 4.1: $n = 3$ binary classifiers with $d = 2$ dimensional input features. We include the balls found by the Chebyshev LP for each argmaxable label assignment. When $d \ll n$, most balls will have a tiny radius.

as a $\text{BSL}_{\square}$, which is parametrised by a bottlenecked $\mathbf{W} \in \mathbb{R}^{n \times d}$ and associates with each label the probability:

$$P(y_i \mid \mathbf{x}) = \begin{cases} \sigma_{\square}(\mathbf{w}_i^\top \mathbf{x}) & \text{if } \mathbf{y}_i = \texttt{+} \\ 1 - \sigma_{\square}(\mathbf{w}_i^\top \mathbf{x}) & \text{otherwise} \end{cases} \tag{4.1}$$

Here, $\sigma_{\square}()$ is the logistic sigmoid function and $\mathbf{w}_i$ is the weight vector of the i-th classifier (hyperplane), i.e. the i^{th} row of $\mathbf{W}$. Note that all $\mathbf{w}_i$ see the same shared $\mathbf{x}$. We focus on such a setup and discuss its limitations because it is the default in mainstream ML libraries such as scikit-learn (Pedregosa et al., 2011) and is largely used as a simple classifier (Baruch et al., 2020; Kulmanov et al., 2019; Mullenbach et al., 2018). We denote the whole multi-label classifier by the parametrisation of its last layer, e.g., we will say "a classifier $\mathbf{W}$", as our analysis is agnostic to the feature encoder, as discussed in Section 3.1.1.

4.1.1 Detecting Unargmaxable Label Assignments

Given a bottlenecked classifier, $\mathbf{W}$, we want to verify if a set of L label assignments of interest $\{\mathbf{y}^{(l)}\}_{l=1}^{L}$ is (un)argmaxable. These labels can belong to a held out set, as in our experiments (Section 4.1.2), and help quantify the generalisability and trustworthiness of the given classifier, as we would expect it to be able to predict all L outputs.

A simple strategy is to verify the argmaxability of each $\mathbf{y}^{(l)}$. For that, we use a Chebyshev Linear Programme (LP), which also gives us a proxy for the size of a region, as we explain next. The LP aims to find the Chebyshev center (Boyd et al., 2004, p.

417) of the region encoded by $\mathbf{y}^{(l)}$: the center of the largest ball of radius ϵ that can be embedded within it (see Fig. 4.1 and https://viz.unargmaxable.ai/chebyshev/ for a 3D visualisation of the regions and Chebyshev balls). As a constrained optimization problem (see derivation in Appendix A.3), we solve:

Label Assignment LP

$$\begin{aligned} \text{maximise} \quad & \epsilon && (4.2)\\ \text{subject to} \quad & -y_i \mathbf{w}_i^\top \mathbf{x} + \epsilon \|\mathbf{w}_i\|_2 \leq 0, \quad i \in [n] && (4.3)\\ & -10^4 \leq x_j \leq 10^4, \quad j \in [d] && (4.4)\\ & \epsilon > \text{eps} && (4.5) \end{aligned}$$

where we abuse notation and $y_i \in \{+1, -1\}$. We constrain each entry of $\mathbf{x}$ in a bounded region, since the Chebyshev center is not defined otherwise. We do not expect this constraint to significantly affect our conclusions, since neural network activations are also bounded in magnitude (see our analysis for softmax in Appendix A.5). If the LP is feasible it returns the maximum radius ϵ and we verify that $\mathbf{y}$ is argmaxable. Note that we add an additional constraint, $\epsilon >$ eps, where eps is within the numerical accuracy the LP solver can operate.[2]

Limits of Argmaxability Detection: Hyperplane Overcrowding

While we have defined argmaxability in absolute terms (Section 3.4), in practice it can only be verified up to some numerical precision: $\mathbf{y}$ could be argmaxable, but an LP may not be able to detect this if the neighbourhood around all representative $\mathbf{x}$ is tiny.

Such tiny neighbourhoods are indeed formed when we have many hyperplanes, in a problem we intuitively call **hyperplane overcrowding**. As we saw in Section 2.4.4, partitioning feature space with many hyperplanes creates an exponentially large number of regions. As a result, a fixed volume in feature space must be split into tiny shards. We illustrate a low-dimensional version of this problem in Fig. 4.2.

Hyperplane overcrowding can be a problem in two ways: a) detecting tiny regions is challenging due to the finite numerical precision of LPs and b) while the existence of a region suggests argmaxability, if the region is tiny, a neural network encoder is unlikely to be able to produce activations that fall within that region, making the corresponding output very hard to predict in practice.

[2] We use eps $= 10^{-8}$ since Gurobi Optimization (2021) has a minimum tolerance of 10^{-9}.

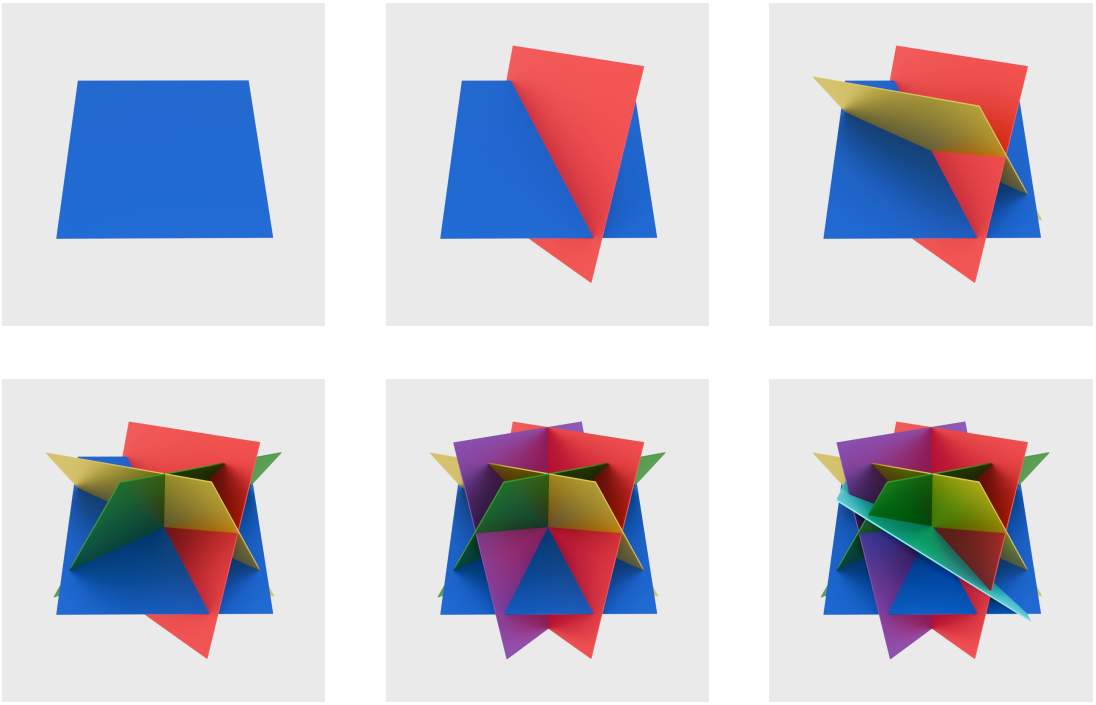


Figure 4.2: The problem of **hyperplane overcrowding**: as we progressively partition the output space with more and more hyperplanes (top left to bottom right) the regions become smaller and smaller. Outputs that correspond to small regions become hard for a neural network to predict with a large margin, if at all. In the extreme case, the regions become too small to detect within the numerical accuracy of a LP.

To quantify how often the regions become very small, we introduce the concept of ϵ-argmaxability, so that we also keep track of regions that are argmaxable with at least some margin ϵ. The margin we use is the radius ϵ of a ball that fits within the region of the hyperplane arrangement (Eq. (4.2)). Intuitively, ϵ-argmaxability characterises robustly argmaxable label assignments.

Epsilon Argmaxability

Def. 4.1.1. A label assignment $\mathbf{y}$ is **$\boldsymbol{\epsilon}$-argmaxable** for a classifier $\mathbf{W}$ if it is argmaxable under the presence of any noise vector $\boldsymbol{\delta}$ of magnitude $\|\boldsymbol{\delta}\|_2 \leq \epsilon$:

$$\mathbf{y} \text{ is } \epsilon\text{-argmaxable} \iff \forall \boldsymbol{\delta}, \|\boldsymbol{\delta}\|_2 \leq \epsilon, \quad \exists \mathbf{x} : \operatorname{sign}(\mathbf{W}(\mathbf{x}+\boldsymbol{\delta})) = \mathbf{y}.$$

Our Chebyshev LP is able to verify ϵ-argmaxability, and therefore argmaxability, as the first implies the second. Note, however, that the reverse is not true. Although verifying that a classifier is able to argmax a certain set of labels is of extreme importance, verification can be computationally expensive, as LPs become intractable as we

MLC Dataset	n	k	N	encoder	modality
MIMIC-III	8921	80	44k	CNN (e=500)	text
BioASQ task A	20000	50	100k	BERT (e=768)	text
OpenImagesV6	8933	50	108k	TResnet (e=2432)	images

Table 4.1: Statistics of the three MLC datasets we run experiments on, together with the feature encoder used for each dataset and its embedding dimensionality (e). We choose these datasets because we want a scenario where it is possible to make all outputs argmaxable even with a bottlenecked classifier. These datasets provide such a scenario, because despite having a very large label vocabulary (n), the number of active labels is bounded by k. As we will see in Section 5.1, there exists an output layer that guarantees that all k-active label assignments are argmaxable, so we verify BSLs make all k-active label assignments argmaxable by default.

scale n, d and L. In Section 5.1 we will see a way around this difficulty: we devise a classifier which guarantees that all labels of interest are argmaxable *by design*. BSLs, on the other hand, can have unargmaxable outputs, so we use the Chebyshev LP to scrutinise them after training them for various bottleneck widths on three datasets.

4.1.2 Experiments

In the experiments that follow we train a BSL and analyse how making a BSL narrower (i.e. shrinking the feature dimensionality) impacts the argmaxability of label assignments. We ask: *Do BSLs have unargmaxable labels in practice?* As we will see, when we shrink the feature dimensionality of a BSL to the low hundreds, there exist label assignments from the validation and test set examples that are unargmaxable.

BSL Output Layer

We now define the *BSL Layer* that is unconstrained and does not guarantee argmaxability of k-active outputs. We postpone the discussion of our *DFT layer*, which does, to the next chapter.

In our experiments we want to study the effect of varying the bottleneck width irrespective of the embedding dimensionality of the encoder which varies across datasets and models. As such, we introduce an *affine projection layer* (parametrised by $\mathbf{P}$ and $\mathbf{b}$) between the feature encoder and the *linear classifier* (parametrised by $\mathbf{W}$). For simplicity, we do not include bias terms for the classifiers. We compute the logits

$\mathbf{z} \in \mathbb{R}^n$ from the encoder activation $\mathbf{e} \in \mathbb{R}^e$ as:

$$\mathbf{z} = \mathbf{W}\mathbf{x}, \quad \mathbf{x} = \mathbf{P}\mathbf{e} + \mathbf{b} \tag{4.6}$$

For the BSL, the projection layer maps from e, the dimensionality of the encoder embeddings, to d, the feature dimensionality using $\mathbf{P} \in \mathbb{R}^{e \times d}$ and bias $\mathbf{b} \in \mathbb{R}^d$. This is followed by the bottlenecked classifier $\mathbf{W} \in \mathbb{R}^{n \times d}$.

Datasets

We work with three datasets, MIMIC-III, BioASQ Task A and OpenImages v6. We summarise their important attributes in Table 4.1. See Appendix A.6 for details on dataset construction and more dataset statistics. We use these datasets because they have been widely used in the literature and are a good test bed for argmaxability in MLC. This is because, a) they have thousands of output labels, so even if we use hundreds of features we will still have a bottlenecked classifier, and b) they have a bounded number of active labels, k, which is less than 100 for all three datasets. The latter constraint means that for these datasets it is possible to not have any unargmaxable outputs, even with a bottlenecked classifier, as we will show in Section 5.1. We now introduce our datasets in more detail.

Clinical Coding (MIMIC-III)

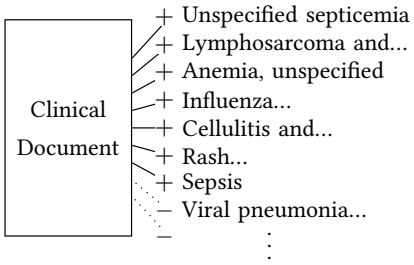


Figure 1.4: The above label assignment is **unargmaxable**; it can never be assigned to any input document for this model.

We start with MIMIC-III (Johnson et al., 2016) where the task is clinical coding. The goal is to tag each clinical note with a set of relevant ICD-9 codes which describe findings, recall Fig. 1.4 (reprinted) from Chapter 1. For this task, we retrain the CNN encoder model defined in Mullenbach et al. (2018) which has $n = 8921$ and $e = 500$. We vary the sigmoid bottleneck width d by altering the projection layer. We follow Mullenbach et al. (2018) and train on 47723 examples and the dev and test set comprise 1631 and 3372 examples, respectively. We use the same word embeddings, preprocessed data, data splits, metrics (Prec@8) and hyperparameters reported in the paper (Mullenbach et al., 2018). We

only change the learning rate of the Adam optimiser to 0.001, as this improves results (as also found by Edin et al. (2023)).

Semantic Indexing (BioASQ Task A)

Next, we focus on the 2021 BioASQ semantic indexing challenge (Krithara et al., 2023; Nentidis et al., 2021; Tsatsaronis et al., 2015). For this task, we are given PubMed abstracts and asked to predict a set of relevant MeSH headings[3] for each article. We create dataset splits (see Appendix A.6.2 for details) with $n = 20000$, making sure that all individual labels occur in both the train and test sets. We do so to avoid claiming a label assignment is unargmaxable when in fact it would be hard to predict the individual labels that constitute it. We use $k = 50$ as this is the maximum number of active labels per example for this dataset by construction. We finetune PubMedBERT (Gu et al., 2021), a domain specific uncased BERT (Devlin et al., 2019) encoder that has been pretrained on PubMed abstracts and has $e = 768$. We use early stopping with a patience of 10 on the validation cross-entropy loss.

Image MLC (OpenImages v6)

We use the OpenImages v6 dataset (Kuznetsova et al., 2020) where the goal is to tag each image with objects that appear in it. Similar to the BioASQ case, we choose the label vocabulary $n = 8933$ such that all examples in the train, validation and test set are covered. We pick $k = 50$ since the training data has at most 45 active labels per example. We finetune the TResnet (Ridnik et al., 2020) with $e = 2432$ that Baruch et al. (2020) pretrained for MLC on this dataset. We use early stopping with a patience of 10 on the validation cross-entropy loss.

Argmaxability Metrics

We now introduce two fine-grained measurements of argmaxability, eps-argmaxability and 1-argmaxability. We discuss a few additional insights in the results section below.

eps-argmaxable This is the estimate of argmaxability we can get at the precision of our LP, which is eps=10^{-8}. This roughly means that we can only detect regions which can contain a ball with a radius that is larger than 10^{-8}.

[3] https://www.nlm.nih.gov/mesh/meshhome.html

1-argmaxable As we discussed in Section 4.1.1, some label assignments may be ϵ-argmaxable with a very small ϵ, which makes it hard to predict them in practice. To get a better estimate for the number of such cases, we also report 1-argmaxability, i.e. argmaxable regions that have a radius greater than 1. This is helpful since any label assignments with $\epsilon < 1$, while argmaxable, are unlikely to be predicted in practice by the models we use. For example, we found that predicted label assignments for most models had regions of radius $\epsilon > 100$.

Interestingly, for BSLs, if an output is eps-argmaxable it is also 1-argmaxable. We will therefore not encounter 1-argmaxability again until the next chapter, where we will use it to contrast the behaviour of BSLs and the layer that we introduce in Section 5.1, in order to highlight that it is not enough to have outputs that are argmaxable; they need to be argmaxable with a reasonable margin.

Results

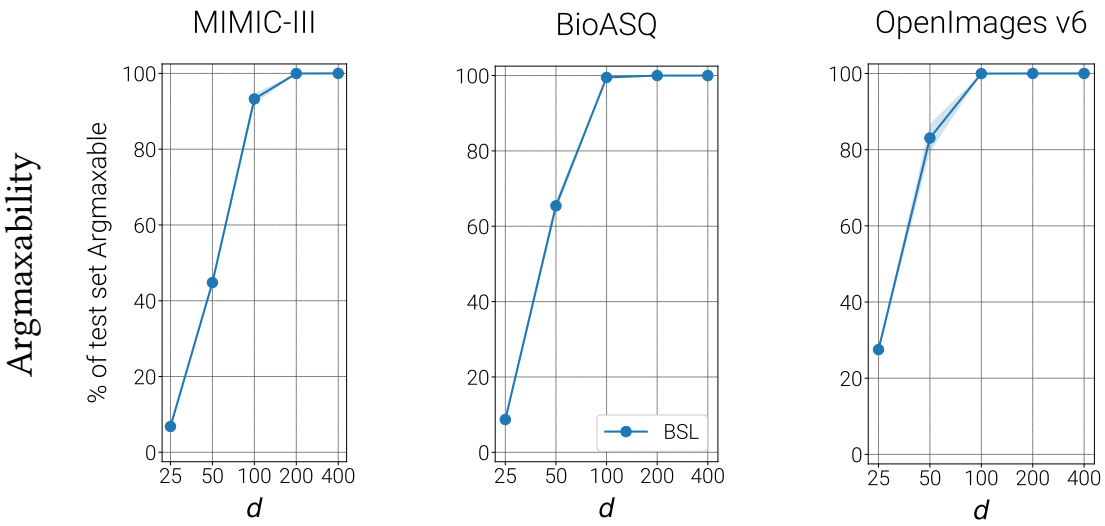


Figure 4.3: We use the LP to verify the BSL on the test sets of three datasets. We plot the percentage of test set examples that are argmaxable (y-axis) as we vary d, the number of trainable dimensions in the BSL (x-axis). We plot the mean and std (shaded) over 3 runs with different random seeds. As can be seen, BSLs have unargmaxable label assignments when d is small enough (i.e. < 200).

BSL: Unargmaxable outputs. We use the LP to verify the BSL on the test sets. As can be seen in Fig. 4.3, for $d > 200$ all the examples in the test set are argmaxable for the BSL. However, as we reduce d, the number of unargmaxable label assignments increases for all datasets. More specifically, we first get unargmaxable

outputs at $d = 200$ for MIMIC-III. Analogous considerations can be drawn for BioASQ and OpenImages (Fig. 5.6(a)). We tabulate the precise number of argmaxable label assignments on the datasets in Table 4.2. As such, we conclude that unargmaxability can indeed be an issue when we have hyperplane overcrowding and d is not large enough. Note that for a BSL one can never determine a d that always guarantees all label configurations of interest to be argmaxable: even an exhaustive verification on all test samples does not imply that future unseen configurations will be argmaxable.

d	MIMIC-III	BioASQ	OpenImages v6
25	229	880	2758
50	1515	6498	8439
100	3137	9951	9997
200	3370	10000	10000
400	3371	10000	10000

Table 4.2: Median number of eps-argmaxable label assignments over 3 random seeds on the test set of each dataset, as we vary d, the number of trainable dimensions of the BSL. We use orange to highlight the presence of unargmaxable label assignments and green to signal that the whole test set is argmaxable. Takeaway: BSL layers have unargmaxable label assignments starting from $d = 200$. We do not report 1-argmaxable label assignments here since the numbers are almost the same, we will discuss these in Table 5.1 in the next chapter.

4.2 Multi-Class Classification

We now turn from detecting unargmaxable label assignments in MLC to detecting unargmaxable categories in MCC. More precisely, we will consider publicly available LLMs and MT models and ask: *Do unargmaxable vocabulary tokens exist in large models used in practice?*

Demeter et al. (2020) showed that a softmax bottlenecked LM "steals" probability from rare words when contrasted to the probabilities assigned by a smoothed n-gram LM. More specifically, they proved that a category is unargmaxable if its softmax weight vector is interior to the convex hull of the remaining weight vectors. They proposed an algorithm to detect unargmaxable tokens and provided evidence of their existence in small LMs, but their proposed algorithm provided no guarantees and they were unable to test large LMs.

Contributions Our contribution in this section is three-fold.

1. we introduce terminology to disentangle the idea of unargmaxability from the more general question of loss in expressivity due to a $\mathrm{BSL}_{\triangle}$.

2. we introduce a detection algorithm for unargmaxable categories that is unambiguous and also works when we have a softmax bias term.

3. we test 7 LLMs and 143 MT models. Out of those, we find that 13 of the MT models exhibit unargmaxable tokens.

4.2.1 Detecting Unargmaxable Tokens

We now introduce an algorithm to detect unargmaxable categories (i.e. tokens) in LLMs and MT models. The algorithm combines an incomplete algorithm, for efficiency, with an exact algorithm for correctness, see Fig. 4.4(c).

We consider a MCC problem with n categories, i.e. n is the vocabulary size of a LLM or MT model. Given a bottlenecked softmax layer parametrised by $\mathbf{W} \in \mathbb{R}^{n \times d}$ and $\mathbf{b} \in \mathbb{R}^n$, are there any categories that are unargmaxable? We first describe a slow, but exact algorithm to answer this question, and then introduce the incomplete algorithm that we can use to get a faster answer in some cases.

Exact Algorithm

An exact algorithm will either prove category c_t is argmaxable by returning a feasible point $\mathbf{x} : \operatorname{argmax}(\mathbf{W}\mathbf{x} + \mathbf{b}) = c_t$ or it will prove c_t is unargmaxable by verifying no such point exists.

As we discussed in Section 3.4.4, to check if a region exists that ranks c_t above all others, we need to find an input $\mathbf{x} \in \mathbb{R}^d$ that satisfies the following constraints:

$$P(c_i \mid \mathbf{x}) < P(c_t \mid \mathbf{x}), \quad \forall i \in [n] : \quad i \neq t, \quad t \in [n] \tag{4.7}$$

Each of the above constraints is equivalent to restricting $\mathbf{x}$ to a halfspace (Definition 2.4.2). Hence, if all above inequalities are enforced, $\mathbf{x}$ is restricted to an intersection of halfspaces (Definition 2.4.3).

$$\begin{aligned} (\mathbf{w}_{c_i} - \mathbf{w}_{c_t})^\top \mathbf{x} + (b_{c_i} - b_{c_t}) < 0 \\ \forall i \in [n] : \quad i \neq t, \quad t \in [n] \end{aligned} \tag{4.8}$$

If the intersection of halfspaces is empty, there is no $\mathbf{x}$ for which category c_t can be ranked above all others; hence c_t is unargmaxable. We can find a point in an intersection of halfspaces via linear programming, albeit we found this algorithm to be slow in practice for $n > 1000$.

Chebyshev Center Linear Programme As in Section 4.1, we introduce a Chebyshev LP. However, this time we detect unargmaxable categories, in our case tokens of a LM or MT system for a $\mathrm{BSL}_{\triangle}$ parametrised by $\mathbf{W} \in \mathbb{R}^{n \times d}$ and $\mathbf{b} \in \mathbb{R}^n$.

Category LP

$$
\begin{aligned}
\text{maximise} \quad & r \\
\text{subject to} \quad & \mathbf{w}_i^\top \mathbf{x} + b_i + r\|\mathbf{w}_i\|_2 \leq 0, \quad i \in [n-1] && (4.9)\\
& -100 \leq x_j \leq 100, \quad j \in [d] && (4.10)\\
& r > \text{eps} && (4.11)
\end{aligned}
$$

Where $\mathbf{w}_i = \mathbf{w}_{c_i} - \mathbf{w}_{c_t}$ are rows of the braid matrix Eq. (2.52) and $b_i = b_{c_i} - b_{c_t}$, $\forall i \in [n] : c_i \neq c_t$, and eps is 10^{-8}. We further constrain $\mathbf{x}$ to guarantee the regions are bounded, since the Chebyshev center is not defined otherwise. This constraint also captures the fact that neural network activations are not arbitrarily large, as we verified for feature encoders in Appendix A.5.

If the above linear programme is feasible, we know that category c_t is argmaxable and we also get a lower bound on the volume of the region for which it is solvable by inspecting r. On the other hand, if the linear programme is infeasible, c_t is not eps-argmaxable.

We note a few differences between this LP and our LP for MLC (Eq. (4.2)). For categories we did not notice any regions become extremely small, mostly because the Voronoi Tesselation $\mathcal{P}\left(\triangle_n, \mathbf{W}\right)$ has n regions at most. On the other hand, for label assignments, $\mathcal{P}\left(\boxplus_n, \mathbf{W}\right)$ has an exponentially large number of regions, which means they can be much smaller. Therefore, for the MLC we used the larger bound $-10^4 \leq x_i \leq 10^4$, since this made the balls larger and allowed us to detect regions with smaller radii.

Approximate Algorithm

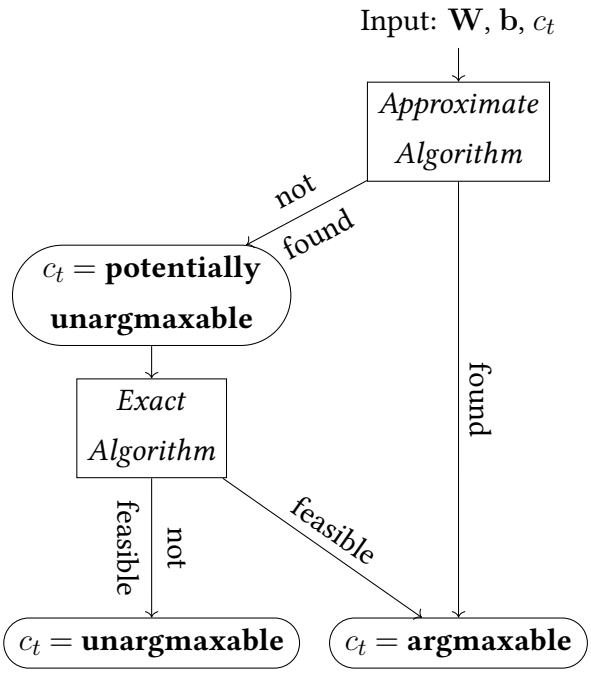
The exact algorithm was too slow to run for the whole vocabulary. In order to avoid running the exact algorithm for every single vocabulary item, we developed an in-

: Braid Reflect

Data: Category index c_t,
$\mathbf{x} \in \mathbb{R}^d$,
$\mathbf{W} \in \mathbb{R}^{n \times d}$, $\mathbf{b} \in \mathbb{R}^n$

1 $c_i = \text{argmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$
2 $\mathbf{w} = (\mathbf{w}_{c_t} - \mathbf{w}_{c_i})^\top$
3 $b = b_{c_t} - b_{c_i}$
4 $\mathbf{w}' = \frac{\mathbf{w}}{\|\mathbf{w}\|_2}$
5 $d = \mathbf{w}'^\top \mathbf{x}$
6 $\mathbf{x} = \mathbf{x} - 2(d + \frac{b}{\|\mathbf{w}\|_2})\mathbf{w}'$

(b) Approximate Algorithm: Move $\mathbf{x}$ to region where $P(c_t) > P(c_i)$.



(c) Combined detection algorithm.

Figure 4.4: Left: Braid Reflect Algorithm: an approximate algorithm used to detect whether category c_t is argmaxable. Right: How we combine approximate and exact algorithms to get a complete algorithm. We first run the approximate algorithm, which quickly proves most vocabulary tokens are argmaxable. If it fails to find a solution in N steps, we rely on the exact algorithm to either find a solution or prove there is no solution, meaning c_t is unargmaxable.

complete algorithm (Kautz et al., 2009) with a one-sided error, which can quickly rule out most tokens, leaving only a small number to be checked by the exact algorithm. It proves that c_t is **argmaxable** by finding an input $\mathbf{x}$ for which c_t has the largest activation. Unlike the exact algorithm, if no solution exists it cannot prove that the token is **unargmaxable**. Hence, we terminate our search after a predetermined number of steps. We denote any tokens not shown to be argmaxable by the approximate algorithm as **potentially unargmaxable** and we run the exact algorithm on them. An illustration of the way we combine the exact and approximate algorithms to decide whether category c_t is argmaxable can be seen in Figure 4.4(c).

Braid Reflect The idea behind this approximate algorithm is to use the Braid Hyperplane Arrangement (Definition 2.4.8) as a map to guide us towards a point $\mathbf{x}$ for which c_t has the largest activation.

To show that category c_t is argmaxable, it suffices to find an input $\mathbf{x}$ for which the largest probability is assigned to c_t. Empirically we found this to be easy for most

categories.

We begin by interpreting the actual weight vector as the candidate input $\mathbf{x} = \mathbf{w}_{c_t}^\top$. We do so since the dot product of two vectors is larger when the two vectors point in the same direction.[4] While the magnitude of the vectors affects the dot product, we found the above initialisation worked well empirically. When c_t is not the argmax for $\mathbf{x}$ and c_i is instead, Relation 4.8 for c_i and c_t will have the wrong sign. The sign of this relation defines which side of the Braid hyperplane for c_i and c_t we are on. To correct the sign, we construct the normal vector and offset of the Braid hyperplane (Lines 2,3 in Fig. 4.4(b)), compute the distance of $\mathbf{x}$ from it (Line 5), and reflect $\mathbf{x}$ across it (Line 6).[5] We repeat the above operation until either c_t is the argmax or we have used up our budget of N steps.

4.2.2 Experiments

In this section we use the combined algorithm from Figure 4.4(c) to search models for unargmaxable tokens. In contrast to the previous section where we limited our analysis to specific test sets, here we exhaustively check that all tokens in the vocabulary of our target models can be produced. This is because text generation models can be used in a variety of downstream tasks where all vocabulary tokens may be needed, so it is vital to show that all tokens can be generated.

Moreover, we do not train models from scratch here since a) there is a plethora of models which are publicly available and widely used, and b) we want to check a large variety of models and training this many models would be too expensive.

We test 7 LMs and 143 MT models. We find that unargmaxable tokens occur in 13 MT models, but these are mostly infrequent and noisy vocabulary tokens. We therefore do not expect such tokens to affect translation quality per se.

We also find that nearly all vocabulary tokens of LMs and student MT models can be verified with less than $N = 10$ steps of the approximate algorithm. In contrast, other MT models need thousands of steps and also rely on the exact algorithm. In this sense, models that need fewer steps are easier to verify: the search problem for their arrangement of softmax weights is easier.

Throughout the following experiments we assumed the softmax inputs were bounded in magnitude for all dimensions $-100 \leq x_i \leq 100$. Even though the outputs are not theoretically bounded, they are practically bounded since neural network

[4] $\mathbf{a}^\top \mathbf{b} = \|\mathbf{a}\|_2 \|\mathbf{b}\|_2 \cos\theta$ is maximised for $\theta = 0$

[5] The reflection operation without the offset is the Householder transformation (Householder, 1958).

feature encoders cannot produce arbitrarily large activations and some regions may be unreachable[6]. For the approximate algorithm, we search for a solution with a patience of $N = 2500$ steps and resort to the exact algorithm if the approximate method fails or returns a point outside the aforementioned bounds. We use Gurobi (Gurobi Optimization, 2021) as the linear programme solver. We accessed the model parameters either via NumPy (Harris et al., 2020) or PyTorch (Paszke et al., 2019). The experiments took 3 days to run on an AMD 3900X 12-core CPU using 10 threads and 64Gb of RAM.

Language Models (0/7 Unargmaxable)

We checked 7 widely used LMs for unargmaxable tokens. While some of these models such as BERT (Devlin et al., 2019) are not directly used for generation, large LMs are used via prompting (F. Liu et al., 2021) for few shot learning. A prompt model obviates the need for a separate classifier by rephrasing a classification task as slot filling given a task specific template. Prompt approaches commonly choose the answer for the slot by argmaxing the softmax distribution obtained by a LM. Hence we verify that there are no answers that are unargmaxable.

BERT (Devlin et al., 2019), RoBERTa (Y. Liu et al., 2019), XLM-RoBERTa (Conneau et al., 2020) and GPT2 (Radford et al., 2019) did not exhibit any unargmaxable tokens and can be assessed without resorting to the exact algorithm. Moreover, the LMs were very easy to verify with the approximate algorithm requiring less than 1.2 steps per token on average.

Machine Translation (13/143 Unargmaxable)

In the case of MT models, the feature encoder comprises the whole encoder-decoder network excluding the last layer of the decoder. We first focus on models which we found to have unargmaxable tokens and then briefly describe models that did not. A summary of the results and characteristics of the models we checked can be seen in Table 4.3. More detailed results can be found in Tables A.1, A.3, A.4 and A.5 in the Appendix.

Helsinki NLP OPUS (13/32 Unargmaxable). The 32 models we use for this subset of experiments are MT models released through Hugging Face (Wolf et al., 2020). We use models introduced in Tiedemann et al. (2020). These models are trained

[6]The validity of our assumption is only relevant for models we find to be bounded. We therefore verified that $-100 \leq \mathbf{x} \leq 100$ holds for two of them, see Appendix A.5.

[7]https://github.com/browsermt/students

model source	Helsinki	FAIR	Edinburgh	Bergamot
unargmaxable	13/32	0/4	0/82	0/25
dataset	OPUS	WMT'19	WMT'17	multiple[7]
architecture	Transf	Transf	LSTM	Transf
feature dim d	512	1024	500,512	256,512,1024
softmax bias	✓	✗	✓	✓
tied embeds	enc+dec+out	dec+out	dec+out	enc+dec+out

Table 4.3: Number of models having unargmaxable tokens for the MT models we verified. We tabulate the ratios of models that have unargmaxable tokens. For example, in the first column we show that 13 out of the 32 models we checked had unargmaxable tokens.

on subsets of OPUS. All models are transformer models trained using Marian (Junczys-Dowmunt et al., 2018). They include a bias term, have a tied encoder and decoder and $d = 512$.

Unargmaxable tokens, if present, will affect generation in the target language. We therefore restrict our analysis to the target language vocabulary. To facilitate this, we inspect translation models for which the source and target languages have different scripts. We explore 32 models with source and target pairs amongst Arabic (ar), Hebrew (he), English (en), German (de), French(fr), Spanish (es), Finnish (fi), Polish (pl), Greek (el), Russian (ru), Bulgarian (bg), Korean (ko) and Japanese (ja). We rely on the script to disambiguate between source and target language and discard irrelevant tokens from other languages. We also ignore vocabulary tokens containing digits and punctuation.

In Figure 4.5 we can see the number of Byte Pair Encoding (BPE; Sennrich et al., 2016) tokens that were unargmaxable for these models, sorted in decreasing order. As can be seen, all tokens are argmaxable for $19/32$ language pairs. For the remaining 13 languages, while there can be quite a few unargmaxable tokens, most would not be expected to affect translation quality.

Out of the set of 427 unique unargmaxable BPE tokens, $307/476$ are single character subword tokens and only 2 are word stem BPE segments: *erecti* (bg-en) and ***Предварительны*** (en-ru) which means "preliminary" in Russian. The rest include the *<unk>* token and noisy subword unicode tokens such as ќЌЌќ, ìîì and ὰὺῆ.

On closer inspection of the SentencePiece tokeniser we found that both ***Предварительны*** and *erecti* come up as tokenisation alternatives that make them rare and irregular. We found that the ***Предварительны*** token was rare since it is capitalised

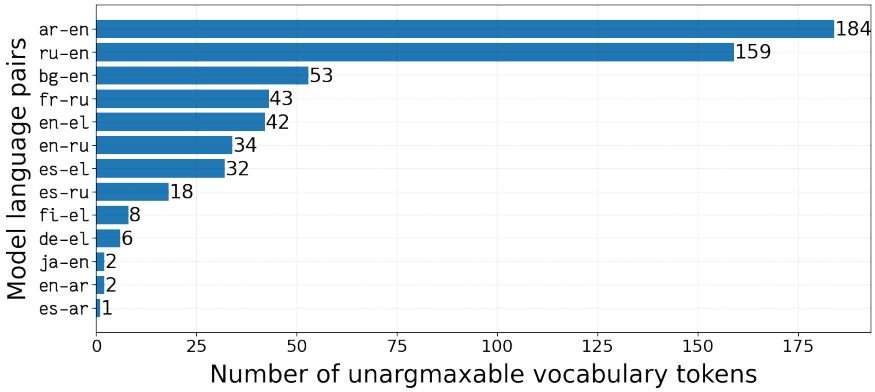


Figure 4.5: 13/32 HelsinkiNLP models have unargmaxable tokens.

and only occurs once, while another occurrence was caused by a BPE segmentation corner case due to Unicode token variation of *Предварительны-е*. Other mentions having *Предварительны* as a substring were split differently. In a similar vein, we found that the *erecti* token occurred due to BPE corner cases for *erecti-0-n*, *erecti-lis-)*, *erecti-l*, *erecti-.* and erecti-cle many of which are misspellings or rare word forms from clinical text. As such, the impact of these tokens being unargmaxable is small since there are alternative ones the MT model can prefer over them which could even correct spelling mistakes.

FAIR WMT'19 (0/4 Unargmaxable). We checked 4 FAIR models (en-ru, ru-en, en-de, de-en) submitted to WMT'19 (Ng et al., 2019). These transformer models have $d = 1024$ and do not employ a softmax bias term.

None of the FAIR models were found to have unargmaxable tokens, but for some tokens we had to rely on the exact algorithm to show this.

Edinburgh WMT'17 (0/82 Unargmaxable). These WMT'17 submissions (Sennrich et al., 2017) were ensembles of left-to-right trained models (l2r) and right-to-left trained models (r2l). These were LSTMs trained with Nematus using $d = 500$ or $d = 512$ and softmax weights tied with the decoder input embeddings. The models include a bias term.

None of the models have unargmaxable tokens. However, we found that models that comprise an ensemble varied a lot in how easy it was to show that the vocabulary was argmaxable, despite them differing solely in the random seed used for weight initialisation. As an example, zh-en.l2r(1) had 8 tokens that needed to be verified with the exact algorithm, zh-en.l2r(2) had 3 and zh-en.l2r(3) had 366. This highlights that random initialisation alone is enough to lead to very different arrangements of

softmax weights.

Bergamot (0/25 Unargmaxable). The Bergamot project[8] model repository contains both large transformer-base and transformer-big teacher models, as well as small knowledge distilled (Kim et al., 2016b) student models. Student models have $d = 256$ (tiny) or $d = 512$ (base), while teacher models have $d = 1024$. Interestingly, we find that it is easier to show that student models are argmaxable when compared to teacher models, despite student models having softmax weights $1/2$ or $1/4$ the dimensions of the teacher model.

4.2.3 Discussion

We conclude from our experiments that $\mathrm{BSLs}_{\triangle}$ *can have unargmaxable tokens, but this rarely occurs in practice* for tokens that would lead to irrecoverable errors in the MT models we checked. A limitation of our conclusions is that beam search is usually preferred over greedy decoding for MT models used in practice. We leave the question of how unargmaxable tokens impact beam search for future work.

It is challenging to make exact claims about what can cause tokens to be unargmaxable because the models we tested varied in so many ways. However, we outline some general trends below.

Infrequent Tokens Are the Victims The most general observation is that the tokens that are more likely to be unargmaxable or are hard to prove to be argmaxable are the infrequent ones. This can be seen in Figures 4.6 and 4.7, where the x-axis contains the vocabulary of the models sorted left to right by increasing frequency. Each dot on the y-axis represents the number of steps needed to check whether a token is argmaxable or not using the approximate algorithm. As can be seen, the values to the right are generally much higher than those to the left.

This result is in line with previous work that highlights the limitations of the softmax layer when modelling rare words for LMs (Chen et al., 2016; Labeau et al., 2019) and MT (Nguyen et al., 2017; Raunak et al., 2020) and infrequent classes for image classification (Kang et al., 2020).

Some Models are Easier to Verify than Others As we have mentioned previously, our approximate algorithm attempts to prove a class is not bounded by searching for an input as proof and giving up after a predetermined number of steps. We interpret

[8] https://browser.mt

the number of steps needed as a measure of how hard it is to prove that a class is unbounded when this is the case. We found that for the LMs and student MT models, the vocabularies can be shown to be argmaxable with one step of the approximate algorithm on average. On the other hand, for Helsinki NLP and FAIR MT models more than 10 steps were needed, as we show in Fig. 4.7.

To put the above observations into context, we also check the behaviour of our algorithms on randomly initialised parameters. If we initialise a softmax layer of $n = 10000$ classes using a uniform distribution $U(-1, 1)$ we do not expect unargmaxable tokens to exist after $d = 30$ (see Figure A.6 in the Appendix). Moreover, any randomly initialised parameters can be checked using the approximate algorithm with fewer steps as we increase d.

From this perspective, it is surprising that student models were easier to show to be argmaxable than the teacher models, despite the softmax weight dimensionality of the student models being much lower (256 for tiny, versus 1024 for teacher). This shows that effective neural MT models do not need to be hard to verify, but nevertheless neural models trained on the original data can converge to arrangement of weights that are hard to verify.

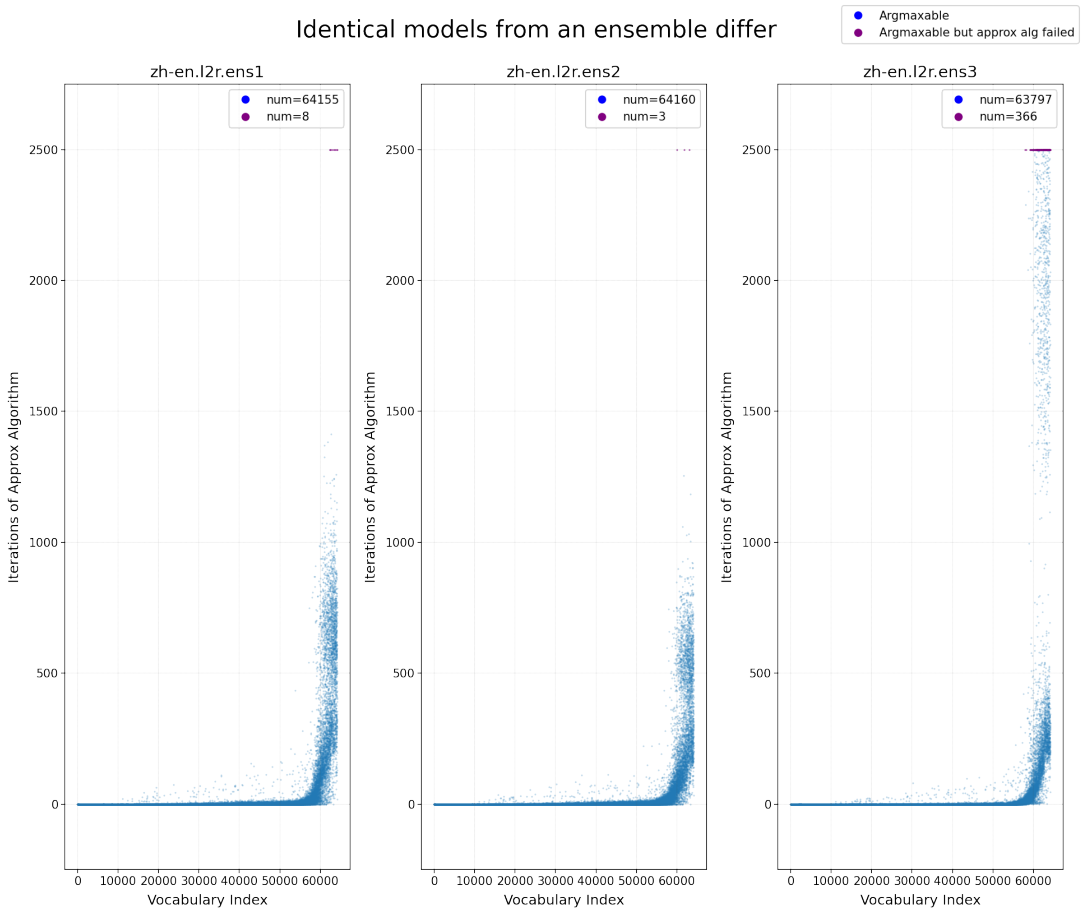
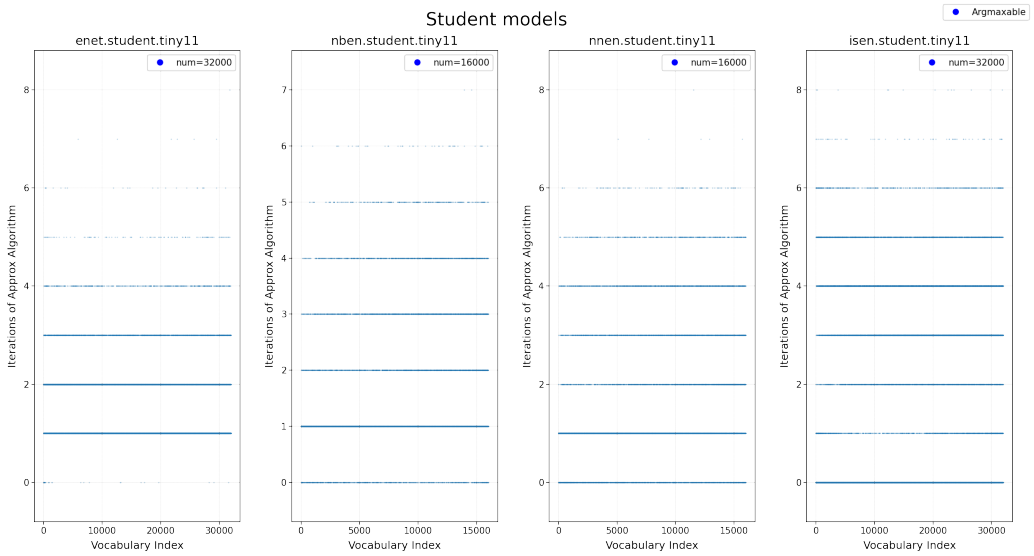
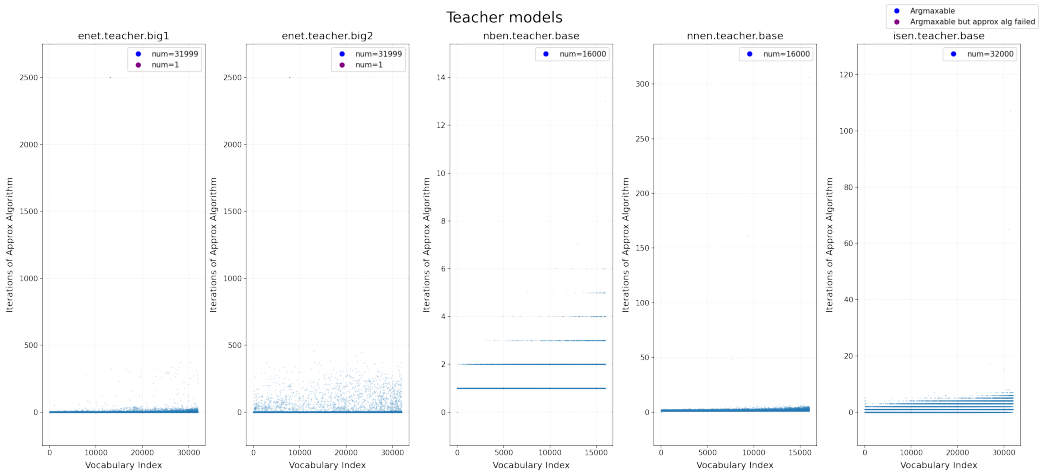


Figure 4.6: Some models of an ensemble are harder to verify than others. On the x-axis we have an entry per vocabulary token with the tokens sorted from most frequent (left) to least frequent (right). On the y-axis, we plot a blue dot at the number of iterations it took to show the token is argmaxable using the approximate algorithm, and a purple dot if the token is argmaxable but the approximate algorithm failed. We plot these values for three models of an ensemble, where each model differs only by the random seed used for initialising the model parameters. As can be seen, the right-most figure has 366 vocabulary tokens that are argmaxable but the approximate algorithm fails to find a solution, compared to 8 and 3 for the other two models.



(a) Student models are easy to verify, as can be seen from the small number of iterations (y-axis) needed to show that all tokens are argmaxable.



(b) Teacher models are harder to verify than student models. This can be seen on the y-axis where we need a much larger number of iterations than for the student models.

Figure 4.7: Student models are easier to verify than teacher models. On the x-axis we have an entry per vocabulary token. We plot a blue dot on the y-axis at the number of iterations it took to show the token is argmaxable using the approximate algorithm. We plot the values for 4 student models in the top subplot and 5 teacher models in the bottom subplots, where the teacher student pairs have the same name. As can be seen by the smaller values on the y-axis, student models are easier to verify. This is surprising given that student models have narrower bottlenecks.

4.3 Conclusions

We began this chapter by asking: *Do unargmaxable outputs impact DNNs in practice?*

Tools for Verifying BSLs

To answer this question, we have filled a gap in the literature by introducing tools to detect unargmaxable outputs in BSLs and released our code so that others can also verify their models. Having tools to verify BSLs is important, because although verifying a large set of models gives us some intuition about how often unargmaxable outputs occur, we cannot draw certain conclusions about other models. For example, even though we did not find unargmaxable tokens in LLMs, we cannot be sure that future LLMs, or the same architectures with different fine-tuned parameters, will not have unargmaxable tokens. As we saw in Fig. 4.6, even models that vary only by initialisation can be harder to verify than others. Moreover, we cannot simply rely on random matrix theory (Donoho et al., 2006), since for a random $\mathbf{W}$ with large enough d the probability of having unargmaxable outputs should be very close to zero (see also Fig. A.6). However, we showed that training $\mathbf{W}$ via stochastic gradient descent does produce $\mathbf{W}$ that have unargmaxable outputs in MT models and MLC models, as we discuss further next.

BSLs can cause Unargmaxable Outputs

Returning to our question, our experiments (Sections 4.1.2 and 4.2.2) confirm that unargmaxable outputs are not only a problem in theory, they can also occur in practice and they can impact outputs needed for our applications.

For MLC, the result is more pronounced, since BSLs have unargmaxable test set examples when we make the bottleneck narrow. For example, the $\mathrm{BSL}_{\square}$ we trained on MIMIC-III has more than 200 unargmaxable test set label assignments when $d = 100$ (Table 4.2). In real world terms, there are more than 200 clinical reports that our model would be unable to label with the correct findings.

On the other hand, for MCC, while none of the LLMs had unargmaxable tokens, some MT models did. However, this does not occur as often as for MLC in our experiments and the tokens affected are unlikely to be needed for our applications, as they are both noisy and rare, as we discuss next.

BSLs Impact Infrequent Outputs

Both for MLC and MCC, we note that the victims of unargmaxability are the outputs that are in the long tail.

For MLC, the unargmaxable label assignments have a large number of active labels. These are rare in the following sense: the more we increase the number of active labels in an assignment, the less likely it is that we have observed large subsets of those labels together in the data.

For MCC, similarly, unargmaxable tokens were infrequent but also noisy tokens. The two longest tokens arised from tokenisation corner cases while shorter ones were noisy tokens that should not create discernible differences in translation quality.

BSLs Hinder the Generalisation of DNNs

We conclude that while BSLs mostly seem to do the right thing, they are intrinsically unreliable. When we make BSLs narrow enough, they can cause unargmaxable outputs making our DNNs unable to generalise. Importantly, this means our DNNs are restricted irrespective of the expressivity of their feature encoder. As such, practitioners should be aware that choosing the feature dimension of a BSL is not an innocuous hyperparameter search: by reducing the rank of $\mathbf{W}$ they are potentially harming their DNNs by making some outputs impossible to predict.

In the next chapter (Chapter 5), we show how to make BSLs reliable by guaranteeing that all outputs needed for our application are argmaxable by construction.

5 Guaranteeing Argmaxability

John Conway: 196883*, that's the dimension of the space it lives in.*
Interviewer: It seems so arbitrary.
John Conway: Ooh no, it's not arbitrary, hah, no, it's got to be 196883
[pauses], yes, $47 \times 59 \times 71$.

—John Horton Conway, on the Monster Set

In Chapter 4 we saw that BSLs are intrinsically unreliable: they can have unargmaxable outputs, hindering the generalisation capabilities of our DNNs. In this chapter we show how to prevent this problem by aligning the unargmaxability constraints of BSLs to the constraints of the application at hand.

As we showed in Section 3.5, $\text{BSLs}_{\square}$ *must* have unargmaxable label assignments and $\text{BSLs}_{\triangle}$ *must* have unargmaxable rankings. Therefore, for applications where all such outputs are candidates, some must be unargmaxable. However, for most applications we have domain knowledge which constrains the outputs to a smaller set of candidates. For example, in MLC, inputs are rarely assigned more than k labels. In such a scenario, we can make a BSL reliable by guaranteeing that the argmaxable outputs subsume all candidates.

We cannot obtain such guarantees if we learn $\mathbf{W}$ without imposing additional constraints. This is because it is intractable to compute which outputs are argmaxable for a general $\mathbf{W}$. In Section 4.2 we got away with exhaustively checking all categories because there were only n of them. However, for label assignments and rankings of

categories this is intractable, since they grow exponentially as we increase n.

We address the unreliability of general BSLs by imposing structure on $\mathbf{W}$ such that all candidates are argmaxable by construction. We show how we can do this for MLC and MCC under realistic assumptions about which outputs are candidates for our application of interest.

For MLC (Section 5.1), we assume that the label assignments are sparse, an assumption which holds for many MLC datasets (Section 5.1.1). We explain the criteria for $\mathbf{W}$ and introduce the DFT layer which guarantees that all sparse label assignments are argmaxable (Section 5.1.2). We train models on three MLC datasets (Section 5.1.3), and show that contrary to general $\text{BSLs}_{\square}$ which have unargmaxable test set examples when the bottleneck is narrow, our DFT layers do not. In addition, our DFT layer obtains better or comparable F1-score performance while training faster and using up to 50% fewer trainable parameters.

For MCC (Section 5.2), we assume that the rankings of categories are subsumed by the top-k rankings. We then show how to guarantee that all top-k rankings are argmaxable, building off our result for MLC.

As an additional contribution, we close this chapter by discussing a rule of thumb for choosing the dimensionality of a BSL.

5.1 Multi-label Classification

As alluded to above, we now show how to construct a $\text{BSL}_{\square}$ which guarantees that all needed label assignments are argmaxable by construction. To achieve this, we rely on the following sparsity assumption.

5.1.1 The Sparsity Assumption (k-active Label Assignments)

In order to impose our guarantees, we assume label assignments are *sparse*; i.e. we assume that there is an upper bound, k, on the number of labels that are active (i.e. assigned to each input).

The sparsity assumption is natural for MLC. It is common for label assignments in real-world MLC datasets to be sparse (Babbar et al., 2019); it is unlikely an image will contain more than k objects or that a clinical document will be assigned more than k clinical codes, where $k \approx \mathcal{O}(\log n)$ is a dataset dependent upper bound on the number of active labels (Jain et al., 2019). Moreover, even when there are many competing

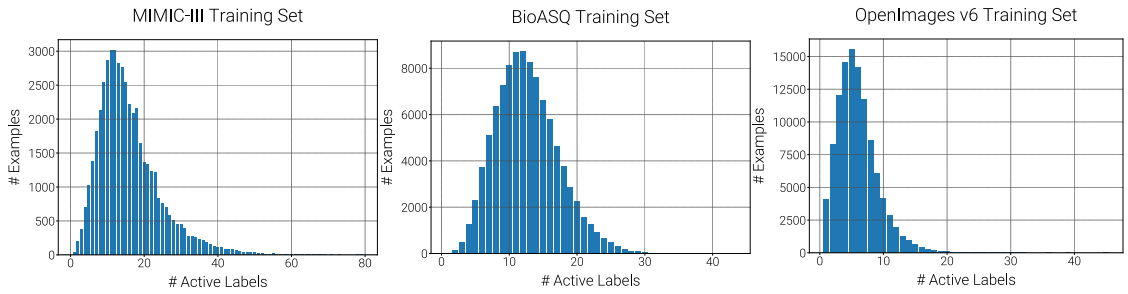


Figure 5.1: For MLC datasets, the majority of label assignments are sparse. We plot the histogram of the number of active labels on the training sets of MIMIC-III, BioASQ and OpenImages MLC datasets, left to right. On the x-axis we plot the number of active labels, k, and on the y-axis we have the number of label assignments in the data that have k active labels. As can be seen, most label assignments are sparse: if we limited the dataset to $k \leq 20$, we would retain most of the dataset. However, all three datasets have a long tail of less sparse label assignments (large k). MIMIC-III has the longest tail, i.e. $k \leq 80$.

labels for a given input, we are motivated to keep the label assignments sparse: we use labels to summarise the input, and we also want to keep annotation manageable. Tagging an input with hundreds of labels from a label vocabulary of thousands would be a daunting task.

As a result, for most MLC tasks, the number of active labels k is bounded (Jain et al., 2019), either empirically (e.g. k=80 for MIMIC-III) or by construction (e.g. k=50 for BioASQ (Tsatsaronis et al., 2015)). In Fig. 5.1 we illustrate a histogram of the number of active labels for the three datasets we introduced in Section 4.1.2. As can be seen, the majority of label assignments are sparse, but there is a long tail of examples that can have quite a large number of active labels (e.g. $k = 80$ for MIMIC-III).

The sparsity assumption restricts the candidate outputs for our MLC task. This is vital, since it enables our BSL to guarantee that all candidate outputs are argmaxable. In our experiments, we choose which k to enforce our guarantees for based on the statistics of the dataset.

5.1.2 Guaranteeing Argmaxable k-active Label Assignments

Before we introduce our guarantees we introduce the argmaxable label assignments.

Argmaxable Label Assignments

As we saw in Section 2.4.4, a BSL $\mathbf{W} \in \mathbb{R}^{n \times d}$ and $d < n$ must have unargmaxable label assignments. The question is, *out of the* 2^n *label assignments,* $\mathbf{y}$*, how many can a BSL*

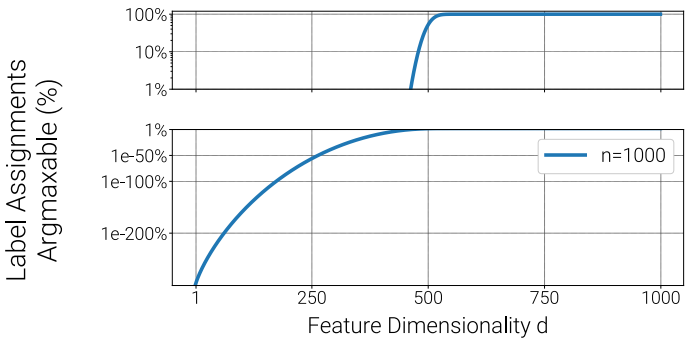


Figure 5.2: We log-plot which percentage of the 2^{1000} label combinations is argmaxable for a BSL with $n = 1000$ labels as we decrease the feature dimensionality d (right to left). When $d \ll n$ we can represent exponentially fewer outputs. We split the y-axis and use two different log scales to highlight how small the percentage becomes as d gets smaller.

actually represent? We define the set of argmaxable label assignments for a classifier $\mathbf{W}$ as:

$$\mathcal{A}(\mathbf{W}) = \{\text{sign}(\mathbf{W}\mathbf{x}) \quad \forall \mathbf{x} \in \mathbb{R}^d\} \tag{5.1}$$

As we saw in Fig. 3.11, the set above is the set of orthants intersected by the span of $\mathbf{W}$. If $\mathbf{W}$ is in **general position** (Definition 2.2.11), we can exactly count the number of argmaxable label assignments, $|\mathcal{A}(\mathbf{W})|$, as we saw in Section 2.4.4.

Number of Argmaxable Label Assignments

Thm. 5.1. (Cover, 1965, Thm 2) If $\mathbf{W}$ is in general position, the number of argmaxable label assignments is:

$$|\mathcal{A}(\mathbf{W})| = 2\sum_{d'=0}^{d-1} \binom{n-1}{d'}. \tag{5.2}$$

It follows that i) the number of argmaxable label assignments depends only on n and d, not the specific $\mathbf{W}$, and ii) most label assignments will be unargmaxable for $d \ll n$ as Eq. (5.2) indicates an exponential growth (see Fig. 5.2). The above argument is similar to that made when defining the VC-Dimension (V. N. Vapnik et al., 1971) of a linear classifier. However, VC-Dimension, like our reasoning above, is agnostic to the precise $\mathbf{W}$. Conversely, we now focus on a specific classifier $\mathbf{W}$ and ask: can we modify $\mathbf{W}$ to guarantee that a specific set of label assignments is argmaxable?

DFT Layers for k-Active MLC

Designing a BSL with $d < n$ that guarantees argmaxability for all 2^n possible label assignments is impossible, according to Theorem 5.1. However, as we saw in Section 5.1.1, for most MLC datasets the label assignments are *sparse*; only a handful of labels are *active* for any given example. As such, herein we choose an upper bound, k, on the number of active labels for each dataset and show how to modify the parametrisation of a BSL so that any k-active label assignment is guaranteed to be argmaxable. We first define sufficient criteria by specifying a *broad family of parametrisations* for which our result holds: the weight matrix should have at least $2k+1$ input features and all its maximal minors (Definition 2.2.13) should be non-zero and have the same sign. Next, we *specify* an implementation satisfying these criteria, the Discrete Fourier Transform (DFT) layer, which is computationally appealing and is accurate in practice.

k-Active Label Assignments We now show how to guarantee that all k-active outputs are argmaxable by imposing structure on the parametrisation, $\mathbf{W}$, of a BSL. We first formalise what a k-active label combination is below.

Number of Active Labels

Def. 5.1.1. For a label assignment $\mathbf{y}$ on n labels we define $\operatorname{act}(\mathbf{y})$ to be the number of active labels in $\mathbf{y}$, i.e:

$$\operatorname{act}(\mathbf{y}) = \sum_{i=1}^{n} [\mathbf{y}_i = +] \tag{5.3}$$

where $[]$ is the Iverson Bracket (see Section 0.2). As an example, we have $\operatorname{act}(-----) = 0$ and $\operatorname{act}(+--+-) = 2$.

k-Active Label Assignments

Def. 5.1.2. The **k-active assignments** on n labels are:

$$A_{n,k} = \{\mathbf{y} \in \{+, -\}^n : \operatorname{act}(\mathbf{y}) \leq k\} \tag{5.4}$$

For example, the MIMIC-III dataset (Johnson et al., 2016; Mullenbach et al., 2018) has $n = 8921$ labels, but no example has more than 80 active labels. We now show how to guarantee that all label assignments in $A_{n,k}$ are argmaxable.

k-Active Argmaxability Guarantees Our goal in this section is to prove Theorem 5.4. It states that a *general criterion* for guaranteeing all k-active labels are

argmaxable is that the weight matrix $\mathbf{W} \in \mathbb{R}^{n \times d}$, $d \geq 2k+1$ that parametrises the BSL has *maximal minors that agree in sign and are non-zero.* As such, we next introduce maximal minors and the family of matrices with the above property. We prove our result by showing that the argmaxable label assignments for this family of matrices are "$2k$-alternating" and that these subsume the k-active ones.

A **maximal minor** Δ_I of a $n \times d$ matrix, $n > d$, is the determinant of any $d \times d$ submatrix formed by deleting the $n-d$ rows not indexed by I (see Section 0.2 for more details on notation). For example, in Fig. 5.3(a), all maximal minors are positive. We say that a matrix $\mathbf{W} \in \mathbb{R}^{n \times d}$ is a representative of an element of $\mathsf{Gr}^{+}_{n,d}$ if its maximal minors are non-zero and agree in sign (see Definition 2.2.15). To prove Theorem 5.4, we use the following facts known about k-alternating outputs.

Number of Sign Changes

Def. 5.1.3. For a label assignment $\mathbf{y}$ on n labels we define $\mathrm{alt}(\mathbf{y})$ to be the number of sign changes encountered when reading the sequence of labels from left to right, i.e:

$$\mathrm{alt}(\mathbf{y}) = \sum_{i=1}^{n-1} [\mathbf{y}_i \neq \mathbf{y}_{i+1}] \tag{5.5}$$

Note that we do not compare the signs of the first and last label. As an example, we have $\mathrm{alt}(\,{+}{+}{-}{-}{-}\,) = 1$ and $\mathrm{alt}(\,{+}{+}{-}{+}{-}\,) = 3$.

k-Alternating Label Assignments

Def. 5.1.4. The **k-alternating assignments** on n labels are:

$$V_{n,k} = \{\mathbf{y} \in \{\,+\,,\,-\,\}^n : \mathrm{alt}(\mathbf{y}) \leq k\} \tag{5.6}$$

k-Active Implies $2k$-Alternating

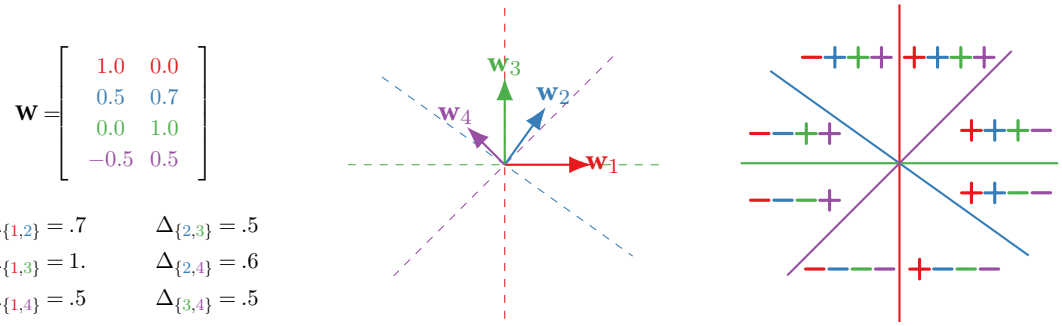
Lemma 5.1.

$$\mathbf{y} \in A_{n,k} \implies \mathbf{y} \in V_{n,2k} \tag{5.7}$$

Proof. We can construct any k-active $\mathbf{y}$ of length n from the all inactive $\mathbf{y}$ by flipping all signs after any of the $n-1$ positions between labels. We need at most $2k$ distinct flips. E.g., we can produce ${-}{+}{-}{-}$ with 2 flips: ${-}{-}{-}{-} \to {-}{+}{+}{+} \to {-}{+}{-}{-}$. □

We are almost ready to obtain our criteria for the argmaxability of $A_{n,k}$. However, we need to arrive to them via theorems about argmaxability of $V_{n,2k}$. We will combine

a counting argument on the number of regions from Theorem 5.1, Lemma 5.1 and the following result on alternating $\mathbf{y}$ from Gantmakher et al. (1961) via Karp (2017).



(a) All $\binom{4}{2} = 6$ maximal minors, Δ_I, are positive, hence $\text{span}(\mathbf{W}) \in \text{Gr}^+_{4,2}$.

(b) Correspondence of $\mathbf{W}$ and geometric picture: each normal vector is a row of $\mathbf{W}$.

(c) The Argmaxable label assignments are the 1-alternating vectors, $\mathcal{A}(\mathbf{W}) = V_{4,1}$.

Figure 5.3: Visual evidence of Theorem 5.3. a) We construct a BSL having $n = 4$ labels and $d = 2$ features parametrised by $\mathbf{W} \in \mathbb{R}^{4\times 2}$ such that all maximal minors are positive, i.e. $\text{span}(\mathbf{W}) \in \text{Gr}^+_{n=4,d=2}$. (b) The rows of the matrix are binary classifiers, we demarcate the decision boundaries for each classifier using a dashed line. (c) We assign each region a sign vector corresponding to which labels the BSL would flag as active for an input falling in that region. As per Theorem 5.3, exactly the $(d-1) = 1$-alternating outputs are argmaxable. More generally, for $d = 2k+1$, all k-active outputs are argmaxable (see Fig. 5.4).

Maximal Minors and Alternating Label Assignments

Thm. 5.2. (Gantmakher et al., 1961) see (Karp, 2017, Theorem 1.1). If all maximal minors of $\mathbf{W} \in \mathbb{R}^{n\times d}$ are non-zero and have the same sign, all label assignments $\mathbf{y}$ computed as $\mathbf{y} = \text{sign}(\mathbf{W}\mathbf{x})$, $\mathbf{x} \in \mathbb{R}^d$ are $d-1$ alternating.

$$\text{span}(\mathbf{W}) \in \text{Gr}^+_{n,d} \implies \text{alt}(\mathbf{y}) \leq d-1 \tag{5.8}$$

But are there any label assignments having $\text{alt}(\mathbf{y}) \leq d-1$ that are unargmaxable? Via a counting argument we can show that this is not the case.

Argmaxable k-Alternating Label Assignments Guarantees

Thm. 5.3. For $\mathbf{W}$: $\text{span}(\mathbf{W}) \in \text{Gr}^+_{n,d}$ the argmaxable label assignments are the $(d-1)$-alternating vectors.

$$\text{span}(\mathbf{W}) \in \text{Gr}^+_{n,d} \implies \mathcal{A}(\mathbf{W}) = V_{n,d-1} \tag{5.9}$$

See Appendix A.8.1 for the proof and Fig. 5.3 for a visual confirmation that

$\text{span}(\mathbf{W}) \in \mathsf{Gr}^{+}_{4,2}$ makes the 1-alternating $\mathbf{y}$ argmaxable. We can now state our main result.

Argmaxable k-Active Label Assignments Guarantees

Thm. 5.4. Consider a BSL parametrised by $\mathbf{W} : \text{span}(\mathbf{W}) \in \mathsf{Gr}^{+}_{n,2k+1}$ which predicts label assignments using argmax prediction, $\mathbf{y} = \text{sign}(\mathbf{W}\mathbf{x})$. All k-active label assignments are argmaxable: $A_{n,k} \subset \mathcal{A}(\mathbf{W})$.

Proof. From Theorem 5.3, for $\mathbf{W}$ such that $\text{span}(\mathbf{W}) \in \mathsf{Gr}^{+}_{n,2k+1}$ the set of $2k$-alternating label assignments $V_{n,2k}$ is argmaxable. Then, from Lemma 5.1, we have $A_{n,k} \subseteq V_{n,2k} = \mathcal{A}(\mathbf{W})$, and therefore all k-active labels are argmaxable. □

Observation: The Order of the Label Vocabulary Matters We note that for our results above, we have assumed that the vocabulary of labels is ordered in a particular way, i.e. the order of the rows of $\mathbf{W}$ matters. This is because the number of sign changes in a sign vector clearly depends on the order of the rows. That being said, the k-active labels would still be argmaxable if we permuted the rows of any $\mathbf{W}$ that meets our criteria. This is so because permutations of the rows cannot change the number of active labels.

Rabbit Hole: Vocabulary Order

Can we choose a better ordering of the vocabulary based on properties of $\mathbf{W}$ and the dataset of labels at hand? We highlight that this question is especially interesting for ϵ-argmaxability: the DFT layers we will introduce next can only model low-frequency signals, so they cannot model sudden changes between neighbouring labels. Because of this limitation, it would make sense to order the vocabulary such that labels that are frequently active or inactive together are neighbouring. Idea: Encode label co-occurrence information on a graph that has the labels as nodes and edge weights that depend on some metric of label co-occurrence. Obtain an order of the vocabulary (i.e. a Hamiltonian on the graph) by solving a Travelling Salesman Problem on the graph.

In summary, we have showed that if $\text{span}(\mathbf{W}) \in \mathsf{Gr}^{+}_{n,2k+1}$, all k-active outputs are argmaxable, but we have not given a concrete implementation. We next introduce the DFT layer, a practical and computationally appealing member of the $\mathsf{Gr}^{+}_{n,2k+1}$ family.

DFT Layers We now engineer a BSL which satisfies Theorem 5.4. We satisfy the criteria by parametrising the BSL with the Discrete Fourier Transform (DFT) matrix. While any parametrisation of a BSL that satisfies the criteria of the previous section is suitable, the DFT is especially appealing because:

1. we can reduce the number of learnable parameters as the DFT matrix is fixed. [1]

2. we can compute the activation of the DFT Layer in $\mathcal{O}(n \log n)$ time via the Fast Fourier Transform (see Appendix A.10) instead of a more expensive $\mathcal{O}(nd)$ generic low-rank matrix-vector product.

We next describe a truncated DFT matrix and show that it provides the k-active guarantees we seek. For k-active guarantees, we need $d = 2k+1$, so we truncate frequencies larger than k to obtain $\mathbf{W}^{\mathsf{DFT}}_{n,2k+1}$, as shown below:

$$\mathbf{W}^{\mathsf{DFT}}_{n,2k+1} = \begin{bmatrix} \frac{1}{\sqrt{n}} & \sqrt{\frac{2}{n}}\cos t_1 & \sqrt{\frac{2}{n}}\sin t_1 & \cdots & \sqrt{\frac{2}{n}}\cos kt_1 & \sqrt{\frac{2}{n}}\sin kt_1 \\ \frac{1}{\sqrt{n}} & \sqrt{\frac{2}{n}}\cos t_2 & \sqrt{\frac{2}{n}}\sin t_2 & \cdots & \sqrt{\frac{2}{n}}\cos kt_2 & \sqrt{\frac{2}{n}}\sin kt_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{1}{\sqrt{n}} & \sqrt{\frac{2}{n}}\cos t_n & \sqrt{\frac{2}{n}}\sin t_n & \cdots & \sqrt{\frac{2}{n}}\cos kt_n & \sqrt{\frac{2}{n}}\sin kt_n \end{bmatrix},$$
$$t_i = \frac{2\pi(i-1)}{n}, i \in [n] \tag{5.10}$$

Parsing the DFT Matrix We can parse the matrix from Eq. (5.10) more easily if we break it down column-wise. The first column on the left is constant and is the offset term of the DFT, sometimes called the DC term in signal processing. The next columns come in pairs of sines and cosines of a given frequency k, where the frequency increases from left to right, i.e. column $2k+1$ has frequency k. The constant multipliers $\frac{1}{\sqrt{n}}$ in front of the DC column and $\sqrt{\frac{2}{n}}$ in front of the sines and cosines ensure that the columns are orthonormal.

Pytorch Implementation Implementation-wise, we can efficiently compute the output of the DFT layer in pytorch by using the `ifft` operation. To do so, we use the `norm='ortho'` option. However, this operation scales all columns by $\frac{1}{\sqrt{n}}$. In our implementation, we therefore scale the inputs $x_i, i = 2, \ldots, 2k+1$ by $\sqrt{2}$. Since the `ifft` operation is defined on complex numbers, we also need to encode $\mathbf{x}$ as a vector of complex numbers. See our code and Appendix A.10 for more details.

[1] In early experiments learning the t_i of the DFT matrix (Eq. (5.10)) had little impact on the results.

Figure 5.4: Decision boundaries for DFT output layer with $d = 3$ and $n = 3, 4, 5$, from left to right. As a result of Theorem 5.4, all 1-active label assignments are argmaxable. The large central region corresponds to $\mathbf{y}$ with no active labels and the n regions surrounding it have a single active label, as we illustrate on $\square_3$ on the left.

The Truncated DFT Matrices Satisfy Our Criteria Now that we have introduced the form of the DFT matrix and explained how to compute its outputs, we show that it satisfies our criteria.

Truncated DFT Matrices Satisfy the Criteria

Proposition 5.1. *A truncated DFT matrix satisfies* $\operatorname{span}\left(\mathbf{W}^{\mathsf{DFT}}_{n,2k+1}\right) \in \mathsf{Gr}^{+}_{n,2k+1}$.

We need to show that the maximal minors of $\mathbf{W}^{\mathsf{DFT}}_{n,2k+1}$ are non-zero and agree in sign. See Appendix A.8.2 for a proof. As a visual confirmation of the result for $k = 1$, we illustrate the decision boundaries of $\mathbf{W}^{\mathsf{DFT}}_{n,3}$ in Fig. 5.4. As expected, you can find a region corresponding to any 1-active label assignment.

Problem: Regions can become too small. While in practice we could use the fixed DFT Layer as defined above and rely on an expressive feature encoder to do the heavy lifting, if the number of labels, n, is much greater than the number of features, $2k + 1$, it becomes hard to classify some label assignments with large confidence. This is due to the hyperplane overcrowding problem (Section 4.1.1): segmenting a low dimensional space with very many hyperplanes induces regions that become arbitrarily small shards. In argmaxability terms, if we fix an ϵ and increase the number of labels, all k-active labels remain argmaxable but increasingly more are not ϵ-argmaxable, see left side of Fig. 5.5. This is a problem, since for successful training and generalisation, we need our encoder to successfully map points to the regions that correspond to the target $\mathbf{y}$, and this cannot happen if the regions are too small.

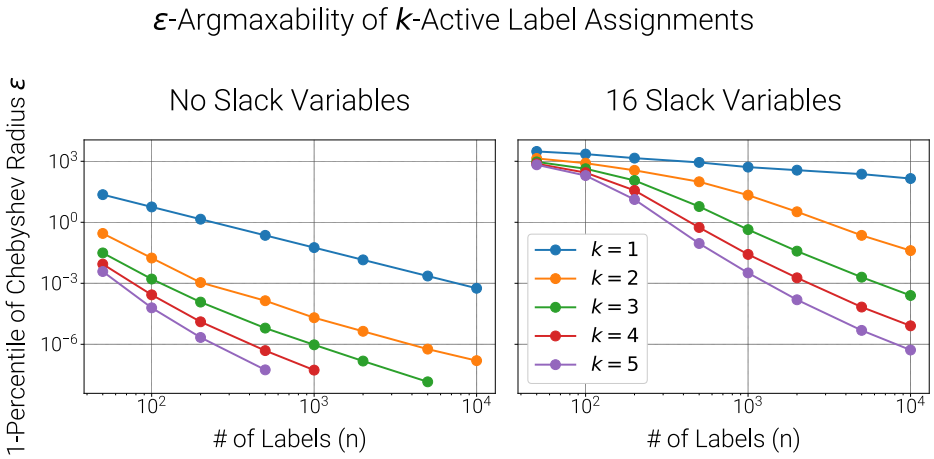


Figure 5.5: Slack variables can alleviate hyperplane overcrowding. Left: As we increase n, the number of labels (and hence hyperplanes) for the DFT Layer, the radii of the regions shrink due to hyperplane overcrowding, making them harder to predict in practice. Right: Adding slack variables ameliorates this problem. We plot ϵ-argmaxability (Definition 4.1.1), measured here for the 1% of labels that have radius less than that plotted. For the DFT Layer, i.e. $\mathbf{W} = \mathbf{W}^{\text{DFT}}_{n,2k+1}$, all k-active label assignments are argmaxable, but as we increase n, some (see $k \geq 3$) cannot be detected at the precision of the LP (10^{-8}). Adding 16 randomly initialised slack columns, i.e. $\mathbf{W} = \left[\mathbf{W}^{\text{DFT}}_{n,2k+1}\ \mathbf{S}\right]$, makes the regions ϵ-argmaxable with larger ϵ.

While hyperplane overcrowding is a problem for any classifier $\mathbf{W}$, we found that DFT layers are more susceptible to it than general BSLs (see Appendix A.9 for a more detailed explanation).

Solution: Slack variables. A practical way to deal with small regions is to increase the dimensionality of the features by adding learnable *slack variables*, see Fig. 5.5. Crucially, when doing so we retain our argmaxability guarantees, as we show below.

Adding Slack Variables Maintains Argmaxability

Proposition 5.2. *Assume a label assignment* $\mathbf{y}$ *is argmaxable for a classifier* $\mathbf{W} \in \mathbb{R}^{n \times d}$. *Consider increasing the dimensionality of the features of the classifier* $\mathbf{W}$ *by adding* s *more randomly initialised slack columns* $\mathbf{S} \in \mathbb{R}^{n \times s}$. *Then* $\mathbf{y}$ *is also argmaxable in* $\mathbf{W}' = \left[\mathbf{W}\ \mathbf{S}\right]$, $\mathbf{W}' \in \mathbb{R}^{n \times (d+s)}$.

Proof. Consider the input feature vector for $\mathbf{W}'$, $\mathbf{x}' = \begin{bmatrix} \mathbf{x} \\ \mathbf{x}_s \end{bmatrix}$, $\mathbf{x} \in \mathbb{R}^d$, $\mathbf{x}_s \in \mathbb{R}^s$. Set

$\mathbf{x}_s = \mathbf{0}$. Then notice that $\mathbf{y} = \text{sign}\left(\begin{bmatrix}\mathbf{W}\ \mathbf{S}\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ \mathbf{0}\end{bmatrix}\right) = \text{sign}\left(\mathbf{W}\mathbf{x}\right)$ is equivalent to the original classifier, so if $\mathbf{y}$ is argmaxable in $\mathbf{W}$ it is also argmaxable in $\mathbf{W}'$ by setting $\mathbf{x}_s$ to zero. □

As such, we propose the DFT layer, with $(2k+1) \times n$ fixed parameters which enforce argmaxability and $s \times n$ learnable parameters which give it flexibility.

5.1.3 Experiments

We now empirically evaluate the DFT layer and the BSL layer we introduced in Section 4.1 on three MLC datasets. In Section 4.1.1 we saw that a narrow BSL does indeed have unargmaxable test set label assignments. Here we answer the following research questions: **RQ1)** Can DFT layers guarantee that the k-active label assignments are argmaxable in practice? **RQ2)** What is the trade-off between performance and the number of trainable parameters? **RQ3)** Do we get any speed up in training time due to the DFT? To answer these questions, we use the datasets we introduced in Section 4.1.2. To answer our research questions above, we introduce the setup of the DFT layer.

DFT Output Layer

We compare the two MLC output layers we use, the general *BSL Layer* which we introduced in Section 4.1.2 that is unconstrained and does not guarantee argmaxability of k-active outputs, and our *DFT layer* which does by construction.

While both classifiers $\mathbf{W}$ have the same number of learnable parameters for both output layers, their parametrisations differ.

For the DFT, we first pick the maximum number of active labels, k, depending on the statistics of the dataset. We then set the number of slack dimensions to be d so we can directly compare to the BSL. As such, we also need to adapt the dimensions of the linear projection layer, $\mathbf{P}$, which maps the embeddings, e, of the feature encoder, to the features of the classifier, as per Eq. (4.6). We therefore set $\mathbf{P} \in \mathbb{R}^{e \times (2k+1+d)}$ and $\mathbf{b} \in \mathbb{R}^{(2k+1+d)}$, since we include $2k+1$ more features to the classifier. The learnable parameters of the classifier comprise d slack columns.

Conceptually, and for the purposes of checking the classifier with our LP, we construct the classifier by concatenating the fixed DFT matrix to the slack columns, i.e. $\mathbf{W} = \left[\mathbf{W}^{\text{DFT}}_{n,2k+1}\ \mathbf{S}\right]$, $\mathbf{W} \in \mathbb{R}^{n \times (2k+1+d)}$. In practice, however, we compute the logits

$\mathbf{z}$ efficiently using the Fast Fourier Transform: $\mathbf{z} = \text{FFT}(\mathbf{x}_{:2k+1}) + \mathbf{S}\mathbf{x}_{2k+2:}$ (see Appendix A.10).

Computational Cost of DFT Compared to the BSL, the cost of the DFT layer with n labels is:

1. an additional cheap $\mathcal{O}(n \log n)$ matrix vector multiplication
2. an additional $e \times (2k+1)$ trainable parameters in the projection layer

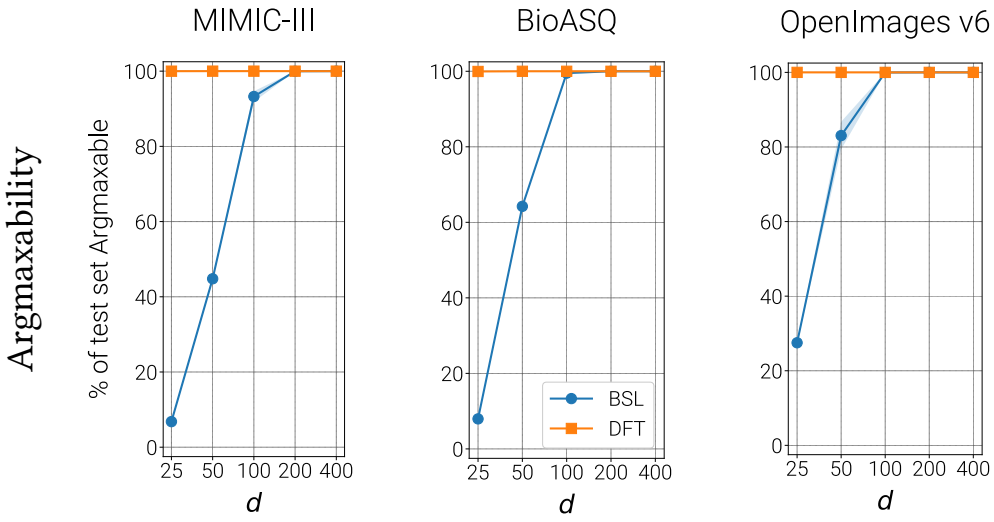
However, for models with large n, we can easily offset the increase in parameters if we want, by modestly shrinking the slack d of the DFT output layer. For example, the MIMIC-III CNN models (Mullenbach et al., 2018) have $n = 8921$ and $e = 500$. For $k = 80$, a DFT adds 500×161 parameters to the projection layer. We could offset this by decreasing d in the output layer by only $\lceil \frac{500 \times 161}{8921} \rceil = 10$. In the results section we show that DFT layers obtain better performance with lower d than BSLs, and as such can be more parameter efficient.

Faster Training of DFT For the DFT, we introduce an initialisation trick to speed up training. We exploit that a) $\mathbf{W}^{\text{DFT}}$ is known and fixed and b) the outputs are k-active. Since the outputs are k-active, we would prefer to assign a probability $\frac{k}{n}$ to all labels when we start training. To achieve this, we can exploit the fact that the first column of $\mathbf{W}^{\text{DFT}}$ is $\frac{1}{\sqrt{n}}$ and initialise the bias vector of the projection layer to be $[\sqrt{n}\,\text{logit}(\frac{k}{n}), 0, \ldots, 0]$, where the logit function is the inverse of sigmoid. This way, assuming logits are close to zero when we begin training, the model will assign probability $\approx \frac{k}{n}$ to each label instead of $\approx \frac{1}{2}$. [2] A similar bias initialisation idea for MLC was discussed in (Schultheis et al., 2022), but it was not used in a neural network.

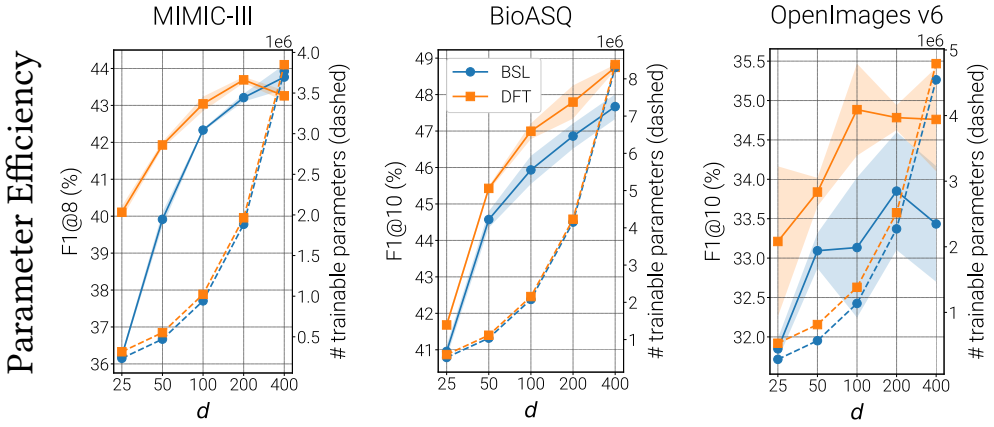
Results

RQ1) DFT: Argmaxability. We showed that the DFT layer guarantees argmaxability in theory. As can be seen in Fig. 5.6(a), it also works in practice. Crucially, these guarantees also apply to unseen k-active label assignments; this would be impossible to enforce with BSLs, as discussed.

[2] We note that we may obtain better initialisation results by including a bias term that captures the marginal probability of each individual label. However, our point is that by knowing that $\mathbf{W}$ contains a constant column vector, we can initialise the output layer sensibly even without adding a bias term. This is not possible without the addition of a bias term if $\mathbf{W}$ is randomly initialised.



(a) Our DFT layer does not suffer from unargmaxable test examples. BSLs cannot provide any such guarantees.



(b) Our DFT layer is more parameter efficient. F1 is overall better than the BSL, so we can match BSL's F1 with fewer trainable parameters (smaller d).

Figure 5.6: Comparison of BSL and our DFT output layer on three MLC datasets. We vary, d, the number of trainable dimensions and plot the mean and std (shaded) over 3 runs with different random seeds. Left: We use the LP to verify the output layers on the test sets. When $d < 200$ a large percentage of label configurations becomes unargmaxable for the BSL, in contrast to our DFT. Right: Performance in F1@8 or F1@10 (left axis) in terms of the number of trainable parameters (right axis, dashed lines). Our DFT is better ($d \leq 200$) or comparable to the BSL (MIMIC-III, $d > 200$). We can therefore retain a high F1@k score for DFT even if we reduce the number of trainable parameters by making the bottleneck narrower. For example, on BioASQ we can surpass the F1@10 of the BSL that has $d = 200$ with a DFT of $d = 100$, which has about 50% fewer trainable parameters.

We note, however, that there is a limit to how far we can shrink the dimensionality in practice. When we shrink the bottleneck to $d = 25$, a handful of outputs become eps-unargmaxable (Section 4.1.2) even for the DFT layer. We illustrate these outputs for the BioASQ task in Table 5.2. Out of these, one example was found to be infeasible, while for the remaining ones numerical difficulties were encountered by the LP, i.e. Gurobi returned status 12: "*Operation terminated due to unrecoverable numerical difficulties*". The above issues are likely caused because of dimensionality pressures, the label assignments are ϵ-argmaxable but with a very small ϵ that cannot be detected with the precision of current LPs (we use eps=10^{-8}, as we discussed in Section 4.1.2). This highlights the importance of our proofs, since our results would be tricky to verify using empirical methods alone. Guaranteeing ϵ-argmaxability with a large ϵ is important future work, since while we showed that our current solution of adding slack variables works in practice, if we increase the pressure on the bottleneck by making d small enough, we can still run into ϵ-argmaxability issues.

Lastly, we note that the behaviour of BSLs in terms of eps-argmaxability and ϵ-argmaxability is quite different when compared to DFT. This can be seen for all 3 datasets in Tables 5.1 and 5.2. For BSLs, if a label combination is argmaxable, it is very often also ϵ-argmaxable. On the other hand, for the DFT some label assignments are argmaxable but are not ϵ-argmaxable, highlighting that the regions do indeed exist, but they can shrink quite a bit in size due to the reduced dimensionality.

RQ2) Parameter Efficiency. In addition to the argmaxability guarantees, the DFT layer outperforms the BSL layer by a wide margin for small d, as we illustrate in Fig. 5.6(b). This allows us to match the performance of the BSL using a DFT with smaller d and hence fewer trainable parameters. As can be seen, in some cases DFT layers obtain better or comparable performance with up to 50% fewer trainable parameters.

RQ3) Faster Training. Lastly, a benefit of DFT layers is that they converge faster than BSLs due to the initialisation trick (Section 5.1.3). We focus on the most demanding datasets, BioASQ and OpenImages v6. In Fig. 5.7, we show how the training loss evolves over time. Meanwhile, in Fig. 5.8 we compare the number of hours it took for the BSL and DFT models to converge on BioASQ and OpenImages v6. Both figures show that the DFT layer leads to faster convergence, as it starts training with a lower loss due to the initialisation trick and maintains its lead over the BSL throughout training.

split	d	MIMIC-III			
		# *eps*-Argmaxable		1-Argmaxable	
		BSL	DFT	BSL	DFT
dev	25	128	1631	128	1572
	50	781	1631	781	1625
	100	1533	1631	1533	1631
	200	1631	1631	1631	1631
	500	1631	1631	1631	1631
test	25	229	3371	229	3216
	50	1515	3371	1515	3356
	100	3137	3371	3137	3370
	200	3370	3371	3370	3371
	500	3371	3371	3371	3371

Table 5.1: Median number of *eps*-argmaxable and 1-argmaxable label assignments over 3 random seeds on the dev and test sets of MIMIC-III. Takeaway: BSL layers have unargmaxable labels for $d < 200$ but it does not have to be this way. DFT layers resolve this problem and make all examples argmaxable. Nevertheless, when slack dimensionality is very small, the regions can be too small to detect with the precision of the LP.

d	BioASQ				OpenImages v6			
	# *eps*-Argmaxable		# 1-Argmaxable		# *eps*-Argmaxable		# 1-Argmaxable	
	BSL	DFT	BSL	DFT	BSL	DFT	BSL	DFT
25	880	9995	879	8174	2758	10000	2757	9981
50	6498	10000	6483	9925	8439	10000	8435	10000
100	9951	10000	9950	10000	9997	10000	9997	10000
200	10000	10000	10000	10000	10000	10000	10000	10000
400	10000	10000	10000	10000	10000	10000	10000	10000

Table 5.2: Median number of *eps*-argmaxable and 1-argmaxable label assignments over 3 random seeds on the test set for the BioASQ and OpenImages v6 datasets.

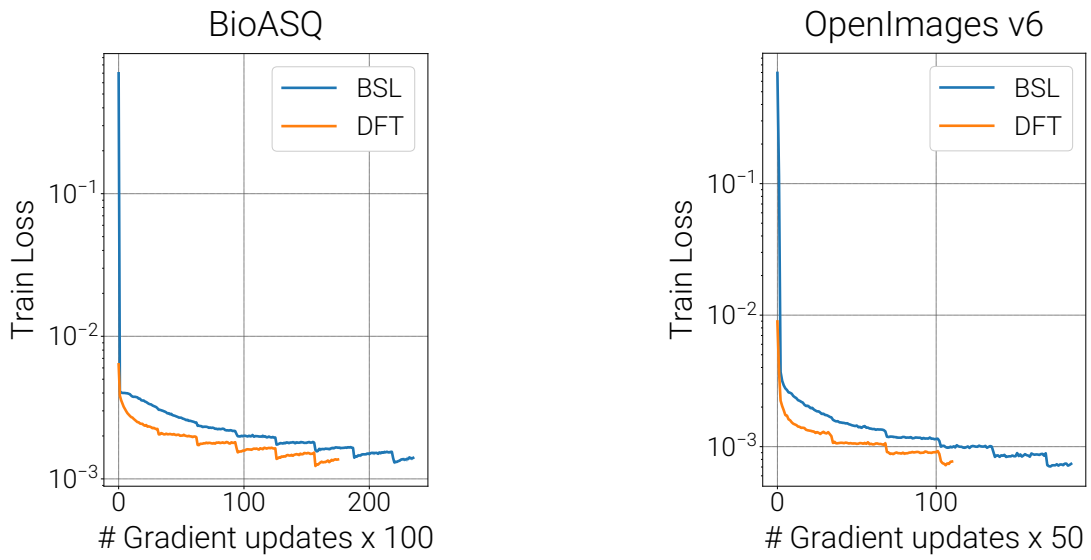


Figure 5.7: Comparison of BSL and DFT for $d = 100$, in terms of the training cross entropy loss (y-axis, log scale) as training evolves (x-axis). Due to the initialisation trick, the DFT starts training at a lower loss and converges faster.

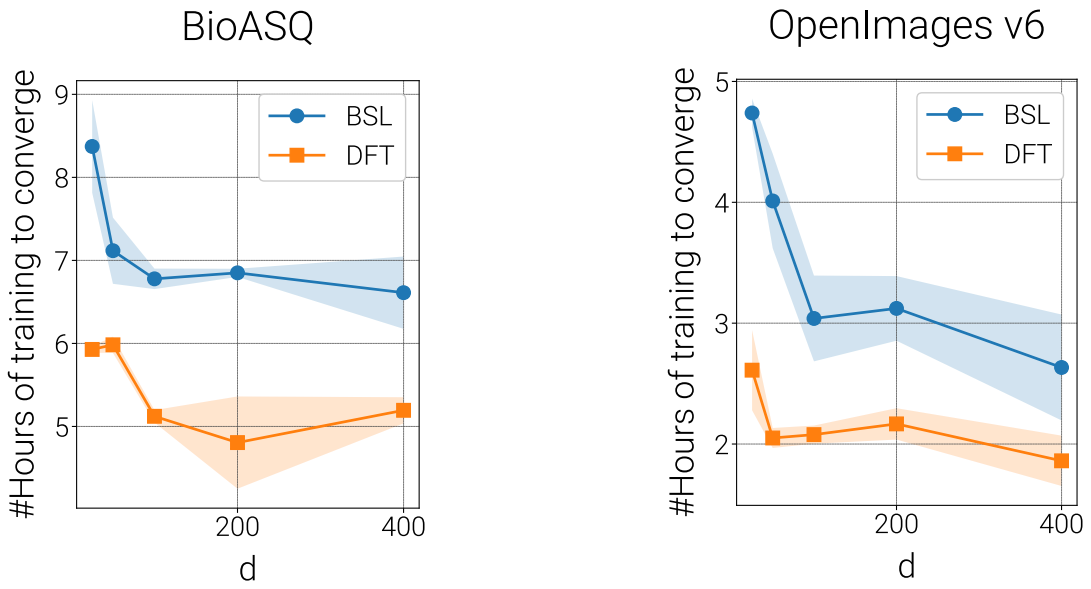


Figure 5.8: Comparison of BSL and DFT in terms of training time (in hours) to convergence. As can be seen, the DFT converges about 25% faster.

5.2 Multi-class Classification

We now turn to $\text{BSLs}_{\triangle}$. As we saw in Section 3.5, for the MCC case there must be unargmaxable rankings. Similarly to the MLC case, we discuss how to choose a family of $\mathbf{W}$ such that any top-k ranking of categories is argmaxable.

We begin with categories, which we can interpret as top-1 rankings, and then discuss the more general case of top-k rankings, which subsume the former.

5.2.1 Guaranteeing Argmaxable Categories

Guaranteeing that all categories are argmaxable is relatively straightforward, as mentioned in (Demeter et al., 2020). All categories are argmaxable if all rows of $\mathbf{W}$ are vertices of a polytope. To guarantee that all rows of $\mathbf{W} \in \mathbb{R}^{n \times d}$ are on the convex hull, we can inscribe them on a d-sphere. We achieve this by normalising all row vectors by dividing them by their euclidean norm. Once the rows are on the unit sphere, we can also scale the radius of the sphere by multiplying $\mathbf{W}$ by any positive constant.

However, we note that this straightforward solution cannot describe all polytopes, since there exist polytopes that cannot be inscribed on a sphere (Doolittle et al., 2020; Manecke et al., 2022). Therefore, this proposed solution, while elegantly simple, restricts the expressivity of $\mathbf{W}$ to a subset of polytopes, i.e. the inscribable ones.

We now turn to our result for top-k rankings of categories. Since argmaxable categories are equivalent to top-1 rankings, our result below subsumes the above, while also not restricting the set of polytopes to the inscribable ones.

5.2.2 Guaranteeing Argmaxable Top-k Rankings

We now show how to build a BSL for which all top-k rankings of categories are argmaxable. We build off our results in Section 5.1. Recall Theorem 5.4.

Argmaxable k-Active Label Assignments Guarantees

Thm. 5.4. Consider a BSL parametrised by $\mathbf{W} : \text{span}(\mathbf{W}) \in \text{Gr}^{+}_{n,2k+1}$ which predicts label assignments using argmax prediction, $\mathbf{y} = \text{sign}(\mathbf{W}\mathbf{x})$. All k-active label assignments are argmaxable: $A_{n,k} \subset \mathcal{A}(\mathbf{W})$.

We can adapt our k-active MLC result to rankings of categories in MCC.

Argmaxable Top-k Ranking Guarantees

Proposition 5.3. *Consider a BSL parametrised by* $\mathbf{W}' : \operatorname{span}(\mathbf{W}') \in \mathsf{Gr}^{+}_{n,2k+1}$. *If* $\mathbf{W}' = [\mathbf{W}\ c\mathbf{1}]$, $c \in \mathbb{R}$, *then if we use* $\mathbf{W} \in \mathbb{R}^{n\times 2k}$ *as a softmax classifier, all top-*k *rankings of the categories are argmaxable.*

Proof. If $\mathbf{W}' = [\mathbf{W}\ c\mathbf{1}]$, $c \in \mathbb{R}$, we can write:

$$\operatorname{sign}\left(\mathbf{z}'\right) = \operatorname{sign}\left(\mathbf{W}'\mathbf{x}'\right) \tag{5.11}$$

$$= \operatorname{sign}\left(\begin{bmatrix}\mathbf{W}\ c\mathbf{1}\end{bmatrix}\begin{bmatrix}\mathbf{x}'_{:2k}\\ x'_{2k+1}\end{bmatrix}\right) \tag{5.12}$$

$$= \operatorname{sign}\left(\mathbf{W}\mathbf{x} - t\right) \qquad \text{set } t = -cx'_{2k+1} \tag{5.13}$$

$$= \operatorname{sign}\left(\mathbf{z} - t\right) \qquad \text{set } \mathbf{z} = \mathbf{W}\mathbf{x} \tag{5.14}$$

We interpret the above as follows: $\mathbf{z} = \mathbf{W}\mathbf{x}$ is a vector of scores for each category and $t \in \mathbb{R}$ is a threshold. The model can encode any $t \in \mathbb{R}$ by changing the value of feature x'_{2k+1}. Now, since $\operatorname{span}(\mathbf{W}') \in \mathsf{Gr}^{+}_{n,2k+1}$, from Theorem 5.4 we know that the k-active outputs are argmaxable. This means that $\exists t$ such that we can isolate any k subset of categories from the remaining ones. To do so, these k categories must have had the top-k scores in $\mathbf{z}$; i.e. even if we dropped the constant column from $\mathbf{W}'$, these k categories would have the top-k scores for $\mathbf{W}$. Moreover, since all r-subsets are argmaxable for $r \leq k$, this implies that all top-k rankings of the categories are argmaxable for $\mathbf{W}$. □

An example $\mathbf{W}'$ that fulfills the above criteria is the DFT matrix Eq. (5.10). Therefore, we could set $\mathbf{W}$ to be $\mathbf{W}'$ with the constant column removed, and all top-k rankings would be argmaxable. Recall also that including the constant column would have no effect on the output of softmax, as we showed in Property 3.1.

5.3 Conclusions

Multi-Label Classification

BSLs are intrinsically unreliable: they must have unargmaxable label assignments and can therefore hinder the generalisation capabilities of our DNNs. However, we showed that this need not be the case. We have shown that if our candidate label assignments

are sparse, then we can still build a $\mathrm{BSL}_{\square}$ which guarantees that all candidate outputs are argmaxable. We introduced the DFT layer, which in addition to the guarantees, allows us to match the performance of a general $\mathrm{BSL}_{\square}$ while training faster and using up to 50% fewer trainable parameters.

However, while introducing the guarantees, we ran into the practical problem of hyperplane overcrowding (Section 4.1.1): we need outputs to be argmaxable with a sizeable margin in order for them to be predicted in practice by a DNN. Guaranteeing a lower bound on this margin is an interesting direction for future work.

Multi-Class Classification

In our search for parametrisations of output layers that come with argmaxability guarantees, we found a lower bound on the dimensionality needed to guarantee k-active labels are argmaxable. We then showed how to transfer our results over to MCC. We described a $\mathrm{BSL}_{\triangle}$ that guarantees that all top-k rankings of categories are argmaxable. However, unlike the MLC case, we have not verified our MCC results empirically, we leave this to future work.

A Rule of Thumb for Choosing d

When engineering neural networks, we mostly treat the number of feature dimensions of the last layer of our DNN as a hyperparameter to be chosen empirically by cross-validation. But can we make an informed guess for d given prior knowledge about our dataset? What is a reasonable choice for d?

In this chapter we built a deeper understanding of what happens when we constrain $\mathbf{W}$ to low dimensions. For a BSL $\mathbf{W} \in \mathbb{R}^{n \times d}$, we showed that:

- For MLC, all k-active label assignments can be argmaxable when $d \geq 2k + 1$.
- For MCC, all top-k rankings of categories can be argmaxable when $d \geq 2k$.

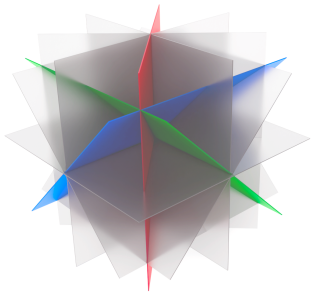
We interpret the above results as a rule of thumb for selecting d: If we have prior knowledge about our data telling us that we need these families of outputs, we should make d at least as large as specified above.

We still need to inform our choice of d by experiments, since due to the problem of hyperplane overcrowding Section 4.1.1, we will likely need to increase d to make all regions argmaxable by a large enough margin. But the insights we have built in

this chapter give us a ballpark. Another direction for future work is to quantify how much we need to increase d to guarantee that all candidate outputs are argmaxable by a sizeable margin.

Conjectures We close this chapter with the following two conjectures. The dimensions we provided our guarantees for are the lowest possible; i.e. there exists no $\mathbf{W} \in \mathbb{R}^{n \times d}$ such that:

1. all k-active label assignments are argmaxable when $d < 2k + 1$.
2. all top-k rankings of categories are argmaxable when $d < 2k$.



6 Conclusions & Future Work

Therefore neural networks are not well-controlled learning machines.
Nevertheless, in many practical applications, neural networks demonstrate good results.

—Vladimir Vapnik, *Statistical learning theory.*

Algorithms like LSTM and RNN may work in practice. But do they work in theory?
In particular, can they learn all the syntactic stuff in the long tail (...) ?

—Mark Steedman, "The Lost Combinator"

In this thesis, we introduced the concept of unargmaxability: an understudied limitation of DNNs that makes some outputs impossible to predict. We made argmaxability precise by introducing terminology (Section 3.4) and understood the phenomenon in depth by connecting it to fundamental concepts in geometry and combinatorics (Chapter 2). We used our understanding to highlight that although recent models, such as LLMs, are extremely expressive, they are restricted by their output layer, which is a BSL (Section 3.3).

In Chapter 4, we showed that BSLs are intrinsically unreliable and can hinder the generalisation capabilities of our otherwise extremely expressive DNNs. To show this, we introduced tools to detect unargmaxable outputs in MLC and MCC models. We used our tools to show that BSLs have unargmaxable outputs in practice, especially when we make the bottleneck narrow by reducing the number of features. While

we showed that unargmaxable outputs are in the long tail, their presence hinders the generalisation and trustworthiness of our DNNs and makes them vulnerable to adversarial attacks.

In Chapter 5, we reparametrised the output layer to prevent unargmaxable label assignments for MLC under mild assumptions about our data (sparsity). We showed via experiments that we can indeed prevent unargmaxable outputs and explained why enforcing structure on $\mathbf{W}$ is the only way to guarantee that our classifier is not hindering our DNN.

6.1 Conclusions

Unargmaxable outputs are a problem in theory, but are they a problem in practice? We believe the answer is: it depends.

If you are a researcher working on long tail classification, extreme classification, model compression/distillation, quantisation, retrieval that leverages dot products, AI safety or robustness against adversarial attacks, we believe that unargmaxable outputs can be a problem you will face in practice. In such cases, understanding argmaxability is important, since it can motivate alternative architectures that circumvent such weaknesses and provide insights on how to make your models more robust.

If you are a practitioner applying models to various tasks, we believe it is prudent to be aware that unargmaxability can happen, so that you can diagnose and debug your model if you have unexpected results. However, we would not expect argmaxability to impact metrics that are mostly influenced by the most frequent outputs.

We now summarise our findings below.

Detecting Unargmaxability

We can unambiguously detect unargmaxable classes up to a tolerance which depends on the numerical precision of the LP solver we have available. Unargmaxability is definitely a problem worth investigating if we have a low-dimensional bottleneck. This is crucial, for example, if we work on model distillation or quantisation. While empirically we find that unargmaxable outputs become rare as we increase the dimensionality of BSLs, we have no guarantees that the problem goes away. By insisting on guaranteeing argmaxability and pushing the dimensionality of the bottleneck to the limit, we became aware of useful structured representations like the Totally Positive

Grassmanian and noticed the problem of hyperplane overcrowding. We believe the latter is an important limitation to investigate, there may be many more outputs that are unargmaxable in practice, either because the regions that correspond to the output are too small or because our idealised encoder hypothesis is wrong (Section 3.1.1).

Unargmaxability in MLC

More specifically, we saw that a $\mathrm{BSL}_{\square}$ used for MLC must have unargmaxable label assignments (Section 3.5). However, we showed that in many cases we can align the unargmaxability constraints with the constraints present in our datasets. Many MLC datasets used in practice have sparse label assignments, i.e. all outputs have at most k active labels assigned. Under this mild sparsity assumption (Section 5.1.1), we introduced criteria which guarantee that unargmaxable label assignments do not occur if we have a $\mathrm{BSL}_{\square}$ with at least $d = 2k + 1$ features (Section 5.1.2). We then introduced the DFT layer, which works in practice (Section 5.1.3), up to the limitation of epsilon argmaxability. Our method also allows us to compress the output layer dimensionality and obtain comparable performance to a standard sigmoid layer while speeding up training and using up to 50% fewer trainable parameters.

Unargmaxability in MCC

For MCC we showed that a $\mathrm{BSL}_{\triangle}$ may have unargmaxable categories, but this does not happen often in LLMs and MT models used in practice Section 4.2.

We also saw that unargmaxable categories are straightforward to prevent. Even with a $\mathrm{BSL}_{\triangle}$ having $d = 2$, we can avert unargmaxable categories by normalising the output layer weights (see Section 5.2.1).

On the other hand, a $\mathrm{BSL}_{\triangle}$ must have unargmaxable rankings (Section 3.5). However, we showed that we can guarantee all top-k rankings of the categories are argmaxable if we have a $\mathrm{BSL}_{\triangle}$ with $2k$ dimensions Section 5.2.2.

6.2 Limitations

The Idealised Encoder Assumption. In this thesis, we have shown how constraints imposed by a BSL make some outputs unargmaxable. For our guarantees we introduced in Chapter 5, we assume we have a fully-expressive encoder that can construct any feature representation (see Section 3.1.1). However, it is likely that

DNN encoders introduce constraints on which feature vectors can be produced. For example, Chiang et al. (2024) ask whether the BSL restricts an LM such that it cannot produce the distributions of a more complex model, in this case the bottlenecked LM model augmented with a non-parametric memory (Khandelwal et al., 2020). They claim that in fact BSLs can represent output distributions that are similar, in terms of KL divergence, to those produced by the mixture. This suggests that the BSL restriction is not enough to account for why augmenting a model with a non-parametric memory can achieve lower perplexity on next token prediction. An alternative explanation is that there may be feature vectors that are unreachable by the feature encoder.

We conjecture that there may be even more outputs that cannot be predicted due to constraints imposed by the encoder, constraints encountered during training, and noise present in our datasets. We have seen some such evidence in the case of glitch tokens, which are empirically hard to predict (Rumbelow et al., 2023).[1]

The Sparsity Assumption In Chapter 5, we relied on an assumption that the labels in MLC are sparse (Section 5.1.1). The sparsity assumption is realistic for most MLC tasks that rely on human annotation, because for large label vocabularies having outputs with hundreds of labels becomes a daunting annotation task. However, for datasets that are either automatically annotated or are structured prediction tasks that have been cast as MLC, this assumption may not hold.

Numerical Precision While the detection methods we have introduced are formal, they have accuracy limitations due to numerical limitations present in current state of the art LP solvers. We note that this limitation, while frustrating in practice, is unlikely to be important. This is because outputs that cannot be detected with a LP would be impossible to predict in practice anyway, as we discussed in Section 4.1.2.

6.3 Future Work

A research project is never completed, merely abandoned.

—Paul Valéry ∩ Adam Lopez

[1] We have checked some of these glitch tokens and they are argmaxable.

6.3.1 Robustly Breaking the Softmax Bottleneck

As we make the softmax bottleneck narrower, the argmaxable regions get smaller (Fig. 5.5), making label assignments harder to predict robustly. While we showed that adding slack dimensions makes the regions larger, we do not get guarantees that all regions we need are large enough to be found by the encoder. This highlights an important research direction: Can we parametrise output layers in a way that guarantees ϵ-argmaxability for large ϵ?

6.3.2 Breaking the Bottleneck via Compressed Sensing

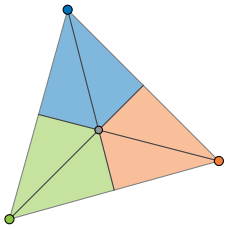
Consider a LLM that has n output tokens and $2k$-dimensional features and suppose all needed distributions over the vocabulary are sparse, i.e. the probability distributions over the vocabulary have at most k non-zero probabilities. While it is impossible for a BSL to produce such a distribution directly, if our output layer is parametrised by the Cyclic Polytope in $\mathbb{R}^{n\times 2k}$, we can obtain any k-sparse distribution exactly by using a Linear Programme (Donoho et al., 2005). This technique is more generally known as Compressed Sensing (Donoho, 2006) (see (Hastie et al., 2015, Chapter 10)). Donoho et al. (2006) give a bound on the dimensionality needed such that any k-sparse vector can be recovered via such an LP. Interestingly, we can also use Lasso regression to get a good approximation, Wainwright (2009) discusses the dimensionality needed such that Lasso retrieves the correct support of the k-sparse vector.

6.3.3 The Totally Positive Grassmanian

As we saw in Chapter 5, constraining $\mathbf{W}$ to be on the Totally Positive Grassmanian, Gr^+ (Definition 2.2.15), is a sensible constraint, both for MLC and MCC. We conjecture that the structure of Gr^+ exists in subsets of our neural network parameters learned via gradient descent. Proving this is true and detecting such subsets of parameters may be interesting for two reasons:

- If it occurs often, we are wasting compute fitting these parameters. We could start training our network with a subset of parameters already on Gr^+ and learn slack parameters or an adaptor for fine-tuning.
- If we can detect subsets of parameters having Gr^+ structure, we may be able to use this information as an argmaxability certificate.

Another relevant direction here is an open question we introduced in Section 5.1. There, we outlined the criteria needed for all k-active labels to be argmaxable (Section 5.1.2). The DFT matrix (Eq. (5.10)) satisfied these criteria, so we picked it due to its alluring properties. But the DFT matrix is just one point on Gr^{+}. Do we gain anything if we parametrise $\mathbf{W}$ such that it can be any point on Gr^{+}?.



A Appendix

A.1 # Regions in Hyperplane Arrangements

Counting the Regions of the Restricted Braid Arrangement

The number of feasible permutations is invariant to specific choices of $\mathbf{W}$ and $\mathbf{b}$ (Cover, 1967; Smith, 2014) and only depends on the dimensionality of the softmax inputs d, the number of categories n and whether we specify a bias term $\mathbf{b}$ not in the columnspace of $\mathbf{W}$. Namely, the cardinality of the set of feasible permutations does not change, but the members of the set do — they depend on the specific values in $\mathbf{W}$ and $\mathbf{b}$. There exists a recurrence formula to obtain the number of feasible permutations for a particular n and d (Good et al., 1977; Kamiya et al., 2005). See our code `https://github.com/andreasgrv/unargmaxable` and the relations in Smith (2014) for more details.

Softmax with no Bias Term The number of feasible permutations as a function of n and d when we have a softmax with no bias term can be seen in Table A.1. When $d \geq n-1$ all permutations corresponding to ways of ranking n categories are feasible (table cells with $d = n-1$ are highlighted in bold). However, as we make the softmax bottleneck narrower, we can represent less permutations, as can be seen from the numbers reported below the diagonal.

Softmax with Bias Term The number of feasible permutations as a function of n and d when we have a softmax with a bias term is larger as can be seen in Table A.2. As we saw in Figure 3.17, this is because a bias term can offset the representable

# Categories n	Bottleneck dimensionality d									
	1	2	3	4	5	6	7	8	9	10
2	**2**	2	2	2	2	2	2	2	2	2
3	2	**6**	6	6	6	6	6	6	6	6
4	2	*12*	**24**	24	24	24	24	24	24	24
5	2	20	72	**120**	120	120	120	120	120	120
6	2	30	172	480	**720**	720	720	720	720	720
7	2	42	352	1512	3600	**5040**	5040	5040	5040	5040
8	2	56	646	3976	14184	30240	**40320**	40320	40320	40320
9	2	72	1094	9144	45992	143712	282240	**362880**	362880	362880
10	2	90	1742	18990	128288	557640	1575648	2903040	**3628800**	3628800

Table A.1: Number of permutation regions defined by a bottlenecked softmax layer $\sigma_{\triangle}(\mathbf{Wx})$ with no bias term. When $d \geq n-1$ all permutations corresponding to ways of ranking n classes are feasible. 12 in italics corresponds to the number of regions shown in the left Subfigure of Figure 3.17. https://oeis.org/A071223.

linear subspace to an affine subspace which can intersect more regions of the Braid Arrangement.

A.2 Sigmoid Bottlenecks are Softmax Bottlenecks

Here we show that a sigmoid layer parametrised by $\mathbf{W} \in \mathbb{R}^{n \times d}$ can be rewritten as a softmax layer parametrised by $\mathbf{A} \in \mathbb{R}^{2^n \times d}$, where $\mathbf{A} = \mathbf{CW}$, and $\mathbf{C} \in \{0,1\}^{2^n \times n}$ comprises the vertices of the n-cube. More concretely, we show:

$$P(\mathbf{y} \mid \mathbf{x}) = \underbrace{\prod_{i=1}^{n} P(y_i \mid \mathbf{x})}_{\substack{\text{sigmoid} \\ \text{factorised}}} = \prod_{i=1}^{n} \sigma_{\square}\left(y_i \mathbf{w}_i^\top \mathbf{x}\right) = \underbrace{\sigma_{\triangle}(\mathbf{Ax})_k}_{\substack{\text{softmax} \\ \text{bottleneck}}} \tag{A.1}$$

where $k = \sum_{i=1}^{n} 2^{i-1} \cdot [y_i = 1]$, i.e. k indexes the probability of each $\mathbf{y}$.

A.2.1 Notation

We have n binary labels leading to 2^n possible label assignments $\mathbf{y} = \{-1, +1\}^n$. We assume feature vectors $\mathbf{x}$ are in $\mathbb{R}^d$. In order to compute $P(\mathbf{y} \mid \mathbf{x})$, we assume conditional independence of each y_i given $\mathbf{x}$ and get:

$$P(\mathbf{y} \mid \mathbf{x}) = \prod_{i=1}^{n} P(y_i \mid \mathbf{x}) = \prod_{i=1}^{n} \sigma_{\square}\left(y_i \mathbf{w}_i^\top \mathbf{x}\right)$$

# Categories n	Bottleneck dimensionality d									
	1	2	3	4	5	6	7	8	9	10
2	**2**	2	2	2	2	2	2	2	2	2
3	4	**6**	6	6	6	6	6	6	6	6
4	7	*18*	**24**	24	24	24	24	24	24	24
5	11	46	96	**120**	120	120	120	120	120	120
6	16	101	326	600	**720**	720	720	720	720	720
7	22	197	932	2556	4320	**5040**	5040	5040	5040	5040
8	29	351	2311	9080	22212	35280	**40320**	40320	40320	40320
9	37	583	5119	27568	94852	212976	322560	**362880**	362880	362880
10	46	916	10366	73639	342964	1066644	2239344	3265920	**3628800**	3628800

Table A.2: Number of permutation regions defined by a bottlenecked softmax layer $\sigma_{\triangle}(\mathbf{W}\mathbf{x}+\mathbf{b})$. When $d \geq n-1$ all permutations corresponding to ways of ranking n categories are feasible. 18 in italics corresponds to the number of regions shown in the right Subfigure of Figure 3.17.

To reduce clutter we will use z_i to denote the ith logit, i.e. $z_i = \mathbf{w}_i^\top \mathbf{x}$. We now show how to rewrite the sigmoid layer as a softmax layer.

A.2.2 Matching the Sigmoid and Softmax Partition Functions

It is instructive to identify the partition function of the large softmax layer that is formed when we construct the product of sigmoid terms. Note that the denominator below for both $P(y_i = 1 \mid \mathbf{x})$ and $P(y_i = -1 \mid \mathbf{x})$ is the same.

$$P(y_i = 1 \mid \mathbf{x}) = \frac{e^{z_i}}{1+e^{z_i}} \tag{A.2}$$

$$P(y_i = -1 \mid \mathbf{x}) = 1 - \frac{e^{z_i}}{1+e^{z_i}} \tag{A.3}$$

$$= \frac{1+e^{z_i}}{1+e^{z_i}} - \frac{e^{z_i}}{1+e^{z_i}} \tag{A.4}$$

$$= \frac{1}{1+e^{z_i}} \tag{A.5}$$

Therefore, when we expand the product $\prod_{i=1}^{n} P(y_i \mid \mathbf{x})$ for any assignment $\mathbf{y} \in \{-1, 1\}^n$ the denominator will be the same term $\mathcal{Z} = \prod_{i=1}^{n}(1+e^{z_i})$. To see this more clearly, see the example below for n=2:

$$P(y_1 = 1, y_2 = 1 \mid \mathbf{x}) = \frac{e^{z_1}}{1+e^{z_1}} \frac{e^{z_2}}{1+e^{z_2}} = \frac{e^{z_1+z_2}}{\mathcal{Z}} = \frac{e^{(w_1+w_2)^\top \mathbf{x}}}{\mathcal{Z}}$$

$$P(y_1 = 1, y_2 = -1 \mid \mathbf{x}) = \frac{e^{z_1}}{1+e^{z_1}} \frac{1}{1+e^{z_2}} = \frac{e^{z_1}}{\mathcal{Z}}$$
$$P(y_1 = -1, y_2 = 1 \mid \mathbf{x}) = \frac{1}{1+e^{z_1}} \frac{e^{z_2}}{1+e^{z_2}} = \frac{e^{z_2}}{\mathcal{Z}}$$
$$P(y_1 = -1, y_2 = -1 \mid \mathbf{x}) = \frac{1}{1+e^{z_1}} \frac{1}{1+e^{z_2}} = \frac{1}{\mathcal{Z}}$$

Now, note that the above is equivalent to $\sigma_{\triangle}(\mathbf{Ax})$ where:

$$\mathbf{A} = \begin{bmatrix} \mathbf{w}_1 + \mathbf{w}_2 \\ \mathbf{w}_1 \\ \mathbf{w}_2 \\ \mathbf{0} \end{bmatrix} \tag{A.6}$$

and

$$\mathcal{Z} = (1+e^{\mathbf{z}_1})(1+e^{\mathbf{z}_2}) = 1 + e^{\mathbf{z}_1} + e^{\mathbf{z}_2} + e^{\mathbf{z}_1+\mathbf{z}_2} \tag{A.7}$$

A.2.3 Equivalent Softmax Parameters $\mathbf{A} = \mathbf{CW}$

More generally, we have $\mathbf{A} \in \mathbb{R}^{2^n \times d}$ where each row $\mathbf{a}_i$ of $\mathbf{A}$ is $\mathbf{a}_i = \sum_{i:y_i=1} \mathbf{w}_i$. Or equivalently, $\mathbf{A} = \mathbf{CW}$, where $\mathbf{C} = \text{verts}(\square_n)$ is the matrix comprising the vertices of the n-cube in 0-1 format (see Definition 2.3.3).

A.3 Derivation of Linear Programmes

A.3.1 MCC

There are two ways to check if a category is argmaxable: a) Check that the halfspace intersections of all rankings that rank the target category above the rest is not empty and b) Check that the representation of the category is *not* internal to the convex hull of the remaining representations.

We already gave the halfspace intersection LP in Eq. (4.9). Here we elaborate on the convex hull version.

Point Interior to Convex Hull

We want to check whether a point $\mathbf{p} \in \text{conv}(\mathbf{W})$. This is true if we can write $\mathbf{p}$ as a convex combination (Section 2.2.1) of the $\mathbf{w}_i$, $i \in [n]$. We can use a LP to check if this is possible. We search for $\mathbf{x}$, the coefficients of the convex combination. Therefore,

$\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{C} \in \mathbb{R}^{n \times (d+1)}$, $\mathbf{d} \in \mathbb{R}^{d+1}$. The additional feature dimension comes from the convex combination constraint $\sum_i x_i = 1$, which we also highlight in orange in the matrix.

$$\mathbf{C} = \begin{bmatrix} 1 & w_{1,1} & w_{1,2} & \dots & w_{1,d} \\ 1 & w_{2,1} & w_{2,2} & \dots & w_{2,d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w_{n,1} & w_{n,2} & \dots & w_{n,d} \end{bmatrix} \quad \mathbf{d} = \begin{bmatrix} 1 \\ p_1 \\ p_2 \\ \vdots \\ p_d \end{bmatrix} \tag{A.8}$$

Point in Convex Hull LP

$$\text{maximise} \quad \mathbf{c}^\top \mathbf{x} \tag{A.9}$$

$$\text{subject to} \quad \mathbf{C}^\top \mathbf{x} = \mathbf{d} \tag{A.10}$$

$$\mathbf{x} > \mathbf{0} \tag{A.11}$$

A.3.2 MLC

In order to check whether a label assignment $\mathbf{y}$ is argmaxable, we need to check whether there exists an input $\mathbf{x}$ which can be assigned such a $\mathbf{y}$. For this to be possible, there must be an input $\mathbf{x}$ for which the dot product with the corresponding binary classifier $\mathbf{w}_i$ agrees in sign with y_i; i.e. $\mathbf{w}_i^\top \mathbf{x} > 0$ for $y_i = +$ and $\mathbf{w}_i^\top \mathbf{x} < 0$ for $y_i = -$. From this perspective, each $\mathbf{w}_i$ and the corresponding sign of the label y_i define a halfspace and we are checking if the intersection of halfspaces (Definition 2.4.3) exists (or not).

A.3.3 Halfspace Constraints

Below we derive the constraints we want to encode for the Linear Programme. More precisely, if the intersection of halfspaces exists we also want to find the largest margin $\epsilon \|\mathbf{w}_i\|_2$ for which this is true (Chebyshev LP). In the case $\mathbf{y}_i = +$ we want the dot product to be positive even if we subtract the margin $\epsilon \|\mathbf{w}_i\|_2$. With the same motivation for $\mathbf{y}_i = -$, we get the constraints $LP(\mathbf{y}_i)$:

$$LP(\mathbf{y}_i) = \begin{cases} \mathbf{w}_i^\top \mathbf{x} - \epsilon \|\mathbf{w}_i\|_2 \geq 0 & \text{for } y_i = + \\ \mathbf{w}_i^\top \mathbf{x} + \epsilon \|\mathbf{w}_i\|_2 \leq 0 & \text{for } y_i = - \end{cases} \tag{A.12}$$

We can rewrite the $y_i = +$ case by multiplying by -1:

$$\mathbf{w}_i^\top \mathbf{x} - \epsilon \|\mathbf{w}_i\|_2 \geq 0 \implies \tag{A.13}$$

$$-\mathbf{w}_i^\top \mathbf{x} + \epsilon \|\mathbf{w}_i\|_2 \leq 0 \tag{A.14}$$

and succinctly combine both cases:

$$-y_i \mathbf{w}_i^\top \mathbf{x} + \epsilon \|\mathbf{w}_i\|_2 \leq 0 \tag{A.15}$$

where we abuse notation and assume y_i takes values $+1$ and -1 correspondingly.

A.3.4 Box Constraints

In order for the Chebyshev center to be defined, we need to bound the magnitude of each dimension of $\mathbf{x}$. As such, we assume the activations $\mathbf{x}$ are independently bounded to have magnitude less than 10^4, i.e. we have box constraints:

$$-10^4 \leq x_j \leq 10^4, \quad 1 \leq j \leq d \tag{A.16}$$

A.3.5 LP Sensitivity

In theory, the constraint for the margin of the Chebyshev LP is that ϵ must be positive. However, Gurobi has a sensitivity limit of 10^{-9}, so we set eps $= 10^{-8}$ and obtain:

$$\epsilon > \text{eps} \tag{A.17}$$

A.3.6 Summary

We combine all the above to get the optimisation problem:

$$\text{maximise} \quad \epsilon \tag{A.18}$$

$$\text{subject to} \quad -y_i \mathbf{w}_i^\top \mathbf{x} + \epsilon \|\mathbf{w}_i\|_2 \leq 0, \quad 1 \leq i \leq n \tag{A.19}$$

$$-10^4 \leq x_j \leq 10^4, \quad 1 \leq j \leq d, \quad \epsilon > \text{eps} \tag{A.20}$$

A.3.7 Matrix Format

We need to find the radius of the largest ball that fits in the intersection of halfspaces. We assign the radius variable to x_1, and the centroid of the ball to $\mathbf{x}_{2:d+2}$. The centroid has $d+1$ dimensions, since we have a bias term. Therefore, $\mathbf{x} \in \mathbb{R}^{d+2}$ and the inequality

constraints are $\mathbf{A} \in \mathbb{R}^{n\times(d+2)}$, $\mathbf{b} \in \mathbb{R}^n$. In order to make the last column of $\mathbf{A}$ a bias term, we also need the equality constraint $\mathbf{C} \in \mathbb{R}^{1\times(d+2)}$, $\mathbf{d} \in \mathbb{R}$ which sets $x_{d+2} = 1$. Lastly, the cost vector $\mathbf{c}$ only takes x_0 into account, as we want to maximise the radius.

$$\mathbf{A} = \begin{bmatrix} \|\mathbf{w}_1\|_2 & w_{1,1} & w_{1,2} & \dots & w_{1,d} & b_1 \\ \|\mathbf{w}_2\|_2 & w_{2,1} & w_{2,2} & \dots & w_{2,d} & b_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \|\mathbf{w}_n\|_2 & w_{n,1} & w_{n,2} & \dots & w_{n,d} & b_n \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} -\epsilon \\ -\epsilon \\ \vdots \\ -\epsilon \\ -\epsilon \end{bmatrix} \tag{A.21}$$

$$\mathbf{C} = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix} \quad \mathbf{d} = \begin{bmatrix} 1 \end{bmatrix} \tag{A.22}$$

$$\mathbf{c} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \end{bmatrix} \tag{A.23}$$

Chebyshev LP in Matrix Format

$$\text{maximise} \quad \mathbf{c}^\top \mathbf{x} \tag{A.24}$$

$$\text{subject to} \quad \mathbf{A}\mathbf{x} \leq \mathbf{b} \tag{A.25}$$

$$\mathbf{C}\mathbf{x} = \mathbf{d} \tag{A.26}$$

$$\text{eps} \leq x_1 \tag{A.27}$$

$$-10^4 \leq x_j \leq 10^4, \quad 2 \leq j \leq d+2 \tag{A.28}$$

A.4 Unargmaxable Token Search Results

model	# potentially unargmaxable	# unargmaxable
opus-mt-ja-en	109	2
opus-mt-ru-en	90	159
opus-mt-bg-en	93	53
opus-mt-ja-en(2)	14	0
opus-mt-ar-en	40	184
opus-mt-en-el	75	42
opus-mt-de-el	115	6
opus-mt-ar-el	41	0
opus-mt-es-el	67	32
opus-mt-fi-el	57	8
opus-mt-ar-he	3	0
opus-mt-de-he	4	0
opus-mt-es-he	3	0
opus-mt-fr-he	1	0
opus-mt-fi-he	7	0
opus-mt-ja-he	0	0
opus-mt-en-ar	21	2
opus-mt-el-ar	12	0
opus-mt-es-ar	17	1
opus-mt-fr-ar	17	0
opus-mt-he-ar	7	0
opus-mt-it-ar	8	0
opus-mt-ja-ar	4	0
opus-mt-pl-ar	52	0
opus-mt-ru-ar	8	0
opus-mt-en-ru	98	34
opus-mt-es-ru	42	18
opus-mt-fi-ru	1	0
opus-mt-fr-ru	34	43
opus-mt-he-ru	5	0
opus-mt-ja-ru	13	0
opus-mt-ko-ru	2	0

Figure A.1: Results for Helsinki NLP OPUS models. For 13/32 models some infrequent tokens were found to be unargmaxable.

model	# potentially unargmaxable	# unargmaxable
bert-base-cased	0	0
bert-base-uncased	0	0
roberta-base	0	0
roberta-large	0	0
xlm-roberta-base	0	0
xlm-roberta-large	0	0
gpt2	0	0

Figure A.2: Results for LMs. No tokens were found to be unargmaxable.

model	# potentially unargmaxable	# unargmaxable
facebook/wmt19-en-ru	5	0
facebook/wmt19-ru-en	64	0
facebook/wmt19-de-en	173	0
facebook/wmt19-en-de	184	0

Figure A.3: Results for FAIR WMT'19 models. No tokens were found to be unargmaxable.

Figures: **potentially unargmaxable** is the number of tokens that the approximate algorithm failed to prove were argmaxable. **unargmaxable** is the number of unargmaxable tokens according to the exact algorithm.

model	# potentially unargmaxable	# unargmaxable
cs-en.student.base	0	0
es-en.teacher.bigx2(1)	0	0
es-en.teacher.bigx2(2)	0	0
en-es.teacher.bigx2(1)	0	0
en-es.teacher.bigx2(2)	0	0
et-en.teacher.bigx2(1)	2	0
et-en.teacher.bigx2(2)	1	0
en-et.teacher.bigx2(1)	1	0
en-et.teacher.bigx2(2)	1	0
nb-en.teacher.base	0	0
nn-en.teacher.base	0	0
is-en.teacher.base	0	0
cs-en.student.base	0	0
cs-en.student.tiny11	0	0
en-cs.student.base	0	0
en-cs.student.tiny11	0	0
en-de.student.base	0	0
en-de.student.tiny11	0	0
es-en.student.tiny11	0	0
en-es.student.tiny11	0	0
et-en.student.tiny11	0	0
en-et.student.tiny11	0	0
is-en.student.tiny11	0	0
nb-en.student.tiny11	0	0
nn-en.student.tiny11	0	0

Figure A.4: Results for Bergamot models. No tokens were found to be unargmaxable. Interestingly, student models were much easier to prove argmaxable than teacher models, despite student model Softmax weights being lower dimensional.

Figures: **potentially unargmaxable** is the number of tokens that the approximate algorithm failed to prove were argmaxable. **unargmaxable** is the number of unargmaxable tokens according to the exact algorithm.

model	# potentially unargmaxable	# unargmaxable
en-cs.l2r(1-4)	≤ 2	0
en-cs.r2l(1-4)	≤ 1	0
cs-en.l2r(1-4)	≤ 2	0
cs-en.r2l(1-4)	0	0
en-de.l2r(1-4)	≤ 1	0
en-de.r2l(1-4)	≤ 2	0
de-en.l2r(1-4)	≤ 2	0
de-en.r2l(1-4)	0	0
en-ru.l2r(1-4)	0	0
ru-en.l2r(1-4)	0	0
ru-en.r2l(1-4)	0	0
en-tr.l2r(1-4)	≤ 5	0
en-tr.r2l(1-4)	≤ 4	0
lv-en.l2r(1-4)	0	0
lv-en.r2l(1-4)	≤ 1	0
tr-en.l2r(1)	2	0
tr-en.l2r(2)	8	0
tr-en.l2r(3)	6	0
tr-en.l2r(4)	2	0
tr-en.r2l(1)	4	0
tr-en.r2l(2)	0	0
tr-en.r2l(3)	6	0
tr-en.r2l(4)	4	0
en-zh.l2r(1)	3	0
en-zh.l2r(2)	3	0
en-zh.l2r(3)	14	0
en-zh.l2r(4)	1	0
en-zh.r2l(1)	2	0
en-zh.r2l(2)	0	0
en-zh.r2l(3)	7	0
en-zh.r2l(4)	7	0
zh-en.l2r(1)	8	0
zh-en.l2r(2)	3	0
zh-en.l2r(3)	366	0
zh-en.r2l(1-3)	≤ 3	0

Figure A.5: Results for Edinburgh WMT'17 submission (ensemble) models. **r2l** and **l2r** refer to training direction, with **l2r** denoting training left to right and **r2l** right to left. The models were ensembles, hence there are more than one model per language pair and direction. When all models per language pair and direction have less than 5 counts, we summarise all models with a single row, e.g. (1-4).

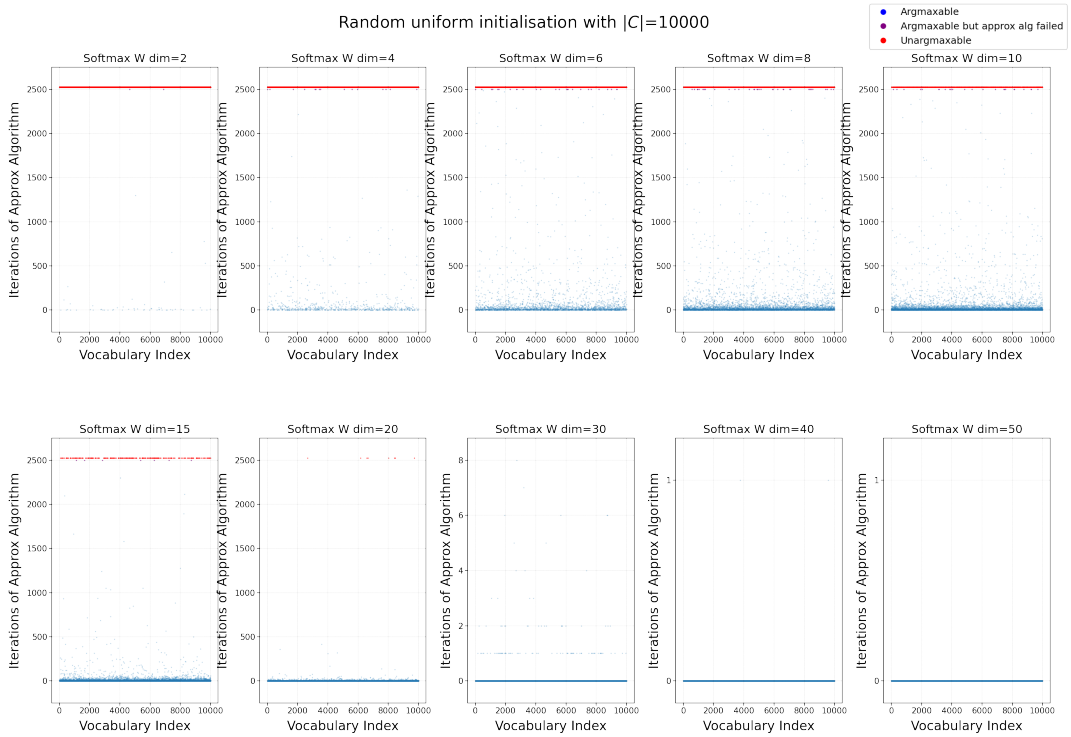


Figure A.6: Unargmaxable tokens become extremely unlikely in random matrices of large dimensions. On the x-axis, we assign each of the 10000 categories of softmax an index. On the y-axis we plot a blue dot for the number of iterations of the approximate algorithm needed to show the corresponding token is argmaxable, or a red dot at the top if it is unargmaxable. The softmax weights, $\mathbf{W} \in \mathbb{R}^{n \times dim}$, and bias term, $\mathbf{b} \in \mathbb{R}^n$, are initialised using a uniform $U(-1, 1)$ distribution. As can be seen by the decreasing number of red dots in the plots from top left to bottom right, unargmaxable tokens are less unlikely to occur as we increase dim. Moreover, the braid reflect approximate algorithm fails less and needs fewer iterations to find an input that proves a token is argmaxable. For example, for the bottom right two figures most tokens are shown to be argmaxable with 1 or 0 iterations.

A.5 Activation Range of Softmax Layer Inputs

Neural network activations are bounded in magnitude in practice, since larger activations can lead to larger gradients and instability during training. In this work, we made the assumption that the softmax layer inputs $\mathbf{x}$ are bounded within a range for all dimensions: $-100 \leq \mathbf{x} \leq 100$. Below we provide some supporting empirical evidence that this assumption is reasonable.

We checked this assumption on 2 Helsinki NLP OPUS models for en-ru and bg-en, which were found to have unargmaxable tokens. We took 10 million sentence pairs

from OPUS as released in Tiedemann (2020) for the corresponding language pairs and input them to the corresponding models, decoding using the gold translations. We then recorded the range of the minimum and maximum activation for the softmax layer inputs.

Since our assumption is that all 512 dimensions are bounded between -100 and 100, we focus on the range of the minimum and maximum activation for each output token across all dimensions. We therefore calculate a 99 percentile for the min and max activation per token across all dimensions as well as the overall min and max activations overall. The results can be seen in Table A.3, from which we can see that for these two models our assumption holds for all activations produces for 10 million sentences and the percentiles show that more than 99% of the extreme values fall within the $[-50, 50]$ range.

model	min range	max range	min	max
bg-en	$[-37.5, -9.4]$	$[12.1, 40.3]$	-57.47	58.87
en-ru	$[-41.6, -9.9]$	$[10.9, 36.4]$	-95.4	94.4

Table A.3: Range of activations for softmax inputs as calculated on 10 million sentence pairs from OPUS. Ranges are 99 percentiles and min and max are the largest activation across all dimensions for all sentences.

A.6 Reproducibility

A.6.1 Dataset Access and Preprocessing

MIMIC-III

While de-identified, the MIMIC-III dataset (Johnson et al., 2016) contains sensitive and detailed information on the clinical care of patients. As such, permission to access this dataset needs to be requested, as explained here.[1] We used the same preprocessing, setup and train, validation and test splits as (Mullenbach et al., 2018). See their github repository for more details.

[1] https://mimic.mit.edu/docs/gettingstarted/

BioASQ Task A 2021

The BioASQ Task A dataset (Nentidis et al., 2021; Tsatsaronis et al., 2015) is available after registering for the task on the BioASQ website.[2] We created dataset splits which cover n =20k labels using a 1m subset of the 2021 BioASQ task A dataset. We construct train, validation and test splits by sampling examples, making sure that all individual labels (not label combinations) occur in both the train and test sets. We encode the concatenation of the journal, title and abstract as text input. Due to the context size limitation of BERT, we truncate the input to the first 512 subwords. See our code for more details.

OpenImages v6

The OpenImages v6 dataset (Kuznetsova et al., 2020) can be accessed from the project website.[3] We downloaded the images from CVDF, which was linked from the website. Since the dataset is very large, we only used $N = 108228$ images, these had hashes that started with 1 and were available as a single zip download from CVDF. Since the validation and test sets are also large, we validate and test on the first 5k examples of the validation set and the first 10k examples of the test set, correspondingly. For preprocessing, we simply reshape all images to 448x448, as done in Baruch et al. (2020).

A.6.2 Dataset Statistics

We tabulate the sizes of the dataset splits in Table A.4.

A.6.3 Hyperparameters

In order to study the sensitivity of our methods to random initialisation we ran all experiments three times, once per random seed in (0, 1, 2). We train all models using binary crossentropy loss. We summarise all hyperparameters in Table A.4. We use early stopping for all models with a patience of 10. The stopping criterion is Prec@8 for MIMIC-III and Validation Loss for BioASQ and OpenImages.

[2] http://participants-area.bioasq.org/
[3] https://storage.googleapis.com/openimages/web/index.html

	Datasets		
	MIMIC-III	BioASQ task A	OpenImagesV6
n	8921	20000	8933
d	25-400	25-400	25-400
encoder	CNN	PubmedBERT	T-Resnet-L
pretrained	no	yes	yes
encoder dim e	500	768	2432
lr (encoder)	0.001	0.00005	0.0001
lr (classifier)	0.001	0.001	0.001
batch size	16	32	64
patience	10	10	10
eval every	1 epoch	500 steps	250 steps
criterion	P@8	Valid loss	Valid loss
# train	44k	100k	108k
# valid	1.6k	5k	5k
# test	3.3k	10k	10k

Table A.4: MLC dataset attributes and model hyperparameters.

A.6.4 Training Resources and Train time

We train and evaluate our models on GPUs. For the MIMIC-III dataset we used a NVIDIA 3090 GPU that has 24Gb of RAM and for the remaining two datasets we used an NVIDIA RTX A6000 which has 48Gb of RAM. The experiments took about two weeks of compute. More specifically, the MIMIC-III runs took 9 hours, the OpenImages runs took 85 hours and the BioASQ runs took 188 hours. We verified our models using the LP on CPU on a cluster with an AMD EPYC 7452 32-Core Processor and 500GB of RAM. We parallelised verification by running the LP for each label in parallel to others of the same model. With this setup and running on 50 threads the verification of 10k examples takes from between 20 minutes to 3 hours, depending on the dimensionality (slower for large d).

A.7 The Cyclic Polytope

To prove Proposition 5.1 in the next section, we leverage the theorem in Cordovil et al. (2000), who present their result in terms of the homogenisation of the Cyclic Polytope.

Here, we highlight the equivalence of the homogenisation of the Cyclic Polytope to a DFT matrix with total order constraints on the t_i. We will need this to make claims about the maximal minors of the DFT matrix.

The Cyclic Polytope We start with the standard definition. The $\mathcal{C}_{n,d}$ with n vertices in $\mathbb{R}^d$ is defined as the convex hull of $n > d$ distinct points on the moment curve in $\mathbb{R}^d$ (Ziegler, 1995, Example 0.6, p.11). The moment curve is a map $m(t) : \mathbb{R} \mapsto \mathbb{R}^d$ defined as:

$$\mathcal{C}_{n,d} = \operatorname{conv}\left(m(t_1), m(t_2), \ldots, m(t_n)\right) \tag{A.29}$$

$$\text{where} \quad m(t) = \begin{bmatrix} t \\ t^2 \\ \vdots \\ t^d \end{bmatrix}, \quad t_1 < t_2 < \ldots < t_n \tag{A.30}$$

Homogenisation In order to study the affine dependencies of a point configuration in $\mathbb{R}^d$ (e.g. the face structure of a polytope), it is convenient to map them to linear dependencies of a vector configuration in $\mathbb{R}^{d+1}$ and study those instead. This can be done via homogenisation: we map points in $\mathbb{R}^d$ to vectors in $\mathbb{R}^{d+1}$ by appending an extra dimension and fixing it to 1 (Ziegler, 1995, Section 6.2), i.e. $\hom : \mathbb{R}^d \to \mathbb{R}^{d+1}, \hom(\mathbf{x}) = \begin{pmatrix} 1 \\ \mathbf{x} \end{pmatrix}$. To abide by earlier notation, we stack the vertices of the Cyclic Polytope in the rows of the matrix. For the standard Cyclic Polytope on the moment curve we get a Vandermonde matrix:

$$\mathcal{C}_{n,d} = \begin{bmatrix} t_1 & t_1^2 & \cdots & t_1^d \\ t_2 & t_2^2 & \cdots & t_2^d \\ \vdots & \vdots & \ddots & \vdots \\ t_n & t_n^2 & \cdots & t_n^d \end{bmatrix} \tag{A.31}$$

$$\hom\left(\mathcal{C}_{n,d}\right) = \begin{bmatrix} 1 & t_1 & t_1^2 & \cdots & t_1^d \\ 1 & t_2 & t_2^2 & \cdots & t_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_n & t_n^2 & \cdots & t_n^d \end{bmatrix} \tag{A.32}$$

Trigonometric Cyclic Polytope Instead of the moment curve, Gale (1963) used the trigonometric moment curve to construct the Cyclic Polytope, see also Donoho et al. (2005, Section 3). We note that its homogenisation is the truncated DFT matrix:

$$\mathcal{C}_{n,2k} = \begin{bmatrix} \cos t_1 & \sin t_1 & \cdots & \cos kt_1 & \sin kt_1 \\ \cos t_2 & \sin t_2 & \cdots & \cos kt_2 & \sin kt_2 \\ \vdots & \vdots & \ddots & \vdots & \\ \cos t_n & \sin t_n & \cdots & \cos kt_n & \sin kt_n \end{bmatrix} \tag{A.33}$$

$$\hom\left(\mathcal{C}_{n,2k}\right) = \begin{bmatrix} 1 & \cos t_1 & \sin t_1 & \cdots & \cos kt_1 & \sin kt_1 \\ 1 & \cos t_2 & \sin t_2 & \cdots & \cos kt_2 & \sin kt_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ 1 & \cos t_n & \sin t_n & \cdots & \cos kt_n & \sin kt_n \end{bmatrix}$$

$$t_i = \frac{2\pi(i-1)}{n}, i \in [n] \tag{A.34}$$

Cyclic Polytopes can be thought of as projections of the simplex that retain as much of the low-dimensional faces as possible given the available dimensionality. That is, if we stack the vertices of the polytope in the rows of a matrix and think of the matrix as a projection matrix, as we did in Section 3.5.2, the projection is such that if we project a high-dimensional polytope, a lot of its low-dimensional faces "survive" the projection (Donoho et al., 2006, Section 1.2). Thinking of the cyclic polytope itself as the projection of the standard simplex:

- In $\mathbb{R}^2$ all vertices of the simplex survive.
- In $\mathbb{R}^4$ all edges of the simplex survive.
- In $\mathbb{R}^6$ all faces of the simplex survive.
- More generally, in $\mathbb{R}^{2d}$ all d-faces survive.

We will make use of this alluring property in Chapter 5 to guarantee specific subsets or rankings are argmaxable.

A.8 Proofs

A.8.1 Theorem 5.3

We show that for a classifier $\mathbf{W} \in \mathsf{Gr}^{+}_{n,d}$ the argmaxable label assignments are the $(d-1)$-alternating vectors. We prove Theorem 5.3 by invoking Theorem 5.2 to guarantee that any sign vector has at most $d-1$ sign changes. We then use a counting argument to show that the number of argmaxable sign vectors is the same as the number of

alternating sign vectors, and as such the argmaxable sign vectors are exactly the $(d-1)$-alternating vectors.

Proof. Since $\mathbf{W} \in \mathsf{Gr}^{+}_{n,d}$, all maximal minors are non-zero and hence the rows of $\mathbf{W}$ are in general position (Definition 2.2.11). By invoking Theorem 5.1, the number of argmaxable label assignments is $|\mathcal{A}(\mathbf{W})| = 2\sum_{d'=0}^{d-1}\binom{n-1}{d'}$. Note that this is exactly the number of $(d-1)$-alternating label assignments $\left|V_{n,d-1}\right|$, as we elaborate next. The binomial coefficient comes from choosing d' out of the $n-1$ positions between labels to flip the sign and we sum over all possible number of sign changes up to $d-1$. For each alternating $\mathbf{y}$, we can produce another by flipping all signs, hence the leading multiplier 2. Now, from Theorem 5.2, none of the label assignments can have more than $d-1$ sign changes; hence they must be exactly the $(d-1)$-alternating label assignments: $\mathcal{A}(\mathbf{W}) = V_{n,d-1}$. □

A.8.2 Proof of Proposition 5.1

We now show that the maximal minors of the DFT matrix are non-zero and have the same sign. In fact, we show this is true more generally for any matrix with rows that are homogenised vertices of an even dimensional Cyclic Polytope.

Proof. The DFT matrix corresponds to the homogenisation of the vertices of a Cyclic Polytope (Cordovil et al., 2000; Gale, 1963), see Appendix A.7, where the vertices of the Cyclic Polytope are in $2k$ dimensions before being homogenised. Cyclic Polytopes in $2k$ dimensions (even dimension) are rigid[4]: their face structure determines their geometric structure (Cordovil et al., 2000, Theorem 5.1). Their geometric structure is that of a Uniform Alternating Oriented Matroid [5]. Any matrix realisation of a Uniform Alternating Oriented Matroid has maximal minors that agree in sign and are non-zero (Cordovil et al., 2000; Sturmfels, 1988, Proposition 3.1), see chirotope representation of Oriented Matroids (Björner et al., 1999, Section 9.4). Any normalisation of the DFT matrix obtained by scaling columns using non-zero scalars does not alter the columnspace of the matrix, and hence the oriented matroid structure is unchanged: the orthants intersected by the columnspace are the same. □

[4] See (Ziegler, 1995, Section 6.6) for more details on rigidity, and Example 6.3 for a polytope that is not rigid.

[5] By this we mean that the matrix obtained by the homogenisation of any Cyclic Polytope has the structure of the Uniform Alternating Oriented Matroid.

A.9 Why DFT Regions Become Very Small

As we saw in Fig. 5.5, while we can provide guarantees that all k-active label assignments are argmaxable, they may not be epsilon argmaxable (i.e. argmaxable with a large margin). This is because, due to hyperplane overcrowding (Section 4.1.1), the regions may become too small to detect via our LP from Chapter 4.

Here we provide some thoughts on why the DFT regions become small when $d \ll n$. For a matrix $\mathbf{W}$ to be in $\mathsf{Gr}^{+}_{n,d}$, all its maximal minors have to be non-zero and have the same sign. Let us assume they are all positive. For the matrix to "robustly" have this structure, the maximal minors should be large; the larger they are the more the row vectors will have to change before some become collinear and make one of the maximal minors zero. We note that the maximal minors are affected by both the angles between the vectors and the Euclidean norm of the vectors. As such, one consideration is whether some vectors are more sensitive to perturbations due to their norm, e.g. row vectors having a small norm would generally be more sensitive to perturbations than row vectors that have a large norm. However, for the DFT matrix, we can ignore the effect of norms because the norm of any row vector is the same.

a) Row vectors of the DFT matrix have equal l2 norm. We can show (see Appendix A.9.1) that all row vectors $\mathbf{w}_i$ of $\mathbf{W} \in \mathbb{R}^{n\times(2k+1)}$, $k \in \mathbb{N}, k \geq 1$ have norm given by:

$$\|\mathbf{w}_i\|_2 = \sqrt{\frac{2k+1}{n}} \tag{A.35}$$

b) Maximal minor constraints for orthonormal matrices. In addition, the truncated DFT matrix is also an orthonormal matrix (its columns are pairwise orthogonal and have norm 1). An orthonormal matrix $\mathbf{M} \in \mathbb{R}^{n\times d}$, $d < n$ with maximal minors Δ_I indexed by d-subsets of rows, has the property that the sum of squares of its maximal minors is 1:

$$\sum_{I\in\binom{[n]}{d}} \left(\Delta_I(\mathbf{M})\right)^2 = 1 \tag{A.36}$$

The above follows from the Cauchy-Binet formula (see Appendix A.9.2). Therefore, we have a bound of 1 on the sum of squares of maximal minors. Since there are $\binom{n}{d}$ maximal minors, a lot of them will have to become very small in magnitude as we increase n while keeping d fixed, i.e. $d << n$. As we saw in a), the magnitude of all row vectors is equal. Therefore, the only way to have small maximal minors is to have

small angles between the vectors, which in turn forces many regions to be small (i.e. narrow wedges).

A.9.1 Derivation of a)

Recall a row $\mathbf{w} \in \mathbb{R}^{2k+1}$, $k \in \mathbb{N}$, $k \geq 1$ of the DFT matrix:

$$\mathbf{w} = \left[\frac{1}{\sqrt{n}} \; \sqrt{\frac{2}{n}}\cos t \; \sqrt{\frac{2}{n}}\sin t \; \cdots \; \sqrt{\frac{2}{n}}\cos kt \; \sqrt{\frac{2}{n}}\sin kt\right] \tag{A.37}$$

Using Eq. (A.37), we compute the Euclidean norm:

$$\|\mathbf{w}\|_2 = \sqrt{\sum_{j=1}^{2k+1} (w_j)^2} \tag{A.38}$$

$$= \sqrt{(w_1)^2 + \sum_{j=1}^{2k} (w_{j+1})^2} \tag{A.39}$$

$$= \sqrt{\left(\frac{1}{\sqrt{n}}\right)^2 + \sum_{k'=1}^{k} \left(\sqrt{\frac{2}{n}}\right)^2 \left((\sin k't)^2 + (\cos k't)^2\right)} \tag{A.40}$$

$$= \sqrt{\frac{1}{n} + \frac{2}{n}\sum_{k'=1}^{k} 1} \tag{A.41}$$

$$= \sqrt{\frac{2k+1}{n}} \tag{A.42}$$

A.9.2 Derivation of b)

The Cauchy-Binet formula (Pinkus, 2009, page 2) expresses the determinant of a product of two rectangular matrices $\det(\mathbf{AB})$ with $\mathbf{A} \in \mathbb{R}^{d\times n}$, $\mathbf{B} \in \mathbb{R}^{n\times d}$ in terms of a sum of maximal minors Δ of $\mathbf{A}$ and $\mathbf{B}$:

$$\sum_{I \in \binom{[n]}{d}} \Delta_I(\mathbf{A})\Delta_I(\mathbf{B}) = \det(\mathbf{AB}) \tag{A.43}$$

Note that for an orthonormal matrix $\mathbf{M} \in \mathbb{R}^{n\times d}$, if we set $\mathbf{A} = \mathbf{M}^\top$ and $\mathbf{B} = \mathbf{M}$, we get:

$$\begin{gathered}\sum_{I \in \binom{[n]}{d}} \Delta_I(\mathbf{M}^\top)\Delta_I(\mathbf{M}) = \det\left(\mathbf{M}^\top\mathbf{M}\right) \implies \\ \sum_{I \in \binom{[n]}{d}} \left(\Delta_I(\mathbf{M})\right)^2 = \det(\mathbf{I}) = 1\end{gathered} \tag{A.44}$$

where the bolded $\mathbf{I}$ is the identity matrix.

A.10 DFT Layer as FFT

We can use the FFT to speed up computations, as we will show that computing the logits $\mathbf{z}$ is equivalent to computing the truncated inverse DFT of the input $\mathbf{x}$, if we reinterpret the vector $\mathbf{x}$ that has $2k+1$ entries as the coefficients of $k+1$ complex numbers. Let us start from the inverse DFT, that computes the complex signal in the time domain from the frequency domain. We use n', d' and k' as variables to avoid confusion with n, d and k, which we have already defined as constants in the thesis. Denote the complex frequency component for frequency k' by $X_{k'}$ and the signal at time n' by $x_{n'}$, we have:

$$x_{n'} = \sum_{k'=0}^{n-1} X_{k'} \left[\cos\left(\frac{2\pi n'}{n} k'\right) + i \sin\left(\frac{2\pi n'}{n} k'\right) \right] \tag{A.45}$$

We take the real part of the iDFT, to obtain:

$$\begin{aligned} x_{n'} &= \sum_{k'=0}^{n-1} X_{k'} \left[\cos\left(k' t_{n'}\right) + i \sin\left(k' t_{n'}\right) \right] \\ &= \sum_{k'=0}^{n-1} \left(a_{k'} + i b_{k'}\right) \left[\cos\left(k' t_{n'}\right) + i \sin\left(k' t_{n'}\right) \right] \\ &= \sum_{k'=0}^{n-1} \left[a_{k'} \cos\left(k' t_{n'}\right) - b_{k'} \sin\left(k' t_{n'}\right) \right] \end{aligned}$$

If we truncate the iDFT to the first k frequencies, we get:

$$x_{n'} = \sum_{k'=0}^{k} \left[a_{k'} \cos\left(k' t_{n'}\right) - b_{k'} \sin\left(k' t_{n'}\right) \right]$$

We will now match the coefficients $a_{k'}$ and $b_{k'}$ to corresponding elements in $\mathbf{x}$. In the equations for simplicity we do not include scaling factors to make the columns orthonormal, see the code for details. From the earlier computation of $\mathbf{W}\mathbf{x}$, we rewrite the logits $\mathbf{z}$ as below:

$$z_{n'} = \mathbf{w}_{n'}^{\top} \mathbf{x} \tag{A.46}$$

$$= x_1 + \sum_{k'=1}^{k} \left[x_{2k'} \cos\left(k' t_n\right) + x_{2k'+1} \sin\left(k' t_n\right) \right] \tag{A.47}$$

From which we see that we can write the DFT layer as a truncated Inverse DFT by matching the coefficients of the sines and cosines: $x_1 = a_0$, $x_{2k'} = a_{k'}$ and $x_{2k'+1} = -b_{k'}$. See also our code test_dft_equivalence.py. From this perspective, this parametrisation is a low-pass filter.

Books

Aguiar, Marcelo and Swapneel Mahajan (2017). *Topics in Hyperplane Arrangements* (cit. on pp. 29, 69, 71).

Bertsekas, Dimitri P. and John N. Tsitsiklis (2008). *Introduction to Probability*. Athena Scientific (cit. on p. 86).

Björner, Anders et al. (1999). *Oriented Matroids*. 2nd ed. Encyclopedia of Mathematics and its Applications. Cambridge University Press. DOI: 10.1017/CBO9780511586507 (cit. on pp. 29, 41, 46, 57, 62, 73, 80, 192).

Boyd, Stephen P and Lieven Vandenberghe (2004). *Convex optimization*. Cambridge University Press (cit. on pp. 66, 128).

Gantmakher, F.R. and M.G. Kreĭn (1961). *Oscillation Matrices and Kernels and Small Vibrations of Mechanical Systems*. AMS/Chelsea Publication Series. USAEC Office of Technical Information. ISBN: 9780821882412. URL: https://books.google.co.uk/books?id=bI28m--C82wC (cit. on p. 155).

Hastie, Trevor, Robert Tibshirani, and Martin Wainwright (2015). *Statistical Learning with Sparsity: The Lasso and Generalizations*. Chapman & Hall/CRC. ISBN: 1498712169 (cit. on p. 175).

MacKay, David J. C. (2003). *Information Theory, Inference, and Learning Algorithms*. Copyright Cambridge University Press (cit. on pp. 76, 77).

Strang, Gilbert (Aug. 2016). *Introduction to Linear Algebra*. 5th ed. Wellesley, MA: Wellesley-Cambridge Press (cit. on pp. 48, 62).

Vapnik, Vladimir (1998). *Statistical learning theory*. Wiley, pp. I–XXIV, 1–736. ISBN: 978-0-471-03003-4 (cit. on p. 171).

Zaslavsky, Thomas (1975). *Facing up to arrangements: Face-count formulas for partitions of space by hyperplanes*. English. Vol. 154. Mem. Am. Math. Soc. Providence, RI: American Mathematical Society (AMS). DOI: 10.1090/memo/0154 (cit. on p. 76).

Ziegler, Günter M. (1995). *Lectures on Polytopes: Updated Seventh Printing of the First Edition*. New York, NY: Springer New York, pp. 149–190. ISBN: 978-1-4613-8431-1. DOI: 10.1007/978-1-4613-8431-1_6. URL: https://doi.org/10.1007/978-1-4613-8431-1_6 (cit. on pp. 46, 59, 64, 65, 77, 81, 110, 190, 192).

Articles

Aghakhani, Hojjat et al. (2021). "Bullseye polytope: A scalable clean-label poisoning attack with improved transferability". In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, pp. 159–178 (cit. on pp. 37, 39, 40).

Ahmed, Kareem et al. (2022). "Semantic Probabilistic Layers for Neuro-Symbolic Learning". In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., pp. 29944–29959. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/c182ec594f38926b7fcb827635b9a8f4-Paper-Conference.pdf (cit. on p. 83).

Ardila-Mantilla, Federico (2020). "CAT(0) Geometry, Robots, and Society". In: *Notices of the American Mathematical Society* 67, p. 1 (cit. on p. 106).

Babbar, Rohit and Bernhard Schölkopf (2019). "Data scarcity, robustness and extreme multi-label classification". In: *Machine Learning*, pp. 1–23 (cit. on p. 150).

Baruch, Emanuel Ben et al. (2020). "Asymmetric Loss For Multi-Label Classification". In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 82–91. URL: https://api.semanticscholar.org/CorpusID:221995872 (cit. on pp. 97, 127, 128, 133, 188).

Bhojanapalli, Srinadh et al. (July 2020). "Low-Rank Bottleneck in Multi-head Attention Models". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, pp. 864–873. URL: https://proceedings.mlr.press/v119/bhojanapalli20a.html (cit. on p. 101).

Blondel, Mathieu, Andre Martins, and Vlad Niculae (Apr. 2019). "Learning Classifiers with Fenchel-Young Losses: Generalized Entropies, Margins, and Algorithms". In: *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*. Ed. by Kamalika Chaudhuri and Masashi Sugiyama. Vol. 89.

Proceedings of Machine Learning Research. PMLR, pp. 606–615. URL: https://proceedings.mlr.press/v89/blondel19a.html (cit. on pp. 91, 96).

Brody, Shaked, Uri Alon, and Eran Yahav (July 2023). "On the Expressivity Role of LayerNorm in Transformers' Attention". In: *Findings of the Association for Computational Linguistics: ACL 2023*. Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, pp. 14211–14221. DOI: 10.18653/v1/2023.findings-acl.895. URL: https://aclanthology.org/2023.findings-acl.895 (cit. on p. 43).

Carlini, Nicholas et al. (2024). "Stealing Part of a Production Language Model". In: URL: https://api.semanticscholar.org/CorpusID:268357903 (cit. on pp. 34, 99).

Chen, Wenlin, David Grangier, and Michael Auli (Aug. 2016). "Strategies for Training Large Vocabulary Neural Language Models". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, pp. 1975–1985. DOI: 10.18653/v1/P16-1186. URL: https://aclanthology.org/P16-1186 (cit. on p. 143).

Chiang, Ting-Rui et al. (June 2024). "On Retrieval Augmentation and the Limitations of Language Model Training". In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*. Ed. by Kevin Duh, Helena Gomez, and Steven Bethard. Mexico City, Mexico: Association for Computational Linguistics, pp. 229–238. DOI: 10.18653/v1/2024.naacl-short.21. URL: https://aclanthology.org/2024.naacl-short.21 (cit. on p. 174).

Choi, Eunsol et al. (July 2018). "Ultra-Fine Entity Typing". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, pp. 87–96. DOI: 10.18653/v1/P18-1009. URL: https://aclanthology.org/P18-1009 (cit. on pp. 97, 127).

Conneau, Alexis et al. (July 2020). "Unsupervised Cross-lingual Representation Learning at Scale". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, pp. 8440–8451. DOI: 10.18653/v1/2020.acl-main.747. URL: https://aclanthology.org/2020.acl-main.747 (cit. on p. 140).

Cordovil, Raul and Pierre Duchet (2000). "Cyclic Polytopes and Oriented Matroids". In: *Eur. J. Comb.* 21, pp. 49–64 (cit. on pp. 189, 192).

Cover, Thomas M. (1965). "Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition". In: *IEEE Trans. Electron. Comput.* 14, pp. 326–334 (cit. on pp. 34, 39, 54, 76, 77, 152).

— (1967). "The Number of Linearly Inducible Orderings of Points in d-Space". In: *SIAM Journal on Applied Mathematics* 15.2, pp. 434–439. ISSN: 00361399. URL: http://www.jstor.org/stable/2946294 (visited on 11/08/2023) (cit. on pp. 34, 39, 54, 76, 177).

Dantzig, George B., Alex Orden, and Philip Wolfe (1954). "Notes on Linear Programming: Part 1. The Generalized Simplex Method for Minimizing a Linear Form under Linear Inequality Restraints". In: URL: https://api.semanticscholar.org/CorpusID:117041661 (cit. on pp. 109, 113).

Demeter, David, Gregory Kimmel, and Doug Downey (July 2020). "Stolen Probability: A Structural Weakness of Neural Language Models". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, pp. 2191–2197. DOI: 10.18653/v1/2020.acl-main.198. URL: https://www.aclweb.org/anthology/2020.acl-main.198 (cit. on pp. iii, 34, 39, 102, 127, 135, 166).

Devlin, Jacob et al. (June 2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423 (cit. on pp. 35, 90, 99, 133, 140).

Donoho, David L. (2006). "Compressed sensing". In: *IEEE Transactions on Information Theory* 52, pp. 1289–1306. URL: https://api.semanticscholar.org/CorpusID:261076528 (cit. on p. 175).

Donoho, David L. and Jared Tanner (2005). "Sparse nonnegative solution of underdetermined linear equations by linear programming." In: *Proceedings of the National Academy of Sciences of the United States of America* 102 27, pp. 9446–51. URL: https://api.semanticscholar.org/CorpusID:15655879 (cit. on pp. 175, 190).

— (2006). "Counting faces of randomly-projected polytopes when the projection radically lowers dimension". In: URL: https://api.semanticscholar.org/CorpusID:3350404 (cit. on pp. 147, 175, 191).

Doolittle, Joseph et al. (2020). "Combinatorial Inscribability Obstructions for Higher Dimensional Polytopes". In: *Mathematika* 66.4, pp. 927–953. DOI: https://doi.

org/10.1112/mtk.12051. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/mtk.12051. URL: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/mtk.12051 (cit. on p. 166).

Edin, Joakim et al. (2023). "Automated Medical Coding on MIMIC-III and MIMIC-IV: A Critical Review and Replicability Study". In: *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*. URL: https://api.semanticscholar.org/CorpusID:258291762 (cit. on p. 133).

Eppstein, David (1996). "Zonohedra and Zonotopes". In: *Mathematica in Education and Research*. Vol. 5, pp. 15–21 (cit. on p. 65).

Fan, Angela, Mike Lewis, and Yann Dauphin (July 2018). "Hierarchical Neural Story Generation". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Iryna Gurevych and Yusuke Miyao. Melbourne, Australia: Association for Computational Linguistics, pp. 889–898. DOI: 10.18653/v1/P18-1082. URL: https://aclanthology.org/P18-1082 (cit. on p. 102).

Finlayson, Matthew, Swabha Swayamdipta, and Xiang Ren (2024). "Logits of API-Protected LLMs Leak Proprietary Information". In: URL: https://api.semanticscholar.org/CorpusID:268384910 (cit. on pp. 34, 99).

Finlayson, Matthew et al. (2023). *Closing the Curious Case of Neural Text Degeneration*. arXiv: 2310.01693 [cs.CL] (cit. on p. 102).

Gale, David (1963). "Neighborly and cyclic polytopes". In: *Proceedings of Symposia in Pure Mathematics*. Vol. 7, pp. 225–232. DOI: https://doi.org/10.1090/pspum/007 (cit. on pp. 190, 192).

Ganea, Octavian et al. (2019). "Breaking the Softmax Bottleneck via Learnable Monotonic Pointwise Non-linearities". In: *ICML*, pp. 2073–2082. URL: http://proceedings.mlr.press/v97/ganea19a.html (cit. on pp. 102, 127).

Geiping, Jonas et al. (2024). *Coercing LLMs to do and reveal (almost) anything*. arXiv: 2402.14020 [cs.LG] (cit. on p. 37).

Giunchiglia, Eleonora, Mihaela Catalina Stoian, and Thomas Lukasiewicz (July 2022). "Deep Learning with Logical Constraints". In: *Proceedings of the 31st International Joint Conference on Artificial Intelligence and the 25th European Conference on Artificial Intelligence, IJCAI-ECAI 2022, Survey Track, Vienna, Austria, July 23–29, 2022*. Ed. by Luc De Raedt. IJCAI/AAAI Press, pp. 5478–5485. URL: https://doi.org/10.24963/ijcai.2022/767 (cit. on p. 83).

Goldberg, Yoav (2015). *A Primer on Neural Network Models for Natural Language Processing*. arXiv: 1510.00726 [cs.CL] (cit. on p. 89).

Good, I.J and T.N Tideman (1977). "Stirling numbers and a geometric structure from voting theory". In: *Journal of Combinatorial Theory, Series A* 23.1, pp. 34–45. ISSN: 0097-3165. DOI: https://doi.org/10.1016/0097-3165(77)90077-2. URL: https://www.sciencedirect.com/science/article/pii/0097316577900772 (cit. on p. 177).

Grivas, Andreas, Nikolay Bogoychev, and Adam Lopez (May 2022). "Low-Rank Softmax Can Have Unargmaxable Classes in Theory but Rarely in Practice". In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Dublin, Ireland: Association for Computational Linguistics, pp. 6738–6758. DOI: 10.18653/v1/2022.acl-long.465. URL: https://aclanthology.org/2022.acl-long.465 (cit. on pp. 42, 127).

Grivas, Andreas, Antonio Vergari, and Adam Lopez (Mar. 2024). "Taming the Sigmoid Bottleneck: Provably Argmaxable Sparse Multi-Label Classification". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 38.11, pp. 12208–12216. DOI: 10.1609/aaai.v38i11.29110. URL: https://ojs.aaai.org/index.php/AAAI/article/view/29110 (cit. on p. 42).

Groeneveld, Dirk et al. (2024). "OLMo: Accelerating the Science of Language Models". In: *Preprint* (cit. on pp. 34, 99).

Gu, Yu et al. (Oct. 2021). "Domain-Specific Language Model Pretraining for Biomedical Natural Language Processing". In: *ACM Trans. Comput. Healthcare* 3.1. DOI: 10.1145/3458754. URL: https://doi.org/10.1145/3458754 (cit. on p. 133).

Guo, Chuan et al. (Aug. 2017). "On Calibration of Modern Neural Networks". In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, pp. 1321–1330. URL: https://proceedings.mlr.press/v70/guo17a.html (cit. on p. 102).

Gurobi Optimization (2021). *Gurobi Optimizer Reference Manual*. URL: https://www.gurobi.com (cit. on pp. 129, 140).

Harris, Charles R. et al. (Sept. 2020). "Array programming with NumPy". In: *Nature* 585.7825, pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2 (cit. on p. 140).

He, Kaiming et al. (2016). "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778. DOI: 10.1109/CVPR.2016.90 (cit. on p. 90).

Hess, S., W. Duivesteijn, and D. Mocanu (2020). "Softmax-based Classification is k-means Clustering: Formal Proof, Consequences for Adversarial Attacks, and Improvement through Centroid Based Tailoring". In: *ArXiv* abs/2001.01987 (cit. on p. 72).

Hewitt, John, Christopher Manning, and Percy Liang (Dec. 2022). "Truncation Sampling as Language Model Desmoothing". In: *Findings of the Association for Computational Linguistics: EMNLP 2022*. Ed. by Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, pp. 3414–3427. DOI: 10.18653/v1/2022.findings-emnlp.249. URL: https://aclanthology.org/2022.findings-emnlp.249 (cit. on p. 102).

Hinton, Geoffrey E and Richard Zemel (1994). "Autoencoders, Minimum Description Length and Helmholtz Free Energy". In: *Advances in Neural Information Processing Systems*. Ed. by J. Cowan, G. Tesauro, and J. Alspector. Vol. 6. Morgan-Kaufmann (cit. on p. 101).

Holtzman, Ari et al. (2020). "The Curious Case of Neural Text Degeneration". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=rygGQyrFvH (cit. on p. 102).

Hooker, Sara (Apr. 2021). "Moving beyond "algorithmic bias is a data problem"". eng. In: *Patterns (New York, N.Y.)* 2.4. S2666-3899(21)00061-1[PII], pp. 100241–100241. ISSN: 2666-3899. DOI: 10.1016/j.patter.2021.100241. URL: https://doi.org/10.1016/j.patter.2021.100241 (cit. on p. 39).

Horn, Grant Van and Pietro Perona (2017). "The Devil is in the Tails: Fine-grained Classification in the Wild". In: *CoRR* abs/1709.01450. arXiv: 1709.01450. URL: http://arxiv.org/abs/1709.01450 (cit. on p. 102).

Householder, Alston S (1958). "Unitary triangularization of a nonsymmetric matrix". In: *Journal of the ACM (JACM)* 5.4, pp. 339–342 (cit. on p. 139).

Jain, Himanshu et al. (2019). "Slice: Scalable Linear Extreme Classifiers Trained on 100 Million Labels for Related Searches". In: *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining* (cit. on pp. 150, 151).

Jiang, Zhengbao et al. (2021). "How Can We Know When Language Models Know? On the Calibration of Language Models for Question Answering". In: *Transactions of the Association for Computational Linguistics* 9. Ed. by Brian Roark and Ani

Nenkova, pp. 962–977. DOI: 10.1162/tacl_a_00407. URL: https://aclanthology.org/2021.tacl-1.57 (cit. on p. 102).

Johnson, Alistair E W et al. (May 2016). "MIMIC-III, a freely accessible critical care database". en. In: *Sci. Data* 3.1, p. 160035 (cit. on pp. 132, 153, 187).

Junczys-Dowmunt, Marcin et al. (July 2018). "Marian: Fast Neural Machine Translation in C++". In: *Proceedings of ACL 2018, System Demonstrations*. Melbourne, Australia: Association for Computational Linguistics, pp. 116–121. DOI: 10.18653/v1/P18-4020. URL: https://aclanthology.org/P18-4020 (cit. on p. 141).

Kalman, Dan (1984). "The Generalized Vandermonde Matrix". In: *Mathematics Magazine* 57.1, pp. 15–21. ISSN: 0025570X, 19300980. URL: http://www.jstor.org/stable/2690290 (visited on 11/12/2024) (cit. on p. 57).

Kamiya, Hidehiko and Akimichi Takemura (2005). "Characterization of rankings generated by linear discriminant analysis". In: *Journal of multivariate analysis* 92.2, pp. 343–358 (cit. on p. 177).

Kang, Bingyi et al. (2020). "Decoupling Representation and Classifier for Long-Tailed Recognition". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=r1gRTCVFvB (cit. on pp. 102, 143).

Karp, Steven N. (2017). "Sign variation, the Grassmannian, and total positivity". In: *Journal of Combinatorial Theory, Series A* 145, pp. 308–339. ISSN: 0097-3165. DOI: https://doi.org/10.1016/j.jcta.2016.08.003. URL: https://www.sciencedirect.com/science/article/pii/S0097316516300759 (cit. on pp. 58, 155).

Kautz, Henry A., Ashish Sabharwal, and Bart Selman (2009). "Incomplete Algorithms". In: *Handbook of Satisfiability* (cit. on p. 138).

Khandelwal, Urvashi et al. (2020). "Generalization through Memorization: Nearest Neighbor Language Models". In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. URL: https://openreview.net/forum?id=HklBjCEKvH (cit. on p. 174).

Kim, Yoon and Alexander M. Rush (Nov. 2016a). "Sequence-Level Knowledge Distillation". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Ed. by Jian Su, Kevin Duh, and Xavier Carreras. Austin, Texas: Association for Computational Linguistics, pp. 1317–1327. DOI: 10.18653/v1/D16-1139. URL: https://aclanthology.org/D16-1139 (cit. on p. 101).

— (Nov. 2016b). "Sequence-Level Knowledge Distillation". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas:

Association for Computational Linguistics, pp. 1317–1327. DOI: 10.18653/v1/D16-1139. URL: https://www.aclweb.org/anthology/D16-1139 (cit. on p. 143).

Krieken, Emile van et al. (2024). "On the Independence Assumption in Neurosymbolic Learning". In: *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. URL: https://openreview.net/forum?id=S1gSrruVd4 (cit. on p. 83).

Krithara, Anastasia et al. (2023). "The road from manual to automatic semantic indexing of biomedical literature: a 10 years journey". In: *Frontiers in Research Metrics and Analytics* 8. ISSN: 2504-0537. DOI: 10.3389/frma.2023.1250930. URL: https://www.frontiersin.org/articles/10.3389/frma.2023.1250930 (cit. on p. 133).

Kuian, Mykhailo, Lothar Reichel, and Sergij V. Shiyanovskii (2019). "Optimally Conditioned Vandermonde-Like Matrices". In: *SIAM J. Matrix Anal. Appl.* 40, pp. 1399–1424 (cit. on p. 103).

Kulmanov, Maxat and R. Hoehndorf (2019). "DeepGOPlus: improved protein function prediction from sequence". In: *Bioinformatics* 36, pp. 422–429. URL: https://api.semanticscholar.org/CorpusID:146065857 (cit. on pp. 97, 127, 128).

Kuznetsova, Alina et al. (2020). "The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale". In: *IJCV* (cit. on pp. 133, 188).

Labeau, Matthieu and Shay B. Cohen (Nov. 2019). "Experimenting with Power Divergences for Language Modeling". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, pp. 4104–4114. DOI: 10.18653/v1/D19-1421. URL: https://aclanthology.org/D19-1421 (cit. on p. 143).

Land, Sander and Max Bartolo (2024). *Fishing for Magikarp: Automatically Detecting Under-trained Tokens in Large Language Models*. arXiv: 2405.05417 [cs.CL] (cit. on p. 37).

Liu, Frederick et al. (2021). "EncT5: A Framework for Fine-tuning T5 as Non-autoregressive Models". In: *arXiv preprint arXiv:2305.05627* (cit. on p. 140).

Liu, Y. et al. (2019). "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *ArXiv* abs/1907.11692 (cit. on p. 140).

Manecke, Sebastian and Raman Sanyal (2022). *Inscribable Fans II: Inscribed zonotopes, simplicial arrangements, and reflection groups*. arXiv: 2203.11062 [math.MG] (cit. on p. 166).

Manhaeve, Robin et al. (2018). "DeepProbLog: Neural Probabilistic Logic Programming". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/dc5d637ed5e62c36ecb73b654b05ba2a-Paper.pdf (cit. on p. 83).

McInnes, Leland et al. (2018). "UMAP: Uniform Manifold Approximation and Projection". In: *The Journal of Open Source Software* 3.29, p. 861 (cit. on pp. 101, 102).

Menon, Aditya Krishna, Ankit Singh Rawat, and Sanjiv Kumar (2021a). "Overparameterisation and worst-case generalisation: friend or foe?" In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=jphnJNOwe36 (cit. on p. 39).

Menon, Aditya Krishna et al. (2021b). "Long-tail learning via logit adjustment". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=37nvvqkCo5 (cit. on p. 102).

Mikolov, Tomas et al. (2013). "Efficient Estimation of Word Representations in Vector Space". In: *ICLR* (cit. on p. 101).

Mnev, Nikolai (1988). "The universality theorems on the classification problem of configuration varieties and convex polytopes varieties". In: (cit. on p. 83).

Mullenbach, J. et al. (2018). "Explainable Prediction of Medical Codes from Clinical Text". In: *NAACL* (cit. on pp. 97, 99, 127, 128, 132, 153, 161, 187).

Nentidis, Anastasios et al. (2021). "Overview of BioASQ Tasks 9a, 9b and Synergy in CLEF2021". In: *Conference and Labs of the Evaluation Forum* (cit. on pp. 133, 188).

Ng, Nathan et al. (Aug. 2019). "Facebook FAIR's WMT19 News Translation Task Submission". In: *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*. Florence, Italy: Association for Computational Linguistics, pp. 314–319. DOI: 10.18653/v1/W19-5333. URL: https://aclanthology.org/W19-5333 (cit. on p. 142).

Nguyen, Toan Q. and David Chiang (2017). "Improving Lexical Choice in Neural Machine Translation". In: *CoRR* abs/1710.01329. arXiv: 1710.01329. URL: http://arxiv.org/abs/1710.01329 (cit. on p. 143).

Papadimitriou, Dimitris and Swayambhoo Jain (2021). "Data-Driven Low-Rank Neural Network Compression". In: *2021 IEEE International Conference on Image Processing (ICIP)*, pp. 3547–3551. DOI: 10.1109/ICIP42928.2021.9506467 (cit. on p. 101).

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830 (cit. on pp. 127, 128).

Postnikov, Alexander (2006). "Total positivity, Grassmannians, and networks". In: *arXiv: Combinatorics*. URL: https://api.semanticscholar.org/CorpusID:118924135 (cit. on p. 58).

Radford, Alec et al. (2019). "Language Models are Unsupervised Multitask Learners". In: (cit. on p. 140).

Raunak, Vikas et al. (Nov. 2020). "On Long-Tailed Phenomena in Neural Machine Translation". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, pp. 3088–3095. DOI: 10.18653/v1/2020.findings-emnlp.276. URL: https://www.aclweb.org/anthology/2020.findings-emnlp.276 (cit. on p. 143).

Ridnik, T. et al. (2020). "TResNet: High Performance GPU-Dedicated Architecture". In: *2021 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 1399–1408. URL: https://api.semanticscholar.org/CorpusID:214714440 (cit. on p. 133).

Rogers, Anna and Alexandra Sasha Luccioni (2024). *Position: Key Claims in LLM Research Have a Long Tail of Footnotes*. arXiv: 2308.07120 [cs.CL] (cit. on p. 33).

Sainath, Tara N. et al. (2013). "Low-rank matrix factorization for Deep Neural Network training with high-dimensional output targets". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 6655–6659. DOI: 10.1109/ICASSP.2013.6638949 (cit. on p. 101).

Savostianova, Dayana et al. (2023). "Robust low-rank training via approximate orthonormal constraints". In: *Advances in Neural Information Processing Systems (NeurIPS)* (cit. on p. 101).

Schultheis, Erik and Rohit Babbar (Nov. 2022). "Speeding-up one-versus-all training for extreme classification via mean-separating initialization". In: *Machine Learning* 111.11, pp. 3953–3976. ISSN: 1573-0565. DOI: 10.1007/s10994-022-06228-2. URL: https://doi.org/10.1007/s10994-022-06228-2 (cit. on p. 161).

Sennrich, Rico, Barry Haddow, and Alexandra Birch (Aug. 2016). "Neural Machine Translation of Rare Words with Subword Units". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, pp. 1715–1725. DOI: 10.18653/v1/P16-1162. URL: https://aclanthology.org/P16-1162 (cit. on p. 141).

Sennrich, Rico et al. (Sept. 2017). "The University of Edinburgh's Neural MT Systems for WMT17". In: *Proceedings of the Second Conference on Machine Translation*. Copenhagen, Denmark: Association for Computational Linguistics, pp. 389–399.

DOI: 10.18653/v1/W17-4739. URL: https://aclanthology.org/W17-4739 (cit. on p. 142).

Shor, Peter W. (1990). "Stretchability of Pseudolines is NP-Hard". In: *Applied Geometry And Discrete Mathematics* (cit. on p. 83).

Stanley, Richard P. (2004). "An Introduction to Hyperplane Arrangements". In: *Lecture notes, IAS/Park City Mathematics Institute* (cit. on pp. 46, 77).

Steedman, Mark (Dec. 2018). "The Lost Combinator". In: *Computational Linguistics* 44.4, pp. 613–629. DOI: 10.1162/coli_a_00328. URL: https://aclanthology.org/J18-4001 (cit. on p. 171).

Sturmfels, Bernd (1988). "Neighborly Polytopes and Oriented Matroids". In: *European Journal of Combinatorics* 9.6, pp. 537–546. ISSN: 0195-6698. DOI: https://doi.org/10.1016/S0195-6698(88)80050-7. URL: https://www.sciencedirect.com/science/article/pii/S0195669888800507 (cit. on p. 192).

Tiedemann, Jörg (Nov. 2020). "The Tatoeba Translation Challenge – Realistic Data Sets for Low Resource and Multilingual MT". In: *Proceedings of the Fifth Conference on Machine Translation*. Online: Association for Computational Linguistics, pp. 1174–1182. URL: https://www.aclweb.org/anthology/2020.wmt-1.139 (cit. on pp. 37, 187).

Tiedemann, Jörg and Santhosh Thottingal (Nov. 2020). "OPUS-MT – Building open translation services for the World". In: *Proceedings of the 22nd Annual Conference of the European Association for Machine Translation*. Lisboa, Portugal: European Association for Machine Translation, pp. 479–480. URL: https://aclanthology.org/2020.eamt-1.61 (cit. on p. 140).

Touvron, Hugo et al. (2023). "LLaMA: Open and Efficient Foundation Language Models". In: *ArXiv* abs/2302.13971 (cit. on pp. 34, 99).

Tsatsaronis, George et al. (2015). "An overview of the BIOASQ large-scale biomedical semantic indexing and question answering competition". In: *BMC Bioinformatics* 16 (cit. on pp. 133, 151, 188).

Vapnik, V. N. and A. Ya. Chervonenkis (1971). "On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities". In: *Theory of Probability & Its Applications* 16.2, pp. 264–280. DOI: 10.1137/1116025. eprint: https://doi.org/10.1137/1116025. URL: https://doi.org/10.1137/1116025 (cit. on p. 152).

Wainwright, Martin J. (2009). "Sharp Thresholds for High-Dimensional and Noisy Sparsity Recovery Using ℓ_{1} -Constrained Quadratic Programming (Lasso)".

In: *IEEE Transactions on Information Theory* 55, pp. 2183–2202. URL: https://api.semanticscholar.org/CorpusID:11046646 (cit. on p. 175).

Wolf, Thomas et al. (Oct. 2020). "Transformers: State-of-the-Art Natural Language Processing". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, pp. 38–45. URL: https://www.aclweb.org/anthology/2020.emnlp-demos.6 (cit. on p. 140).

Yang, Zhilin et al. (2018). "Breaking the Softmax Bottleneck: A High-Rank RNN Language Model". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=HkwZSG-CZ (cit. on pp. iii, 34, 39, 99, 102, 103, 127).

Zhang, Liwen, Gregory Naitzat, and Lek-Heng Lim (Oct. 2018). "Tropical Geometry of Deep Neural Networks". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, pp. 5824–5832. URL: https://proceedings.mlr.press/v80/zhang18i.html (cit. on p. 77).

Zhang, Zhisong, Xuezhe Ma, and Eduard Hovy (July 2019). "An Empirical Investigation of Structured Output Modeling for Graph-based Neural Dependency Parsing". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, pp. 5592–5598. DOI: 10.18653/v1/P19-1562. URL: https://www.aclweb.org/anthology/P19-1562 (cit. on p. 76).

Zhu, Chen et al. (Sept. 2019). "Transferable Clean-Label Poisoning Attacks on Deep Neural Nets". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, pp. 7614–7623. URL: https://proceedings.mlr.press/v97/zhu19a.html (cit. on pp. 37, 39, 40).

Blog Posts

Paul, Sayak and Soumik Rakshit (2014). *Large-scale multi-label text classification*. [Online; accessed 15-August-2023] (cit. on p. 127).

Rumbelow, Jessica and Matthew Watkins (2023). *SolidGoldMagikarp (plus, prompt generation)*. [Online; accessed 13-May-2024] (cit. on pp. 37, 174).

Smith, Warren D. (2014). *D-dimensional orderings and Stirling numbers*. [Online; accessed 05-November-2021]. URL: https://rangevoting.org/WilsonOrder.html (cit. on p. 177).