THE UNIVERSITY
of EDINBURGH

Flexible Neural Architectures for Sequence Modeling

Ben Krause

Doctor of Philosophy
Institute for Language, Cognition and Computation
School of Informatics
University of Edinburgh
2019

Abstract

Auto-regressive sequence models can estimate the distribution of any type of sequential data. To study sequence models, we consider the problem of language modeling, which entails predicting probability distributions over sequences of text. This thesis improves on previous language modeling approaches by giving models additional flexibility to adapt to their inputs. In particular, we focus on multiplicative LSTM (mLSTM), which has added flexibility to change its recurrent transition function depending on its input as compared with traditional LSTM, and dynamic evaluation, which helps LSTM (or other sequence models) adapt to the recent sequence history to exploit re-occurring patterns within a sequence. We find that using these adaptive approaches for language modeling improves their predictions by helping them recover from surprising tokens and sequences.

mLSTM is a hybrid of a multiplicative recurrent neural network (mRNN) and an LSTM. mLSTM is characterized by its ability to have recurrent transition functions that can vary more for each possible input token, and makes better predictions as compared with LSTM after viewing unexpected inputs in our experiments. mLSTM also outperformed all previous neural architectures at character level language modeling.

Dynamic evaluation is a method for adapting sequence models to the recent sequence history at inference time using gradient descent, assigning higher probabilities to re-occurring sequential patterns. While dynamic evaluation was often previously viewed as a way of using additional training data, this thesis argues that dynamic evaluation is better thought of as a way of adapting probability distributions to their own predictions. We also explore and develop dynamic evaluation methods with the goals of achieving the best prediction performance and computational/memory efficiency, as well as understanding why these methods work. Different variants of dynamic evaluation are applied to a number of different architectures, resulting in improvements to language modeling over a longer contexts, as well as polyphonic music prediction. Dynamically evaluated models are also able to generate conditional samples that repeat patterns from the conditioning text, and achieve improved generalization in modeling out of domain sequences. The added flexibility that dynamic evaluation gives models allows them to recover faster when predicting unexpected sequences.

The proposed approaches improve on previous language models by giving them additional flexibility to adapt to their inputs. mLSTM and dynamic evaluation both contributed to improvements to the state of the art in language modeling, and have potential applications to a wider range of sequence modeling problems.

Acknowledgments

Many thanks to my supervisors Steve Renals and Iain Murray, my publication coauthors Liang Lu and Emmanuel Kahembwe, and my PhD reviewers Adam Lopez and Chris Dyer.

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.

(*Ben Krause*)

Table of Contents

Acronyms

AWD-LSTM Averaged SGD weight dropped LSTM

GPT Generative Pretrained Transformer

LSTM Long-short term Memory

MAML Model agnostic meta-learning

MIDI Musical Instrument Digital Interface

MI-LSTM Multiplicative integration LSTM

mLSTM Multiplicative LSTM

mRNN Multiplicative RNN

NADE Neural auto-regressive distribution estimator

NLP Natural language processing

PTB Penn Treebank

QRNN Quasi recurrent neural network

RMS Root mean-squared

RNN Recurrent neural network

SGD Stochastic Gradient Descent

Chapter 1

Introduction

Models that can use information from sequential data are of great interest in machine learning. Probabilistic sequence models can be used compress, generate, or estimate the log-likelihood of sequences. To study sequence modeling, this thesis considers the specific case of language modeling, which entails predicting a probability distribution over the next word or symbol, conditioned on the previous words or symbols. Starting by predicting the first word, and then predicting each next word conditioned on the previous words, language models can assign probabilities to sequences of text. Many sequence modeling problems can be posed as language modeling, including modeling of continuous sequences by discretizing values into bins, as is often done for audio generation (Oord et al., 2016). Although we mainly focus on textual language modeling, we develop methods that are general enough that they could, in theory, be applied to other sequence modeling problems outside of the text domain (as opposed to approaches that use linguistic features).

Language models in this thesis are mainly evaluated by the negative log-likelihood (or cross entropy) that the model assigns to text sequences held out from the training set. This metric directly measures how well a language model theoretically could compress text. Beyond compression capability, language models can also be evaluated by the quality of the text that they generate, however this is not the main focus of this thesis. While not guaranteed, language prediction ability often strongly relates to generation quality. For instance, deep learning based machine translation systems are usually trained as conditional language models with a cross entropy objective (Sutskever et al., 2014), even though they are evaluated by generation quality. Language models can also be used to re-score outputs of a speech recognition systems, often leading to better transcriptions (Mikolov et al., 2010). Further more, strong language models can

sometimes perform other text generation tasks without any additional training, including summarization, machine translation, and question answering (Radford et al., 2019). Models pretrained with language modeling and related objective functions can achieve strong results when finetuned to downstream text classification tasks (Radford et al., 2018; Devlin et al., 2018; Yang et al., 2019). Language modeling can also be used to study the ability of sequence modeling architectures in using long sequential contexts to help make predictions, which is a challenging problem in general sequence modeling.

Early approaches to language modeling, such as n-grams (Shannon, 1951; Brown et al., 1992a), used the simplifying assumption that only the recent sequence history matters, ignoring long range statistical dependencies in sequences. Recurrent neural networks (Elman, 1990, RNNs) addressed the limitations of previous approaches by using a hidden state to summarize the sequence history. An RNN processes a sequence of inputs, which could be words or characters for instance, one input at a time. An RNN's hidden state at given time step in a sequence is a function of the input at the current time step, and the hidden state at the previous time step. If the task is language modeling, the model then uses a function of the hidden state at the current timestep to predict the token at the next time step. By using a hidden state recursively in this way, RNNs can use inputs from farther in the past to help predict the next token, as compared with a model that uses a fixed context to make predictions. Early approaches to RNNs proved difficult to train (Robinson, 1994), but improvements to RNNs such as the long short-term memory (Hochreiter and Schmidhuber, 1997, LSTM) architecture allowed them to achieve widespread success at sequence modeling. Long-short term memory uses a series of gates to preserve information from previous hidden states better than a traditional RNN. LSTMs have achieved widespread success, outperforming previous approaches at language modeling (Zaremba et al., 2014), speech recognition (Graves et al., 2013), and handwriting generation (Graves, 2013), among other tasks.

Concurrently with this thesis, the more recently proposed Transformer (Vaswani et al., 2017) architecture has demonstrated an even stronger ability to use long contexts to help make predictions. Like an RNN, a Transformer has a hidden state associated with every time step in the input sequence. Unlike an RNN, a Transformer continues to store and use all previous hidden states within some fixed context window as it processes a sequence. It uses a mechanism called "self-attention" that allows it to look up previous hidden states within that context window. The ability to go back and remember previous hidden states makes the Transformer especially strong for using large amounts of context to make predictions, and as a result, this architecture has

achieved strong results in language modeling (Radford et al., 2019) and other natural language processing tasks (Devlin et al., 2018). Due to the timeline of this thesis, the majority of experiments use LSTM as a baseline, as it was considered state-of-the-art in language modeling at the time that most of the work was carried out. However, some of the later experiments consider Transformers as a baseline to show that the principles in this thesis can still give improvements to stronger architectures.

In the language modeling setting considered in this thesis, language models are trained to minimize cross entropy on training data and evaluated on cross entropy in held out data. When making predictions on held out sequences, language models will generally encounter tokens and sequences that they were not expecting, and will fail to correctly predict. When such a situation occurs, the model must adapt, meaning it needs to reinterpret the current state as a function of the unexpected input, the sequence history, and the data it was trained on. Otherwise, it will continue to make wrong predictions on later parts of the sequence. To illustrate this, consider the following text, taken from the WikiText-2 corpus:

The Gambia won the first match 3 - 0 in Banjul , the Gambia 's capital . The return match was delayed in for 24 hours and played in Makeni. The Gambia beat Sierra Leone 4 - 3 to qualify for the final round. The Gambia then beat Tunisia 1 - 0 at home and won 2 - 1 in Tunisia .

The names of African countries are common words in the context of this sequence, but rare words in the context of the entire distribution. This sequence also repeatedly uses words that pertain to sports and competition. We refer to these repetitions as "re-occurring sequential patterns", a term we use to refer to both the direct repetition of words like "Gambia" and "Tunisia", as well as the repeated use of words that relate to sports and competition. When the model encounters the first instance of one of these patterns, it will likely struggle no matter what, since the patterns specific to this sequence are rare in the context of the training data. However, a model that correctly adapts to the early parts of the sequence should be able to better predict words that relate to African countries or competition later on in the sequence. It is evident that some language models struggle to adapt to their inputs; for instance, Grave et al. (2017b) found that augmenting an LSTM language model with a simple unigram cache could improve

its prediction performance. This indicates that adapting to the unigram distribution of recent text, while relatively simple in principle, is a challenging problem for LSTM language models. Studying approaches to make language models more adaptive is therefore well motivated for improving their predictions.

In this thesis, we improve language models by giving them added flexibility to adapt to their inputs. This is done by allowing models to make larger changes to their representation as a result of text they observe. We consider adaptation at two levels; adapting to individual tokens and adapting to sequences of many tokens. Models that are more adaptive at the token level are able to reinterpret their context and make large changes to their predictions as a result of different tokens they observe. For instance, an RNN that is able to make large changes to its hidden-to-hidden transition function, and thus large changes to its predictions, as a result of an input token, would be considered more adaptive at the token level. For adapting to longer sequences, a model may need to be able to make larger changes to its representation to fully adapt to the whole context. For instance, a model that is able to change all of its weights as a function of the sequence history would likely be more adaptive than a model that can only represent the sequence history with a hidden state vector of limited size.

For adapting to language at the token level, we consider the case of recurrent neural networks, where the hidden state is expressed as a function of the previous hidden state and the current input. The vanilla RNN is able to modify its hidden state somewhat as a function of each token it observes by having a different bias for each possible input token, given by the model's embedding matrix. However, a vanilla RNN cannot make very large changes to its hidden state without erasing information from the past, giving it a somewhat limited token level adaptation ability. Architectural enhancements in RNNs such as LSTM (Hochreiter and Schmidhuber, 1997) allow them to be more adaptive to their inputs by having gates, which partially depend on connections from the input, to control information flow in the network. This thesis considers multiplicative LSTM (mLSTM) to make LSTMs even more adaptive, by allowing them to effectively have a different recurrent transition matrix for each possible input.

In order to achieve broader adaptation at the level of entire sequences, we consider using dynamic evaluation to adapt models to the recent sequence history, a method first proposed by Mikolov et al. (2010) and greatly extended in this thesis. In spite of its name, which implies that it is an evaluation metric, this thesis argues that dynamic evaluation is actually an architectural enhancement that adapts sequence models to their own predictions. Dynamic evaluation uses gradient descent updates on the loss of

the recent sequence predictions to update the parameters of the model, thus giving the model the ability to adapt to its inputs.

The principle claim in this thesis is that **giving language models additional flexibility to adapt to their inputs can improve their predictions by helping them recover from surprising tokens or sequences**. Specifically, by adaptation, we mean the ability for a language model to modify its prediction distribution over future sequences as a result of its input, and by recovering from surprise, we mean the ability to accurately model tokens or continuing sequences immediately after a token or sequence that was not expected by the model. We show that multiplicative LSTM (mLSTM) language models, which are more adaptive at the token level, recover more quickly from surprising tokens than regular LSTMs. While mLSTMs have the ability to recover from a surprising token, they still struggle to adapt to surprising sequences. For instance, mLSTMs struggle when evaluated on sequences from a language that is different from the training language. We show that applying dynamic evaluation to mLSTM and other models (including Transformers), which allows them to be more adaptive at the sequence level, results in models that are more robust to surprising sequences. For instance, dynamic evaluation can give large improvements in the case when the testing sequence is in a different language from the training sequences. Dynamic evaluation is also often able to give large improvements in general language modeling due to its ability to adapt to re-occurring sequential patterns that are unique to every sequence.

The work on mLSTM is motivated by the concept of "flexible input dependent transitions" for RNNs, which means the ability to have a very different hidden to hidden transition function for each possible input token, thus giving them the ability to adapt to the token. We empirically show that mLSTM gives improvements in language modeling compared with LSTM and other architectures, and suggest its adaptability is the reason for this improvement. We also show that mLSTM's advantage over LSTM is larger after a surprising token than it is in general, suggesting that mLSTM's ability to adapt its hidden representation when encountering a surprising input at least partially explains its advantage.

We use dynamic evaluation to adapt language models to the sequence history. We show that dynamic evaluation can give large improvements to language modeling, which we hypothesize is due to its ability to adapt to unexpected re-occurring sequential patterns. Supporting this hypothesis is the ability of dynamic evaluation augmented models to predict repeating words, generate samples that repeat patterns in the conditioning text, and predict text that is in a different language from the training text. We also show that

dynamic evaluation augmented models are able to use very long contexts to improve their predictions, which is necessary for fully adapting to longer sequences. Lastly, we observe large performance gains applying dynamic evaluation to polyphonic musical note prediction, suggesting that adapting to surprising re-occurring sequential patterns may be useful for other modes of sequential data besides text.

The experiments in this thesis were performed during a time when the field was rapidly changing. Some of the LSTM baselines used in experiments early in this thesis that were state-of-the-art at the time were superseded by stronger Transformer-based methods used as baselines later in the thesis. However, studying sequence modeling in RNN based models is still of interest, since recurrence is a necessary feature for models to utilize longer than fixed length contexts using a fixed amount of memory. Our experiments show that dynamic evaluation can help improve both RNNs and Transformers, showing a greater generality of the method. Even as methods continue to improve, studying models with the flexibility to adapt to their inputs remains useful for interpreting recent architectural improvements as well as developing new ones.

Chapter 2

Background

This background chapter covers previous and concurrent approaches that are necessary to understand the adaptive methods considered in this thesis, as well as the context under which these methods were developed. First, we define the problem of language modeling, the main benchmark task used to evaluate our proposed methods, in Section 2.1. Section 2.2 considers previous approaches to make language models adaptive at the sequence level. Section 2.3 reviews RNNs, including vanilla RNNs, backpropagation through time, LSTMs. LSTMs are the backbone for many of the architectural modifications in the thesis, and back propagation through time is important for understanding dynamic evaluation. Next, Section 2.4 reviews other sequence modeling architectures that are important for understanding the context of this thesis, including architectural ideas that we build off of, and other architectures that have similar motivations to approaches in this thesis. Section 2.5 reviews optimizers that we use throughout this thesis, both for training models from scratch as well as for dynamic evaluation of trained models at test time. Lastly, Section 2.6 covers high dimensional sequence modeling, which is later used for polyphonic music prediction, a secondary sequence modeling task considered in this thesis.

2.1 Language modeling

Language modeling is the task of modeling a probability distribution over sequences of language. While there are many ways to model probability distributions, for language this is usually done with models that predict the next word or character of text, given the history of text. These models start by predicting the first word and iteratively predict the next word conditioned on the previous words in a sequence one at a time, which makes

it possible to model a probability distribution over all possible sequences of text allowed by the vocabulary. Traditionally, n-gram models that assume the Markov property in language were used for probabilistic text modeling (Shannon, 1951; Brown et al., 1992a). Neural network based models in place of n-gram models were considered by Bengio et al. (2003). Since then, RNNs have been applied to language modeling (Mikolov et al., 2010), allowing language models to use longer range statistical dependencies and avoid assuming the Markov property. Language modeling at the word and character level are the two main benchmarks considered in this thesis.

2.1.1 Auto-regressive sequence modeling

Language modeling and relies on an auto-regressive factorization to perform density estimation and generation of data. Auto-regressive sequence models assign a probability to a sequence $x_{1:T} = \{x_1, \ldots, x_T\}$ by factorizing it using the chain rule as

$$P(x_{1:T}) = P(x_1) \prod_{t=2}^{T} P(x_t | x_{1:t-1}). \tag{2.1}$$

Auto-regressive sequence modeling is illustrated in Figure 2.1. In auto-regressive sequence modeling, models condition on inputs after predicting them. When evaluating the probability of a sequence of text, the model conditions on the ground truth tokens of that sequence. Even though this means the model will view test time tokens, this is a valid way to measure probability, because the auto-regressive factorization ensures that the sum of probabilities assigned to all possible sequences will sum to 1. When sampling from the model, the model conditions on inputs that were generated by the model at the previous timesteps. Since auto-regressive models condition on the same tokens they predict, it is possible perform the adaptation methods that fit to the sequence history considered in Section 2.2 and throughout this thesis.

2.1.2 Language model evaluation

Language models can be trained using a cross entropy error objective, where the loss for each sequence element x_t at time step t occurring in the context of the sequence history up to t, $x_{1:t-1}$, is given by

$$-logP(x_t | x_{1:t-1}) \tag{2.2}$$

and the overall training loss is given by the average cross entropy error on the training set. In this thesis, language models are evaluated by applying the same cross

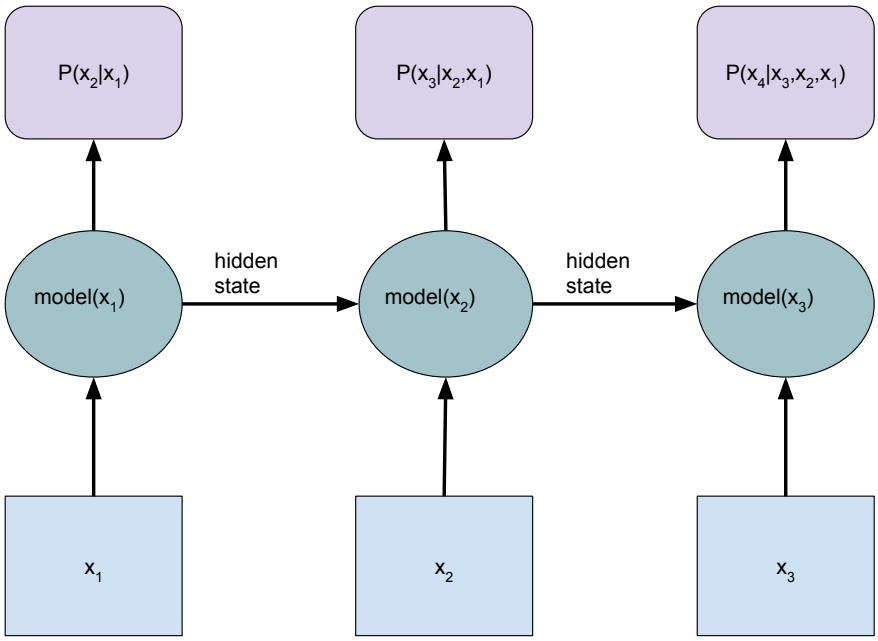


Figure 2.1: A neural auto-regressive sequence model. The model repeatedly predicts a probability distribution over the next symbol conditioned on all previous symbols, using a hidden state to summarize information about the past.

entropy loss on held out sequences. The sum on all the cross entropy errors on held out sequences is equivalent to the theoretical compression limit that the model could achieve encoding the test dataset (Shannon, 1948). In practice, it is possible to actually achieve compression to a number of bits within a small constant of the theoretical compression limit using a language model, for instance with arithmetic coding (Witten et al., 1987).

Language models can also be evaluated for their text generation ability. Language models can generate text by feeding tokens output by the model back into the model as inputs. It is possible to sample directly from the probability distribution given by the model in this way. To attempt to find the most likely sequences or conditional sequences under the model, greedy decoding (always outputting the most likely token) or beam search can be used. Conditional language models that are trained with a cross-entropy objective can be used for generation tasks such as summarization (Nallapati et al., 2016) or machine translation (Sutskever et al., 2014). Evaluating language models for generation is somewhat more difficult, and often uses imperfect n-gram matching based measures such as BLEU score (Papineni et al., 2002) or ROUGE score (Lin and Och, 2004), or expensive human evaluation. The main difference between evaluation by generation and evaluation by cross-entropy, is that when evaluating by cross entropy, the model only ever sees ground truth tokens as input, whereas when evaluating by generation quality, the model sees tokens that it output itself at previous timesteps. There is no guarantee that a strong cross entropy model will be a strong generation model or visa versa, but in practice the two are usually correlated. In this thesis, we consider almost exclusively cross-entropy evaluation.

2.1.3 Language model tokenization

In order to perform language modeling, text needs to be mapped to a sequence of tokens, through a process called “tokenization”. There are many different ways to tokenize text–two of the most straightforward ways considered in this thesis are as a sequence of characters, or as a sequence of words. Neural network based language models typically have a fixed vocabulary size, due to the need to have a fixed number of input and output units, where each input/output unit corresponds to a single word in the vocabulary. The choice of tokenization can restrict the range of sequences that a language model can assign a probability to. For instance, a word level language model with a predefined vocabulary cannot model words from outside of that vocabulary; it would have no way of predicting, for instance, a misspelled word or a word from another language. In

exchange for sacrificing this flexibility, word level language models have the advantage that each token generally corresponds to a unit of meaning, and sequences are shorter, making them faster to process and making it potentially easier to model statistical dependencies across larger gaps of text.

Character-level tokenization for language models gives them added flexibility to model any possible words that can be composed with the predefined character alphabet. A case insensitive character level tokenization for the English language might include a vocabulary of 27 for the letters a-z plus spaces. More characters can be included in the vocabulary to give the model even more flexibility to predict upper case, punctuation, or non-English characters. In order to achieve even greater flexibility, some experiments in this thesis use a UTF-8 byte tokenization, where text is modeled using a vocabulary of 256 possible bytes. Many types of text files in many languages are stored using UTF-8 bytes, So using this byte-level tokenization gives a language model the flexibility to model almost any kind of text sequence. Character level language modeling requires using longer contexts to make predictions as compared with word level language modeling, making it a challenging problem, but also an interesting benchmark for neural sequence modeling architectures.

For a trade off between word and character level tokenization, other kinds of subword tokenizations are sometimes used. In this thesis, we have experiments using a byte-pair encoding (Sennrich et al., 2015), which starts at the character or byte-level, and replaces common sub-sequences of tokens with a single new token, and does this iteratively to reduce the length of text datasets to a smaller number of tokens. Subword tokenizations allow tokens to be more meaningful than individual characters, while also allowing the flexibility to model out of vocabulary words.

Typically, language models are compared with each other using the same tokenization. The average negative log-likelihood per token (which can be exponentiated to get perplexity, which is also sometimes used) can be used to compare the prediction ability two language models that use the same tokenization. Two language models with different tokenizations can be comparable if the tokenization is invertable, meaning that the original text can be recovered exactly from the tokenized text. In this case, both models can be used to assign negative log-likelihoods to the entire dataset of text by summing the negative log-likelihoods of each prediction under each model's respective tokenization. The negative log-likelihood of the dataset is then a comparable metric of how well each model would be able to compress that dataset.

2.2 Adaptive language modeling

As this thesis focuses on making language models more adaptive, we describe previous approaches that adapt language models to the sequence history. Adaptive language modeling was first considered for n-grams, adapting to recent history via caching (Kuhn, 1988; Jelinek et al., 1991; Kuhn and De Mori, 1992), and other methods (Bellegarda, 2004). Since, methods aimed at adapting neural language models have been developed, such as the neural cache (Grave et al., 2017b), the closely related pointer-sentinel RNN (Merity et al., 2017), and dynamic evaluation (Mikolov et al., 2010).

2.2.1 Neural cache

The neural cache model learns a non-parametric output layer on the fly at test time, enabling the network to adapt to recent observations. Each past hidden state h_i is paired with the next input x_{i+1}, and stored as a tuple (h_i, x_{i+1}). When a new hidden state h_t is observed, the output probabilities are adjusted to give a higher weight to words that coincide with past hidden states, with a large inner product $(h_t^\top h_i)$:

$$P_{\text{cache}}(x_{t+1}|x_{1:t}, h_{1:t}) \propto \sum_{i=1}^{t-1} e_{(x_{i+1})} \exp(\omega h_t^\top h_i), \tag{2.3}$$

where $e_{(x_{i+1})}$ is a one hot encoding of token x_{i+1} (meaning the resulting vector has a value of 1 at the index of the token, and 0 for the rest of the vocabulary), and ω is a scale parameter. Since $P_{\text{cache}}(x_{t+1}|x_{1:t}, h_{1:t})$ is composed of a weighted sum of one-hot vector encoding of previously occurring tokens, $P_{\text{cache}}(x_{t+1}|x_{1:t}, h_{1:t}) = 0$ for all tokens that have not previously occurred in the sequence. The neural cache could also be thought of as a type of attention (see Section 2.4.5.1), where the query is the current hidden state h_t, the keys are the previous hidden states, and the values are the one hot encodings of the labels. Test time adaptation is carried out by interpolating the cache probabilities with the base network probabilities.

Another closely related method, the pointer-sentinel RNN (Merity et al., 2018b), uses a similar non-parametric adaptation mechanism to the neural cache. The main difference is that the pointer-sentinel RNN has this adaptation mechanism built in during training, whereas the neural cache trains the model normally, and only adds this during evaluation time.

The neural cache is not able to change the hidden representation used to encode sequences–this remains fixed at test time. This capability is critical for adapting to

sequences in which each element has very little independent meaning, e.g. character level language modeling. Additionally, the neural cache can only raise the probability of symbols it has previously seen in a test sequence, which could limit its generalization ability in word-level language modeling.

2.2.2 Dynamic evaluation

Mikolov et al. (2010) proposed dynamic evaluation of neural language models at test time, with stochastic gradient descent (SGD) updates at every time step, computing the gradient with fully truncated backpropagation through time. Dynamic evaluation has since been applied to character and word-level language models (Graves, 2013; Ororbia II et al., 2017; Fortunato et al., 2017) (and by Krause et al. (2016) in work that is a precursor to Chapter 3 on mLSTMs). Previous work on dynamic evaluation applied it as additional updates at test time, and did not study dynamic evaluation methodology or explore why dynamic evaluation could work. Dynamic evaluation was often understood as a way of using the test data as extra training data, where as this thesis takes the perspective that dynamic evaluation is an additional way for a probabilistic sequence model to condition on the sequence history. Dynamic evaluation is studied in Chapters 4, 5, and 6.

Both dynamic evaluation and the neural cache can be used to adapt a base model at test time. The main difference is the mechanism used to fit to recent history: the neural cache uses a non-parametric, nearest neighbors-like method, whereas dynamic evaluation uses gradient descent. Both methods rely on an autoregressive factorization, as they depend on observing sequence elements after they are predicted in order to perform adaptation. Dynamic evaluation and neural caching methods are therefore both applicable to sequence prediction and generation tasks, but not directly to more general supervised learning tasks.

Dynamic evaluation as applied at test time, could be considered a form of fast weights (Schmidhuber, 1992; Ba et al., 2016a) – recurrent architectures with dynamically changing weight matrices as a function of recent sequence history. In traditional fast weights, the network learns to control changes to the weights during training time, allowing it to be applied to more general sequence problems including sequence labeling. In dynamic evaluation, the procedure to change the weights is automated at test time via gradient descent, making it only directly applicable to autoregressive sequence modeling. As dynamic evaluation leverages gradient descent, it has the potential to

generalize better to previously unseen pattern repetitions at test time.

2.3 Recurrent neural networks

Recurrent neural networks (RNNs) are a class of neural network architecture designed to model sequential data by using a hidden state to summarize the history of inputs. Recurrent neural networks were of historical interest as they were recognized to be powerful function approximators theoretically capable of representing any algorithm that can be run on a computer (Siegelmann and Sontag, 1995). The majority of the experiments in this thesis attempt to make recurrent neural networks more adaptive, so understanding them is important to understanding the context of this work. This section first walks through the vanilla RNN, which was used in the earliest RNN based approaches to language modeling (for instance in Mikolov et al. (2010)). We then cover the backpropogation through time algorithm, which is used to compute the gradient of RNNs. Gradients are typically used to train models, in dynamic evaluation, gradients are also used during test time. Therefore, at test time, the backpropagation through time algorithm is inherently built into a dynamic evaluated RNN. Lastly, we cover the LSTM architecture, an improved RNN that is the baseline for the majority of the experiments in this thesis.

2.3.1 Vanilla RNN

A vanilla RNN takes in an input sequence $x_{1:T} = \{x_1, \ldots, x_T\}$, and maps it to an output sequence $y_{1:T} = \{y_1, \ldots, y_T\}$ (or in some cases just a single output y at the end of the input sequence). The hidden state vector of an RNN at time t, h_t, is a function of the previous hidden state h_{t-1} and the new input x_t. In a vanilla RNN, this function is given by

$$\hat{h}_t = W_{hx} x_t + W_{hh} h_{t-1} + b_h \tag{2.4}$$

and

$$h_t = \tanh(\hat{h}_t), \tag{2.5}$$

where W_{hx} and W_{hh} are the input-to-hidden and hidden-to-hidden weight matrices respectively, b_h is a bias vector, $\hat{h}_t$ is the unsquashed hidden state vector, and tanh is the hyperbolic tangent function (which squashes the hidden state to be in the range $[-1, 1]$).

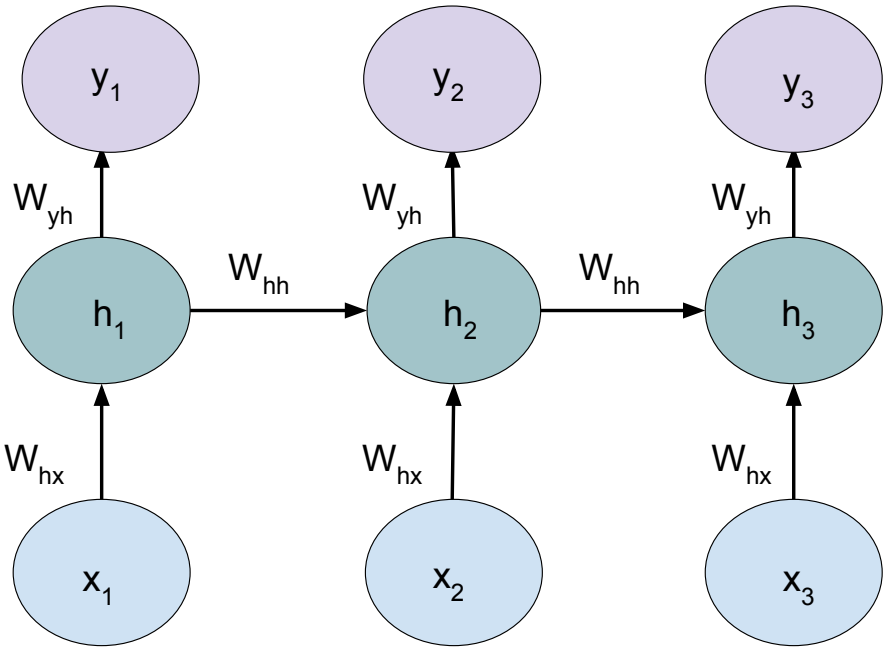


Figure 2.2: A vanilla RNN. Edges represent multiplication by a weight matrix, and nodes represent state vectors. A non-linear function, such as a $tanh$, is typically applied at the hidden state.

The unnormalized output $\hat{y}_t$ is then given by

$$\hat{y}_t = W_{yh} h_t + b_y. \tag{2.6}$$

If real valued outputs are desired, $y_t = \hat{y}_t$ can be used, whereas if probabilistic outputs are desired, $y_t = \text{softmax}(\hat{y}_t)$ can be used (where $\text{softmax}(z_i) = \frac{exp(z_i)}{\sum_j exp(z_j)}$ (Bridle, 1990)). An RNN is represented graphically in Figure 2.2.

2.3.2 Backpropagation through time

Neural networks are generally trained and evaluated using a loss $\mathcal{L}$ representing how close the outputs of the network y are to the desired outputs or targets γ (where for language modeling, γ will be a one hot vector with a 1 at the index of the target word, and a zero everywhere else). Backpropagation (Rumelhart et al., 1986) can be used to find derivatives of the loss with respect to the parameters of the network (gradients),which allow the parameters of the network to be updated in ways that reduce the loss (see Section 2.5 on optimization). Backpropagation through time (Werbos, 1990) specifically

refers to backpropagation of RNNs, where applying backpropagation involves unfolding the RNN in time.

For recurrent neural networks modeling categorical probabilistic outputs, the loss at each time step is typically given by the cross entropy error between the RNNs outputs at time t, y_t, and the target outputs at time t, γ_t.

$$\mathcal{L}_t = -\gamma_t^\top \log(y_t) \tag{2.7}$$

The total loss $\mathcal{L}$ is the sum of the losses over time ($\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t$).

The partial derivatives of the cross entropy error $\mathcal{L}$ with respect to the parameters of the network θ, $\frac{\partial \mathcal{L}}{\partial \theta}$, also known as the gradient or direction of steepest descent, can be computed using the chain rule with the back propagation through time algorithm, which is given for a standard RNN with a softmax output layer and cross entropy loss with the following equations below.

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_t} = y_t - \gamma_t \tag{2.8}$$

$$\frac{\partial \mathcal{L}}{\partial W_{yh}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial \hat{y}_t} h_t^\top \tag{2.9}$$

$$\frac{\partial \mathcal{L}}{\partial b_y} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial \hat{y}_t} \tag{2.10}$$

$$\frac{\partial \mathcal{L}}{\partial h_t} = W_{yh}^\top \frac{\partial \mathcal{L}}{\partial \hat{y}_t} + W_{hh}^\top \frac{\partial \mathcal{L}}{\partial \hat{h}_{t+1}} \tag{2.11}$$

$$\frac{\partial \mathcal{L}}{\partial \hat{h}_t} = \frac{\partial \mathcal{L}}{\partial h_t} \odot (1 - h_t \odot h_t) \tag{2.12}$$

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t=2}^{T} \frac{\partial \mathcal{L}}{\partial \hat{h}_t} h_{t-1}^\top \tag{2.13}$$

$$\frac{\partial \mathcal{L}}{\partial W_{hi}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial \hat{h}_t} x_t^\top \tag{2.14}$$

$$\frac{\partial \mathcal{L}}{\partial b_h} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial \hat{h}_t} \tag{2.15}$$

Note that the $\odot$ operator is the Hadamard product for element-wise matrix multiplication. RNNs can then be trained using gradient-based update schemes, such as the ones given in Section 2.5.

In the experiments in this thesis, the training sequences are often too long to efficiently compute backpropagation through time over full sequences. In this case, truncated backpropagation through time (Williams and Peng, 1990) is used, where the gradient is computed on shorter sequence segments and used to update the network before progressing to the next sequence segment. In this case, the hidden state from the end of the previous sequence segment can be used to initialize the hidden state on the next sequence segment (Graves, 2013). This variant of truncated backpropagation through time, we use both for training RNNs as well as for dynamic evaluation, is given in Algorithm 1.

$t \leftarrow 1$;
$h_0 \leftarrow zeros$;
while $t \leq T$ **do**
  $y_{t:t+\tau-1}, h_{t+\tau-1} \leftarrow RNN(x_{t:t+\tau-1}, h_{t:t+\tau-1})$;
  compute $\mathcal{L} = \sum \mathcal{L}_{t:t+\tau-1}$ using outputs $y_{t:t+\tau-1}$ and $\gamma_{t:t+\tau-1}$;
  compute gradients $\nabla \mathcal{L}(\theta)$;
  apply update rule ;
  $t \leftarrow t + \tau$;
end

Algorithm 1: Truncated backpropagation through time with a reused hidden state. Backpropagation through time is computed over sequence segments of length τ, and gradients are used to update the network before progressing to the next sequence segment. The final hidden state from a sequence segment is used as the initial hidden state in the next sequence segment. Note that the subscript notation $i : j$ specifies the range of vectors associated with timestep i through timestep j inclusive.

Training RNNs can result in a training difficulty known as the vanishing/exploding gradient problem (Bengio et al., 1994a; Hochreiter et al., 2001), where the gradient tends to decay or explode exponentially as it is back-propagated through time. It can be seen why this problem arises by considering the matrix of derivatives of the hidden states at a given time point with respect to hidden states n time steps in the past $\frac{\partial h_t}{\partial h_{t-n}}$.

$$\frac{\partial h_t}{\partial h_{t-n}} = \prod_{k=0}^{n-1} W_{hh}^T \operatorname{diag}(1 - h_{t-k} \odot h_{t-k}) \tag{2.16}$$

For a large n, this matrix of derivatives will tend to either explode or decay expo-

nentially because it is the product of many matrices. This result makes it difficult for RNNs to learn to use long contexts in their predictions. When the gradient vanishes, the updates to the weights will not help with learning statistical dependencies over long time lags because this contribution to the gradient will be exponentially small, and when the gradient explodes learning becomes unstable. For this reason, more advanced architectures and/or learning algorithms are usually needed to train RNNs on difficult problems. One way of addressing the exploding gradient (but not vanishing gradient) problem is to apply gradient norm clipping (Pascanu et al., 2013b), where the norm of the gradient $||\nabla\mathcal{L}(\theta)||$ is reduced when it exceeds the norm threshold λ.

if $||\nabla\mathcal{L}(\theta)|| > \lambda$ **then**
  $\nabla\mathcal{L}(\theta) \leftarrow \frac{\lambda\nabla\mathcal{L}(\theta)}{||\nabla\mathcal{L}(\theta)||}$
end

Algorithm 2: Gradient norm clipping

Some RNN architectures are designed to partially address exploding and vanishing gradients, including the long short-term memory (LSTM) architecture, which is covered next.

2.3.3 Long short-term memory

LSTM is a commonly used RNN architecture that uses multiplicative gates to control how information flows in and out of internal states of the network (Hochreiter and Schmidhuber, 1997). We use LSTM as one of the main baseline architectures for methods in this thesis. Each LSTM unit has a gated memory cell which allows the network to preserve or overwrite the state of each unit. The ability to preserve the value of memory cells gives LSTMs the ability to retain information over longer time periods.

Like a standard RNN, the LSTM hidden state receives inputs from the input layer x_t and the previous hidden state h_{t-1}:

$$\hat{h}_t = W_{hx}x_t + W_{hh}h_{t-1}. \tag{2.17}$$

The LSTM network also has 3 gating units – input gate i, output gate o, and forget gate f (introduced in Gers et al. (2000)) – that have both recurrent and feed-forward

connections:

$$i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1}) \tag{2.18}$$

$$o_t = \sigma(W_{ox}x_t + W_{oh}h_{t-1}) \tag{2.19}$$

$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1}), \tag{2.20}$$

where σ is the logistic sigmoid function. The input gate controls how much of the input to each hidden unit is written to the memory cell vector c_t, and the forget gate determines how much of the previous memory cell vector c_{t-1} is preserved. This combination of write and forget gates allows the network to control what information should be stored and overwritten across each time-step. The memory cell vector is updated by

$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(\hat{h}_t). \tag{2.21}$$

The output gate controls how much of each unit's activation is preserved. It allows the LSTM cell to keep information that is not relevant to the current output, but may be relevant later. The final output of the hidden state is given by

$$h_t = \tanh(c_t) \odot o_t. \tag{2.22}$$

LSTM was derived to address the vanishing/exploding gradients in RNNs, as the LSTM's memory cells make it possible to pass information forward or pass gradients backward undisturbed more easily. LSTM has proven useful in practice, achieving strong results in many sequence modeling tasks (Graves et al., 2013; Graves, 2013; Zaremba et al., 2014).

2.4 Sequence modeling architectures

We review sequence modeling architectures that are either directly used or helped motivate approaches in this thesis. For instance, we use architectural features from the multiplicative RNN (Sutskever et al., 2011) to make LSTMs more adaptive at the token level in our work with multiplicative LSTM in Chapter 3. The use of depth and recurrent depth are competing approaches to make RNNs more adaptive, and thus are important to the context of the field. Multiplicative integration RNNs (Wu et al., 2016) also closely relate to the multiplicative LSTM approach. Normalization methods are used as an architectural feature throughout this thesis, and are thus important for having a full understanding of the architectures we use. Transformers (Vaswani et al., 2017)

are a stronger and more recent architecture that is used as a baseline in Chapter 6. Tied embedding matrices (Press and Wolf, 2017; Inan et al., 2017) are a useful architectural feature for language modeling used in several experiments, and considered in analysis of how dynamic evaluation can generalize to words with similar embeddings in Section 5.3. Lastly, we describe the importance of having strong baselines when making claims about architectural improvements, a principle used for experiments in this thesis.

2.4.1 Multiplicative RNN

The multiplicative RNN (Sutskever et al., 2011, mRNN) is an architecture designed specifically to allow flexible input-dependent transitions. mRNNs are a precursor to multiplicative LSTMs, which are the focus of Chapter 3. mRNN's formulation was inspired by the tensor RNN, an RNN architecture that allows for a different transition matrix for each possible input. The tensor RNN features a 3-way tensor $W_{hh}^{1:N}$, which contains a separately learned transition matrix W_{hh} for each input dimension. The 3-way tensor can be stored as an array of matrices

$$W_{hh}^{(1:N)} = \{W_{hh}^{(1)}, ..., W_{hh}^{(N)}\}, \tag{2.23}$$

where superscript is used to denote the index in the array, and N is the dimensionality of x_t. The specific hidden-to-hidden weight matrix $W_{hh}^{(x_t)}$ used for a given input x_t is then

$$W_{hh}^{(x_t)} = \sum_{n=1}^{N} W_{hh}^{(n)} x_t^{(n)}. \tag{2.24}$$

For sequence modeling problems where x_t is one-hot (such as language modeling 2.1), and $W_{hh}^{(x_t)}$ will be the matrix in $W_{hh}^{(1:N)}$ corresponding to that unit. Hidden-to-hidden propagation in the tensor RNN is then given by

$$\hat{h}(t) = W_{hh}^{(x_t)} h_{t-1} + W_{hx} x_t. \tag{2.25}$$

The large number of parameters in the tensor RNN make it impractical for most problems. mRNNs can be thought of as a shared-parameter approximation to the tensor RNN that use a factorized hidden-to-hidden transition matrix in place of the normal RNN hidden-to-hidden matrix W_{hh}, with an input-dependent intermediate diagonal matrix $\text{diag}(W_{mx} x_t)$. The input-dependent hidden-to-hidden weight matrix, $W_{hh}^{(x_t)}$ is then

$$W_{hh}^{(x_t)} = W_{hm} \text{diag}(W_{mx} x_t) W_{mh}. \tag{2.26}$$

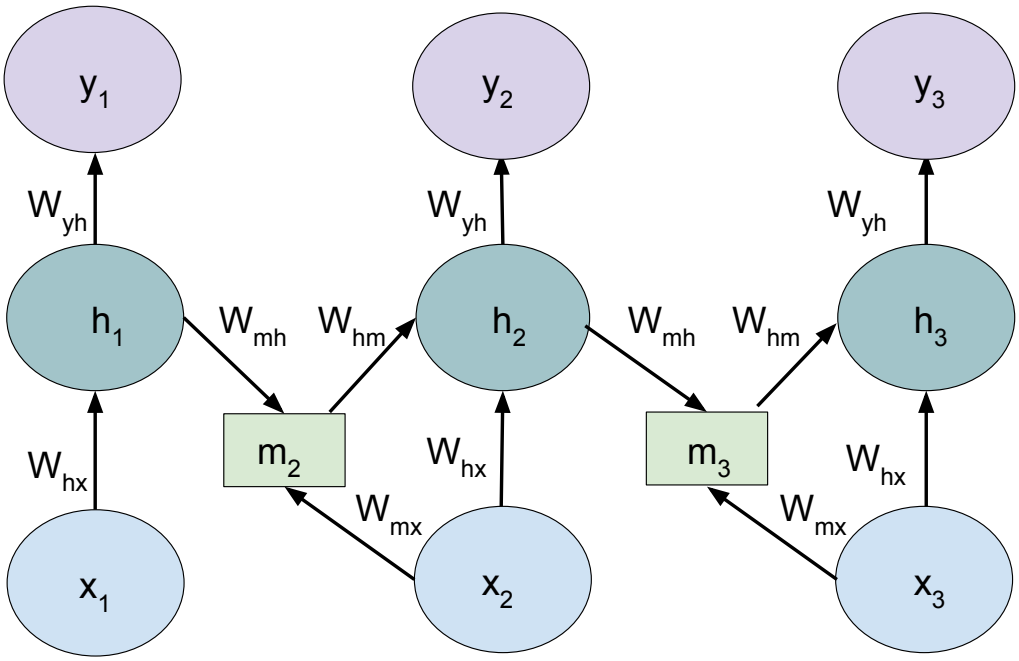


Figure 2.3: A multiplicative RNN. Inputs to the elliptical nodes are added, and inputs to the rectangular nodes are multiplied.

An mRNN is thus equivalent to a tensor RNN using the above form for $W_{hh}^{(x_t)}$. For readability, an mRNN can also be described using intermediate state m_t as follows:

$$m_t = (W_{mx}x_t) \odot (W_{mh}h_{t-1}) \tag{2.27}$$

$$\hat{h}_t = W_{hm}m_t + W_{hx}x_t. \tag{2.28}$$

mRNNS are illustrated graphically in Figure 2.3.

mRNNs have improved on vanilla RNNs at character level language modeling tasks (Sutskever et al., 2011; Mikolov et al., 2012a), but have fallen short of the more popular LSTM architecture, for instance as shown with LSTM baselines from (Cooijmans et al., 2017). The standard RNN units in an mRNN do not provide an easy way for information to bypass its complex transitions, resulting in the potential for difficulty in retaining long term information.

2.4.2 Depth and recurrent depth

In deep learning, depth is the concept stacking multiple non-linear functions. While traditional RNNs are already deep in the sense that they repeatedly apply non-linear layers over time, various methods have been developed to make RNNs deeper (Pascanu

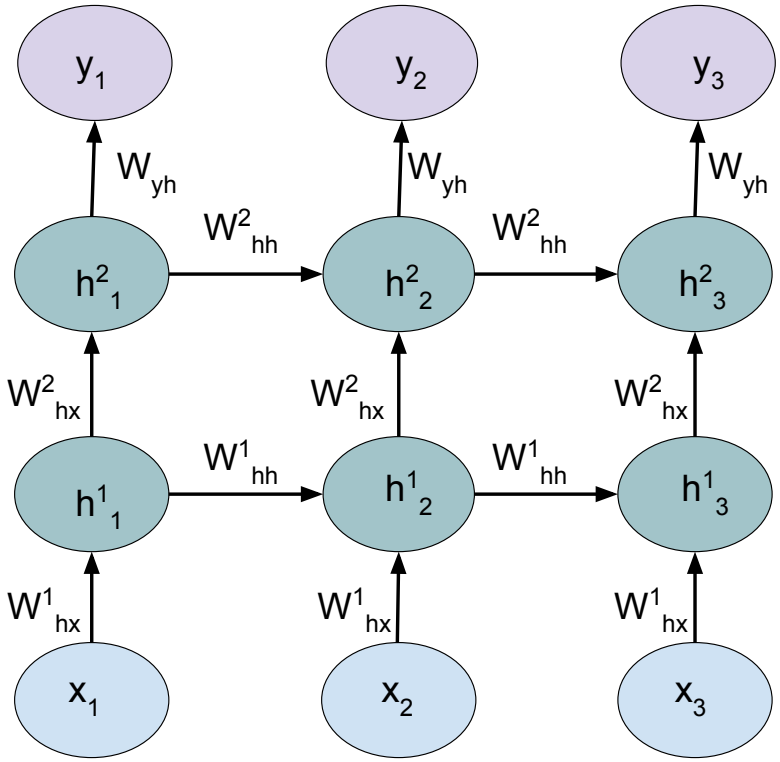


Figure 2.4: A 2-layer stacked RNN. The second RNN takes the hidden state of the first RNN as its input.

et al., 2013a; Graves, 2013; Hermans and Schrauwen, 2013). Adding depth to an RNN can making it more adaptive by giving it the ability to model a greater range of functions. All RNN language models combine the input from the previous hidden state, and from the current token, to create a new hidden state. A "shallow RNN" is limited to combining these inputs using a linear combination of them, with single non-linearity applied. Increasing the depth means that a greater range of functions can be learned to combine the contributions from the previous hidden state and the current input. This could make it easier for a model with greater depth to make large changes to its hidden state as a result of a new input. While adding depth makes a model more adaptive, it also makes it more expensive, because computation over successive layers cannot be paralellized (since the output from one layer is the input to the next layer). This thesis seeks a simpler method to make RNNs more adaptive without adding extra depth.

One of the most common ways of making an RNN deep is to use a stacked RNN (Graves, 2013), where the hidden state h_t of one RNN becomes the input x_t of another RNN, as illustrated in Figure 2.4.

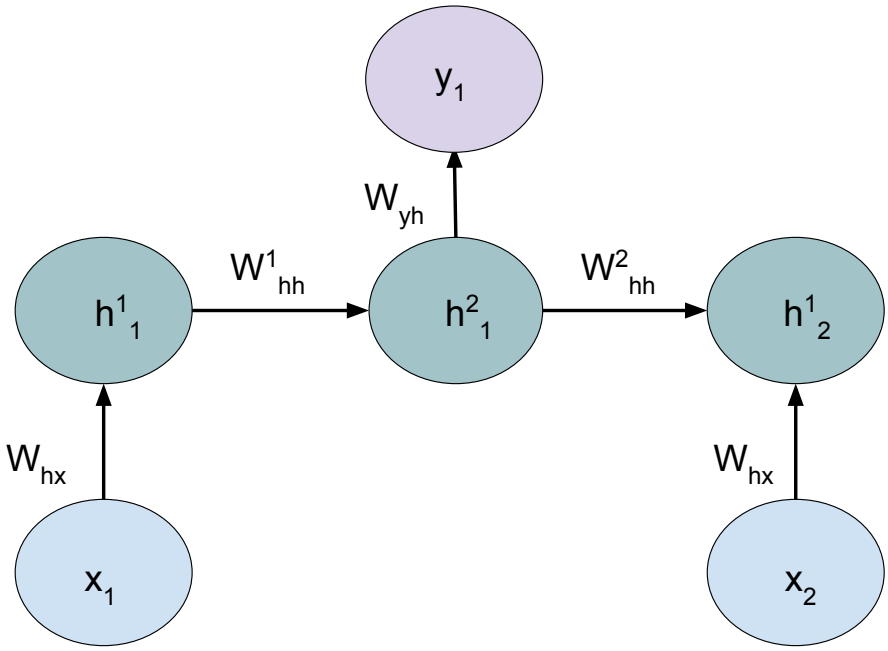


Figure 2.5: An RNN with a recurrent depth of 2. The model has an additional non-linear layer between recurrent steps.

Many recently proposed RNN architectures use recurrent depth, which is depth between recurrent steps, as illustrated in Figure 2.5. Recurrent depth allows more non-linearity in the combination of inputs and previous hidden states from every time step. Recurrent depth has been found to perform better than other kinds of non-recurrent depth for some sequence modeling problems (Zhang et al., 2016).

Recurrent highway networks (Zilly et al., 2017, RHNs) use a more sophisticated recurrent depth that carefully controls propagation through layers using gating units. Fast-slow recurrent neural networks (Mujika et al., 2017) combined the ideas of having RNNs at different scales, with recurrent depth. Overall, the ability to have complex transitions between timesteps seems to help RNNs.

2.4.3 Multiplicative integration RNNs

Multiplicative integration RNNs (Wu et al., 2016, MI-RNNs) use Hadamard products instead of addition when combining contributions from input and hidden units. This allows the hidden states to be modified by the input token to a greater degree, thus making MI-RNNs more adaptive at the token level. In the case of vanilla MI-RNNs,

Equation 2.17 for the input to the RNN hidden state becomes

$$\hat{h}_t = W_{hx} x_t \odot W_{hh} h_{t-1}. \tag{2.29}$$

This modification can also be extended to LSTMs. Like recurrent depth, this multiplicative interaction allows for more complex transitions between timesteps, but unlike recurrent depth, it does not require additional sequential computation between timesteps. MI-RNNs relate closely to mRNNs, with the differences being that mRNNS apply an additional matrix multiplication in the transition between timesteps, and mRNNs still use addition when combining contributions from input and hidden units.

2.4.4 Normalization methods

Normalization methods are an architectural feature aimed at improving both regularization and optimization in RNNs and other sequence models. A number of the architectures used in experiments in this thesis use normalization methods to improve performance. Normalizing the hidden state and/or weights of RNNs can make them more robust to vanishing and exploding gradients, and thus more stable. In feed forward neural networks, batch normalization (Ioffe and Szegedy, 2015) is an effective approach. Batch normalization normalizes across the hidden states of dimensionality d in a minibatch of dimensionality n so that they always have the same mean and variance, by applying the equation

$$h = \frac{\hat{h} - \mu(\hat{h})}{\sqrt{\sigma^2(\hat{h}) + \varepsilon}} \tag{2.30}$$

where $\hat{h} \in \mathbb{R}^{n,d}$ is the mini-batch of pre-normalized hidden states, $h \in \mathbb{R}^{n,d}$ is the mini-batch of final hidden states, $\mu(\hat{h})$ and $\sigma^2(\hat{h})$ are vectors of the mini batch means and variances of each hidden unit, and ε is used for stability. The values of $\mu(\hat{h})$ and $\sigma^2(\hat{h})$ depend on the minibatch sampled and therefore add noise to the network. At test time, $\mu(\hat{h})$ and $\sigma^2(\hat{h})$ are set to the means and variances across the entire training set. Applying batch normalization naively to RNNs results in instability because the hidden state will have different means and variances near the beginning of a sequence. Recurrent batch normalization (Cooijmans et al., 2017) attempts to address this by only normalizing across hidden states with the same amount of context (So for instance, when normalizing the 5th hidden state in a sequence, recurrent batch normalization only uses statistics from other hidden states that are also 5th in their respective sequences).

Recurrent batch normalization also separately normalizes hidden-to-hidden and input-to-hidden contributions to the hidden state. While recurrent batch normalization can help in practice, it adds extra steps to the normalization process, requiring more design decisions, hyper parameters, and overall effort to implement.

Other normalization approaches are more straightforward to implement in RNNs. Layer normalization (Ba et al., 2016b) normalizes hidden states across a layer rather than across a batch. Layer normalization introduces additional bias and gain parameters b and g.

$$h_t = \frac{g}{\sigma_t} \odot (\hat{h}_t - \mu_t) + b \tag{2.31}$$

where μ_t and σ_t are the mean and variance across unnormalized hidden layer $\hat{h}_t$.

Another approach known as weight normalization (Salimans and Kingma, 2016) does not directly modify the model, and simply reparameterizes the weight matrices. Weight normalization is applied separately to the weights of each unit in a layer (as opposed to the entire weight matrix of a layer). Each unit has as associated unnormalized weight vector $\hat{w}$, and a scalar parameter v. The weights w of a unit are then given as

$$w \leftarrow v \frac{\hat{w}}{||\hat{w}||_2}. \tag{2.32}$$

The w for a particular unit corresponds to a row of the overall weight matrix in the layer. In weight normalization, $\hat{w}$ and v are treated as learnable parameters, whereas standard models would just learn w directly. This reparameterization allows for the weights of a unit to be rescaled quickly, which can improve the stability of training and result in better generalization.

2.4.5 Attention and Transformers

The ability to use long contexts to make predictions is important for adaptation; it is impossible for a model to adapt to inputs that it cannot remember. Architectures based on attention, including the Transformer (Vaswani et al., 2017), are designed to better use long contexts to make predictions. Transformers have been shown to effectively use longer range dependencies than RNNs (Dai et al., 2019) and have achieved recent state of the art in language modeling and a number of sequence modeling and classification tasks (Vaswani et al., 2017; Radford et al., 2018; Devlin et al., 2018; Dai et al., 2019). Much of the work on Transformers is concurrent with or after the work done in this thesis. For instance, Transformers first achieved strong results on common language

modeling benchmarks in work by (Al-Rfou et al., 2018), which was after work in chapters 3, 4, and 5 was published. Chapter 6 shows that adaptive methods developed in this thesis can also improve Transformer baselines.

2.4.5.1 Attention

Attention is a generally useful architectural feature in sequence modeling that helps models use context to make predictions. In content based attention, the model stores key vectors $K = k_1, ..., k_t, .., k_{d_t}$, and value vectors $V = v_1, ..., v_t, .., v_{d_t}$ associated with each timestep as it processes a sequence, where $K \in \mathbb{R}^{d_t, d_h}$, $V \in \mathbb{R}^{d_t, d_h}$, d_h is the embedding size, and d_t is the sequence length over which attention is applied (note that the second dimension size for K and V can be different, but are typically set to be the same, so for simplicity we assume they are both d_h). The model then can query the stored keys with query vector $q \in \mathbb{R}^{d_h}$ (the dimensionality of q must match the second dimension of K). The model effectively "attends" to each position in a sequence based on a similarity metric between q and k_t at each time step t. The unnormalized attention score for time t could for instance be given by the inner product $q^\top k_t$. These attention scores are them normalized with a softmax applied over the temporal dimension, resulting normalized attention score a

$$a = softmax(q^\top K^\top) \tag{2.33}$$

The normalized attention scores can be interpreted as the weight at which the model attends to each time step. The output of the attention operation is then a weighted sum of the values vectors, weighted by the normalized attention scores at each timestep.

$$y = V^\top a^\top \tag{2.34}$$

There are many different variants of attention, including different ways of computing scores between key and query vectors, and ways of encoding positional information. Neural attention mechanisms have been generally useful for sequence modeling tasks, including question answering (Sukhbaatar et al., 2015), neural programming (Graves et al., 2014), and machine translation (Bahdanau et al., 2015).

2.4.5.2 Transformers

Transformers use stacked layers (see Section 2.4.2 on stacked RNNs) composed of self-attention (where each position in the sequence has attention over every previous position in the sequence) and position-wise feed forward operations to model sequences (Vaswani

et al., 2017). Transformers are largely motivated by direct paths for information to travel across time via attention, and the ability to process sequences in parallel. Some of the design decisions of a Transformer vary slightly from paper to paper and depending on the application. The description of a Transformer given here is for Transformers as used in language modeling, which only use uni-directional attention, meaning each position only has attention over previous positions. The original Transformer (Vaswani et al., 2017) used an encoder and decoder, and required separate attention to allow interactions between the encoder and decoder, in addition to bi-directional attention within the encoder.

The input to Transformer layer l is a sequence of vectors representing the state at each timestep, stored in matrix $X_l \in \mathbb{R}^{d_t,d_x}$, where d_x is the dimension of the embedding vector (and typically also the dimension of every layer in the model), and d_t is the sequence length over which the Transformer operates. Since Transformers only use attention to model the past context, positional encodings are needed to use information about the order of the sequence history. Transformers from Vaswani et al. (2017) used a positional encoding matrix $U \in \mathbb{R}^{d_t,d_x}$, given by

$$U(t,2n) = \sin\left(\frac{t}{10000^{\frac{2n}{d_x}}}\right) \tag{2.35}$$

$$U(t,2n+1) = \cos\left(\frac{t}{10000^{\frac{2n}{d_x}}}\right) \tag{2.36}$$

where the 10000 term could be seen as a constant that controls the rate of wavelength progression in the positional encoding. For each embedding dimension, the position is encoded by the value of a sine or cosine wave. The sine and cosine waves run at different frequencies at each embedding dimension so that each embedding dimension contains unique information. Positional encodings are added to X_l, to result in positionally encoded inputs X_p

$$X_p = X_l + U \tag{2.37}$$

Attention in Transformers is typically multi-headed, meaning that multiple instances of attention with different parameters are applied simultaneously. This allows the model to attend to several different places in the sequence history. The following equations are applied in parallel for each attention head i.

$$Q^i = X_p(W^i_{qx})^\top \tag{2.38}$$

$$K^i = X_p(W^i_{kx})^\top \tag{2.39}$$

$$V^i = X_p(W^i_{vx})^\top \tag{2.40}$$

Where $W^i_{qx} \in \mathbb{R}^{d_h,d_x}, W^i_{kx} \in \mathbb{R}^{d_h,d_x}, W^i_{vx} \in \mathbb{R}^{d_h,d_x}$ are learnable parameters, $Q^i \in \mathbb{R}^{d_t,d_h}$,$K^i \in \mathbb{R}^{d_t,d_h}$,$V^i \in \mathbb{R}^{d_t,d_h}$, are the query, key, and value vectors for each attention head, and d_h is the dimensionality of each attention head. Typically, d_h is set to $\frac{d_x}{NumHeads}$. Attention scores are then computed using a softmax over a similarity metric between keys and values.

$$A^i = \text{softmax}(\frac{Q^i(K^i)^\top}{\sqrt{d_h}}) \tag{2.41}$$

Where $A^i \in \mathbb{R}^{d_t,d_t}$, and the softmax is applied over the second dimension of the attention scores. The $\sqrt{d_h}$ term in the denominator helps normalize the attention values so that the softmax does not become too sharply peaked for models with a larger d_h. In a Transformer used for language modeling, the model at timestep t is not allowed to have access to any information at timesteps $n > t$, since this would allow the model to "cheat" by seeing the future. For this reason, masked softmax is applied instead of a normal softmax. A masked softmax enforces that $A^i_{t,n\geq t} = 0$. In practice, this is done by adding large negative numbers to the pre-softmax attention scores at the indices that need to be 0. The attention values for each head, $H_i \in \mathbb{R}^{d_t,d_h}$, are computed by

$$H^i = A^i V^i \tag{2.42}$$

The outputs of all the attention heads are concatenated together, run through a linear layer with learnable parameters $W_{zh} \in \mathbb{R}^{d_h,d_h}$. This result is added to the initial input to the layer X_l, making the Transformer layer a type of residual layer (He et al., 2016), making it easier for a Transformer to retain information propagated through many layers. A learnable layer normalization function (Section 2.4.4) is then applied to help prevent layer activations/gradients from blowing up.

$$Z = LayerNorm(X_l + concat(H^1,..,H^i,..,H^n)^\top W_{zh}) \tag{2.43}$$

The result is then run through a position-wise feed forward network with one hidden layer.

$$\hat{X}_{l+1} = relu(Z^\top W_{yz1} + b_{z1})^\top W_{yz2} + b_{z2} \tag{2.44}$$

Where W_{yz1}, W_{yz2}, b_{z1}, and b_{z2} are learnable parameters. Finally, another residual and layer normalization operation are applied to get the output of the layer.

$$X_{l+1} = LayerNorm(\hat{X}_{l+1} + Z) \tag{2.45}$$

Transformers are composed of many stacked Transformer layers, where the output of Transformer layer l, X_{l+1}, is then the input to Transformer layer $l+1$. We define the series of equations described above that maps X_l to X_{l+1} as the function $TransformerLayer_l(X_l)$. The full Transformer architecture uses stacked Transformer layers to map a sequence of inputs to a sequence of output probabilities. While this is trivial, for completeness, this process is given in Algorithm 3.

Apply input layer across the sequence to obtain initial embeddings for X_1 ;
#Apply L layers of Transformer ;
for *l=1...L* **do**
  $X_{l+1} \leftarrow TransformerLayer_l(X_l)$;
end
Apply output layer and softmax to X_{L+1} ;

Algorithm 3: Transformer decoder. Each Transformer layer is applied to a sequence of embeddings to create a new sequence of embeddings. The final embeddings are then fed into a softmax output layer applied in parallel across the sequence to obtain token probabilities at each time step.

2.4.6 Tied embedding matrices

A useful architectural modification to language models often used in this thesis that can improve performance is to tie input and output word embeddings (Press and Wolf, 2017; Inan et al., 2017). Language models have input and output word embedding matrices, which are the input and output weights of the network. If the vanilla RNN from Section 2.3 was applied directly to language modeling, then W_{hx} would be the input word embedding matrix, and W_{yh} would be the output word embedding matrix. Each row/column of the input/output embedding matrix is a vector associated with a corresponding word in the vocabulary of possible words. Tying input and output embedding matrices in this case means setting $W_{hx} = W_{yh}^{\top}$, and treating the input and output embedding matrices and one parameter matrix, and updating them together

during training. This case of tied embeddings is illustrated graphically in Figure 2.6.

Often language models will have a separate linear embedding layer, instead of directly connecting the input layer to the RNN hidden state. In this case, W_{hx} is factorized into the product of embedding to hidden matrix W_{he} and input embedding matrix W_{ex}. With tied embeddings, $W_{ex} = W_{yh}^{\top}$, forcing the last hidden layer dimensionality to match the embedding layer dimensionality. So generally, if a model has an embedding layer e_t (which could be recurrent or non-recurrent), then

$$e_t = W_{ex} x_t \tag{2.46}$$

Then in the output layer,

$$\hat{y}_t = (W_{ex})^{\top} h_t \tag{2.47}$$

where $\hat{y}_t$ are the unnormalized probabilities for $P(x_{t+1})$. Tying the input and output word embeddings gives an advantage on several standard language modeling benchmarks. Constraining models to have the same input and output embedding representation for words likely has a regularization effect and gives these weights a stronger learning signal, since gradients in the input layer can affect the output layer and visa versa.

2.4.7 Importance of strong baselines

This thesis explores whether more adaptive methods can improve language models, but a fair comparison between different model architectures can be difficult, since performance on a task can largely depend on the hyper-parameter tuning of the model. Factors such as the regularization in each layer and the optimization method can have a large impact on performance, and the optimal settings can be different for different models. Melis et al. (2018) showed that LSTM can sometimes match other more complex RNNs on common benchmarks if appropriately tuned. For this reason, this thesis seeks to have strong baselines when justifying architectural modifications. Without this, it can be difficult to know if improvements are due to better hyper-parameter tuning, or if improvements would wash away if other stronger baselines were used.

2.5 Optimization

This thesis considers optimization algorithms in two contexts; training neural networks from a random initialization, and adapting neural networks from a trained initialization.

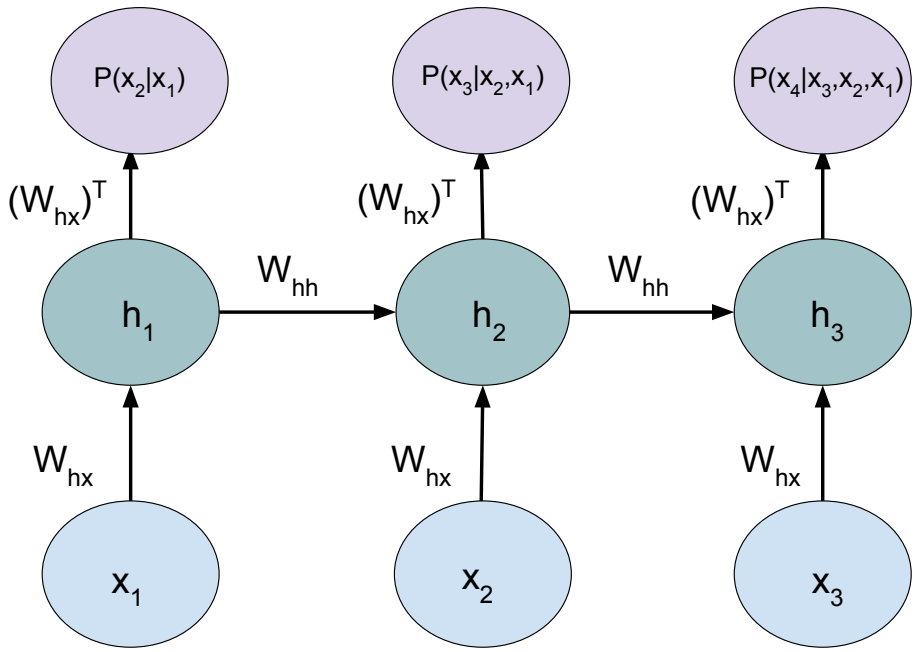


Figure 2.6: When using tied embeddings, the output weights of the RNN are set to be the transpose of the input weights to the RNN. This figure shows tied embedding matrices for a 1-layer RNN, however, in many cases a separate non-recurrent linear embedding layer will be used, and multiple RNNs may be stacked. In this case, the dimensionality of the embedding layer and last RNN layer must be equivalent for tied embedding matrices to be used.

In addition to being important for understanding general deep learning, optimization is especially important to understanding this thesis, because dynamic evaluation uses on-the-fly optimization to adapt to the sequence history. This thesis explores optimizers commonly used in training for dynamic evaluation, and proposes novel optimization algorithms that are especially useful in this setting. The optimization algorithms presented in this section are necessary to understand work in this thesis on optimization in a dynamic evaluation setting.

Most optimization algorithms used for training neural networks use derivatives to optimize an objective function. These methods are derived from gradient descent, where the derivatives of a training loss function $\mathcal{L}(\theta)$ with respect to the model parameters, θ, $\frac{\partial \mathcal{L}(\theta)}{\partial \theta}$, are used to train the network. $\mathcal{L}(\theta)$ is decreased by taking steps in the direction of the negative gradient with sufficiently small step sizes. The hope is that this will also result on a lower loss when encountering new data points outside the training set.

2.5.1 Stochastic gradient descent

Gradient descent is slow to run in practice because it requires computing the gradient across the whole training set. It is often more efficient to estimate the gradient using a subset of the training set (Robbins and Monro, 1951). At each iteration, stochastic gradient descent (SGD) uses a single training example, rather than the entire training set, to estimate the training loss and training gradient of the model. This requires much less computation per update than full gradient descent. The gradient estimate on a single training example is often a reasonable enough approximation of the true gradient for SGD to converge with much less computation compared with gradient descent.

In practice, it is often more efficient to use a minibatch of several training examples to estimate the gradient, as opposed to a single training example as in pure SGD. Since the computation for each training example can be done independently, the use of a minibatch allows for more efficient use of hardware designed for parallel computation. This allows for a more accurate gradient estimate at a limited extra cost. In minibatch SGD, at each training iteration i, a minibatch is sampled from the training set, and the gradient with respect to the loss on that minibatch, $\nabla \mathcal{L}(\theta_i)$, is computed, and used to update the weights of the network before proceeding to the next minibatch. The update rule for each training iteration is given by

$$\theta_{i+1} = \theta_i - \eta \nabla \mathcal{L}(\theta_i) \tag{2.48}$$

where η is the learning rate, or step size in the direction of the negative gradient. Typically training is either carried out for some fixed number of iterations, or until the loss on a held out validation set stops improving. The learning rate may be decayed as training continues, as smaller step sizes are generally required later in training to continue reducing the loss. The use of smaller minibatches in SGD as compared with large batch SGD or full batch gradient descent sometimes leads to better generalization (Keskar et al., 2016).

2.5.2 SGD with momentum

SGD can be modified to include momentum, which has the effect of delaying how gradients update the weights of the network, instead allowing the gradients to accumulate. Classical momentum (Polyak, 1964) keeps track of a running average of the mean gradient v_i (scaled by the negative learning rate η) as follows

$$v_i = \mu v_{i-1} - \eta \nabla \mathcal{L}(\theta_i), \tag{2.49}$$

where μ is the momentum constant for the running average. The update to the weights is then given by

$$\theta_{i+1} = \theta_i + v_i. \tag{2.50}$$

Momentum can improve training convergence speed for deep and recurrent neural networks (Rumelhart et al., 1986; Sutskever et al., 2013).

2.5.3 RMSprop

RMSprop (Tieleman and Hinton, 2012) is an optimization algorithm that reduces learning rates on weights that have higher average gradients, and increases learning rates on weights that have lower average gradients. RMSprop can be thought of as the minibatch version of Rprop (Riedmiller and Braun, 1993), a method that steps in the direction of the sign of the gradient. RMSprop is also partially inspired by inspired by Adagrad (Duchi et al., 2011), another method for reducing the learning rate of weights that have had high past gradients.

Weights that have higher gradients may also have higher 2nd order derivatives, and setting the learning rates for each weight proportional to the inverse curvature can help convergence. RMSprop using a running averaging of the recent squared gradients

(weighted by averaging coefficient α), MS, which is updated at each training iteration using

$$MS_i = \alpha MS_{i-1} + (1-\alpha)(\nabla \mathcal{L}(\theta_i))^2. \tag{2.51}$$

The gradients for each update are then divided by the square root of the running average of the mean squared gradients, before updating the weights of the network, using

$$\theta_{i+1} = \theta_i - \eta \frac{\nabla \mathcal{L}(\theta_i)}{\sqrt{MS_i} + \varepsilon}, \tag{2.52}$$

where ε is a hyper parameter needed for numerical stability in case the running average of the squared gradient for a particular weight is close to 0.

2.5.4 Adam

Adam (Kingma and Ba, 2014) combines RMSprop like updates with momentum, and is one of the most popular deep learning optimizers due to its fast convergence in a large variety of deep learning settings. Like SGD with momentum, Adam stores a running average of the recent gradients using

$$v_i = \mu v_{i-1} - \eta \nabla \mathcal{L}(\theta_i). \tag{2.53}$$

Like RMSprop, Adam stores a running average of the squared gradients using

$$MS_i = \alpha MS_{i-1} + (1-\alpha)(\nabla \mathcal{L}(\theta_i))^2. \tag{2.54}$$

Near the beginning of training, the running average for the mean and mean squared gradients will be too low if computed naively (since they are averaged with 0 at the first time step), so Adam applies a bias correction step to fix this.

$$\hat{v}_i = \frac{v_i}{1-\mu^i} \tag{2.55}$$

$$\hat{MS}_i = \frac{MS_i}{1-\alpha^i} \tag{2.56}$$

Adam then combines the momentum and RMSprop update rules, to yield the update equation

$$\theta_{i+1} = \theta_i + \frac{\hat{v}_i}{\sqrt{\hat{MS}_i} + \varepsilon}. \tag{2.57}$$

2.6 High-dimensional sequence modeling

While this thesis mainly considers the task of textual language modeling, we also apply dynamic evaluation to polyphonic music prediction to show that adaptive approaches can also help with non-textual sequences. Language modeling can be applied directly to model monophonic music, where the data are sequences of musical notes, with one note per timestep, making the notes analogous to words or tokens. Modeling polyphonic music is slightly more complicated, because each timestep can contain multiple musical notes. In this case, a softmax output layer applied at each time step is no longer sufficient for density estimation of sequences.

Polyphonic music can be represented as a "high dimensional sequence", or sequence of binary vectors, $x_{1:T}^{1:N}$, where each $x_t^{1:N}$ is a binary vector representing which notes out of N possible notes are on or off at timestep t. Since these sequences have a temporal and non-temporal component, they require different models to most suitably capture the structure of the data. We would like to avoid making an unnecessary independence assumptions; we could for instance have a sigmoid unit for each musical note predict whether that note was on or off, but this would assume that the musical notes within a timestep are conditionally independent given the sequence history. We describe a model used for our baseline that avoids making this independence assumption.

2.6.1 NADEs

For language modeling problems, the position of the elements in the sequence is important for making predictions. However, for density estimators of arbitrary sets of variables with no implicit order, positional information is not important, and ideally, we would want a model to treat the variables as permutation invariant. RNNs are highly sensitive to the permutation of the variables they model, making them they are less suited as density estimators of joint distributions of permutation invariant variables. Neural auto-regressive distribution estimators (Larochelle and Murray, 2011, NADEs) are an efficient method suited for density estimation in these cases.

A NADE uses a 1-layer neural network to predict a set of variables one-by-one. NADEs use a similar auto-regressive factorization to sequence models to predict a distribution over variables $P(x_{1:N})$, however unlike sequence models, the ordering of $x_{1:N}$ is arbitrary in the sense that any ordering can be used as long as it is fixed for

training and test time.

$$P(x_{1:N}) = P(x_1)\prod_{n=2}^{T} P(x_n|n_{1:n-1}). \tag{2.58}$$

A NADE can be described by the following series of equations. The probability distribution over the first variable x_1 is a function of the first output bias b_y^1.

$$P(x_1) = \sigma(b_y^1) \tag{2.59}$$

For the additional variables in $x_{1:N}$, a hidden state h_n is used to summarize inputs $x_{1:n}$ to estimate $P(x_{n+1}|x_{1:n})$. The hidden state preactivation vector $\hat{h}_n$ can be calculated as a sum of the rows of the hidden weights, where W_{hx_m} denotes the column of weights connecting x_m to the hidden state.

$$\hat{h}_n = b_h + \sum_{m=1}^{n} (W_{hx_m} x_m). \tag{2.60}$$

The hidden state preactivation can also be calculated as a function of the previous hidden state preactivation with

$$\hat{h}_n = \hat{h}_{n-1} + (W_{hx_n} x_n). \tag{2.61}$$

The activation function, which could for instance be a rectified linear (relu) function, is then applied to the hidden state.

$$h_n = \text{relu}(\hat{h}_n) \tag{2.62}$$

The inner product of the column of output weights W_{yh_n} is taken with hidden state h_n to determine the output probabilities for x_{n+1}.

$$P(x_{n+1}|x_{1:n}) = \sigma((W_{yh_n})^\top h_n + b_y^{n+1}) \tag{2.63}$$

A NADE is illustrated graphically in Figure 2.7.

2.6.2 RNN-NADEs

When dealing with sequences of high dimensional vectors, $x_{1:T}^{1:N}$, a naive application of a standard RNN assumes each variable in $x_t^{1:N}$ is conditionally independent given the sequence history, as an RNN is unable to utilize $x_t^{1:n-1}$), when predicting $p(x_t^n)$. By hybridizing an RNN and a NADE, as first done in Boulanger-Lewandowski et al.

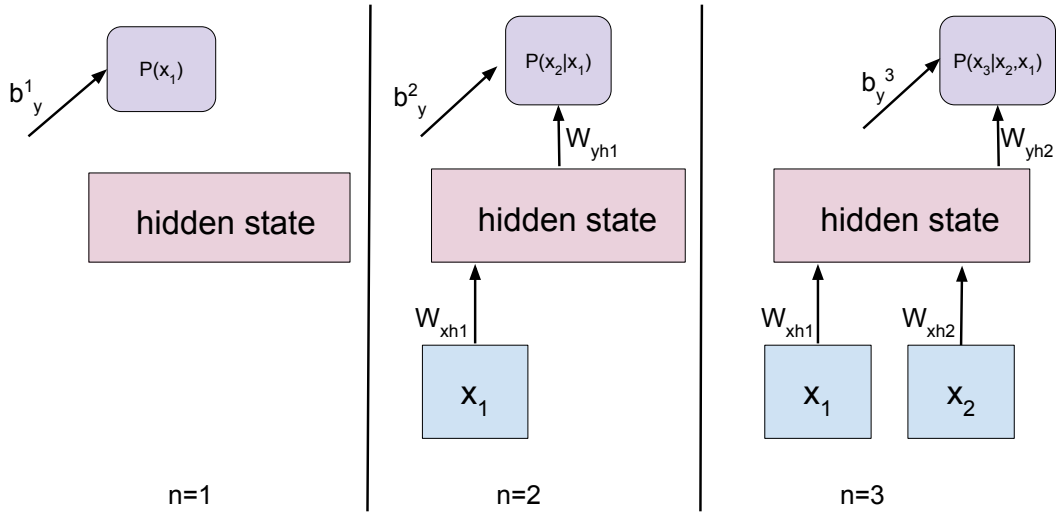


Figure 2.7: A NADE iterating through a high dimensional vector. The model iteratively predicts a distribution over the next symbol, and then conditions on the ground truth value of that symbol.

(2012), it is possible to drop this conditional independence assumption, and predict $p(x_t^n|x_{1:t-1}^{1:N}, x_t^{1:n-1})$. In an RNN-NADE, the NADE's biases at each time step b_h^t and b_y^t are modified as a function of the RNN's hidden state, and weight matrices W_{bh}^h and W_{bh}^y as follows

$$b_t^h = b_h + W_{bh}^h h_t \tag{2.64}$$

$$b_t^y = b_y + W_{bh}^y h_t. \tag{2.65}$$

The modified biases b_t^h and b_t^y are then plugged into the NADE to model $P(x_{1:N})$. An RNN-NADE is illustrated in Figure 2.8.

2.7 Conclusion

Moving forward, this thesis combines and expands on a number of ideas from this background section, with the goal of making them more adaptive. For instance, multiplicative LSTM (Chapter 3) uses ideas from the mRNN from Section 2.4.1 to make LSTMs 2.3.3 more adaptive at the token level. Dynamic evaluation methods presented in Chapters 4, 5, and 6 incorporate optimizers (Section 2.5) into Transformer (Section 2.4.5.2) and LSTM-based language modeling architectures to allow them to better adapt

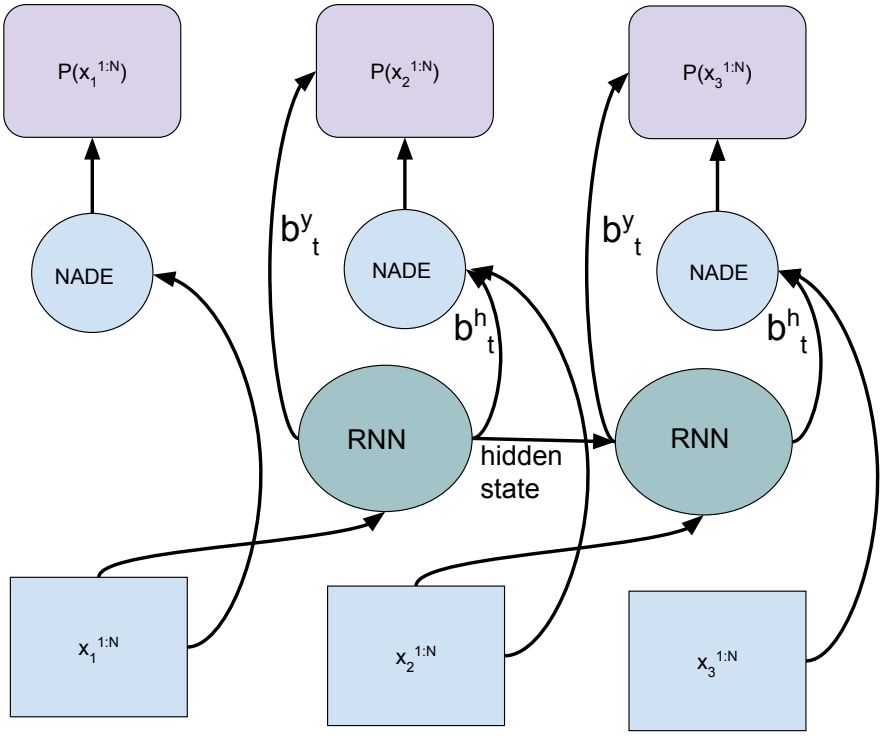


Figure 2.8: An RNN-NADE acts as a probabilistic model of sequences of high dimensional vectors. A NADE is used to model each high dimensional vector $x_t^{1:N}$, where the NADE receives contextual information about $x_{1:t-1}^{1:N}$ from the RNN in the form of hidden biases b_t^h and output biases b_t^y.

to sequences. Chapter 4.8 applies high dimensional sequence modeling (Section 2.6) and dynamic evaluation to improve adaptability in music prediction.

Chapter 3

Multiplicative LSTM for sequence modeling

This chapter studies multiplicative LSTM (mLSTM), a recurrent neural network architecture for sequence modeling that combines the long short-term memory (LSTM) and multiplicative recurrent neural network architectures. mLSTM is characterized by its ability to have different recurrent transition functions for each possible input, making it more adaptive to the input token than a regular LSTM. We demonstrate empirically that mLSTM outperforms standard LSTM and its deep variants for a range of character level language modeling tasks. We also show that mLSTM's advantage over LSTM was greater after a surprising input than it was in general, supporting the claim that language models that can adapt to their inputs are better equipped to recover from surprising tokens.

This ability to adapt to surprising tokens helped mLSTM outperform previous approaches to character level language modeling. A regularized mLSTM achieves 1.27 bits/char on text8 and 1.24 bits/char on enwik8, which were state of the art results at the time the work was done. We also apply an mLSTM on the WikiText-2 dataset to achieve a character level entropy of 1.26 bits/char, corresponding to a word level perplexity of 88.8, which is comparable to word level LSTMs regularized in similar ways on the same task. Work that appears in this chapter is published in two partially overlapping works on mLSTM (Krause et al., 2016, 2017b). It should be noted that multiplicative LSTM was first proposed in a master's thesis by Krause (2015), however experiments and results were very preliminary (for instance, the mLSTM from Krause (2015) achieved 1.69 bits/char on enwik8 vs. 1.24 bits/char in the present chapter), and the main focus of that thesis was to explore Hessian-free optimization (Martens, 2010)

in LSTMs.

3.1 Introduction

RNNs (Section 2.3) can model sequences using a hidden state to summarize past inputs. In a more general formulation of an RNN, the hidden state vector h_t is updated recursively using the previous hidden state vector h_{t-1} and the current input x_t as

$$h_t = \mathcal{F}(h_{t-1}, x_t), \tag{3.1}$$

where $\mathcal{F}$ is a differentiable function with learnable parameters. In a vanilla RNN, $\mathcal{F}$ multiplies its inputs by a matrix and squashes the result with a non-linear function such as a hyperbolic tangent (tanh). The updated hidden state vector is then used to predict a probability distribution over the next sequence element, using function $\mathcal{G}$. In the case where $x_{1:T}$ consists of mutually exclusive discrete outcomes, $\mathcal{G}$ may apply a matrix multiplication followed by a softmax function:

$$P(x_{t+1}) = \mathcal{G}(h_t). \tag{3.2}$$

Auto-regressive RNNs can evaluate log-likelihoods of sequences exactly, and their parameters are differentiable with respect to these log-likelihoods. As noted in Section 2.3, RNNs can be difficult to train due to the vanishing gradient problem (Bengio et al., 1994b), but advances such as the LSTM architecture (Hochreiter and Schmidhuber, 1997; Gers et al., 2000, 2.3.3) have allowed RNNs to be consistently successful. Despite their success, generative RNNs (as well as other conditional generative models) are known to have problems with recovering from mistakes during generation (Graves, 2013), meaning that if the model generates something wrong, and then has to make future preictions based on a wrong input, it can become unstable. While we study language modeling on ground truth text only, models could still have trouble recovering from a surprising input. Each time the recursive function of the RNN is applied and the hidden state is updated, the RNN must decide which information from the previous hidden state to store, due to its limited capacity. If the RNN's hidden representation overwrites important information, which may be especially likely when encountering an unexpected input, it may take many time-steps to recover.

We argue that RNN architectures with hidden-to-hidden transition functions that are input-dependent are better suited to recover from surprising inputs, meaning that the model will make better predictions after encountering an input that it was not expecting.

Our approach to generative RNNs combines LSTM units with multiplicative RNN (mRNN) factorized hidden weights, allowing flexible input-dependent transitions that are easier to control due to the gating units of LSTM . We compare this multiplicative LSTM hybrid architecture with other variants of LSTM on a range of character level language modeling tasks. We find that multiplicative LSTM is able to improve overall prediction performance, and this improvement is especially strong after a surprising input, supporting our hypothesis that RNNs with transition functions that are input-dependent are better able to predict text after a surprising input.

3.2 Input-dependent transition functions

RNNs learn a mapping from previous hidden state h_{t-1} and input x_t to hidden state h_t. Let $\hat{h}_t$ denote the input to the next hidden state before any non-linear operation:

$$\hat{h}(t) = W_{hh}h_{t-1} + W_{hx}x_t, \tag{3.3}$$

where W_{hh} is the hidden-to-hidden weight matrix, and W_{hx} is the input-to-hidden weight matrix. For problems such as language modeling, x_t is a one-hot vector, meaning that the output of $W_{hx}x_t$ is a column in W_{hx}, corresponding to the unit element in x_t.

The possible future hidden states in an RNN with one-hot input data can be viewed as a tree structure, as shown in Figure 3.1. For an alphabet of N inputs and a fixed h_{t-1}, there will be N possible transition functions between h_{t-1} and $\hat{h}_t$. The relative magnitude of $W_{hh}h_{t-1}$ to $W_{hx}x_t$ will need to be large for the RNN to be able to use long range dependencies, and the resulting possible hidden state vectors will therefore be highly correlated across the possible inputs, limiting the effective width of the tree and making it harder for the RNN to form distinct hidden representations for different sequences of inputs. However, if the RNN has flexible input-dependent transition functions, the tree will be able to grow wider more quickly, giving the RNN the flexibility to represent more probability distributions.

In a vanilla RNN, if the contribution to the new hidden states from the inputs is large, the old hidden state will be mostly erased. This makes it difficult to allow inputs to greatly affect the hidden state vector without erasing information. An RNN with the ability to have very different transition functions mappings $h_t \leftarrow h_{t-1}$ for different inputs would allow the relative values of h_t to vary more with each possible input x_t, without overwriting the contribution from the previous hidden state, allowing for more long term information to be stored. This ability to adjust to new inputs quickly

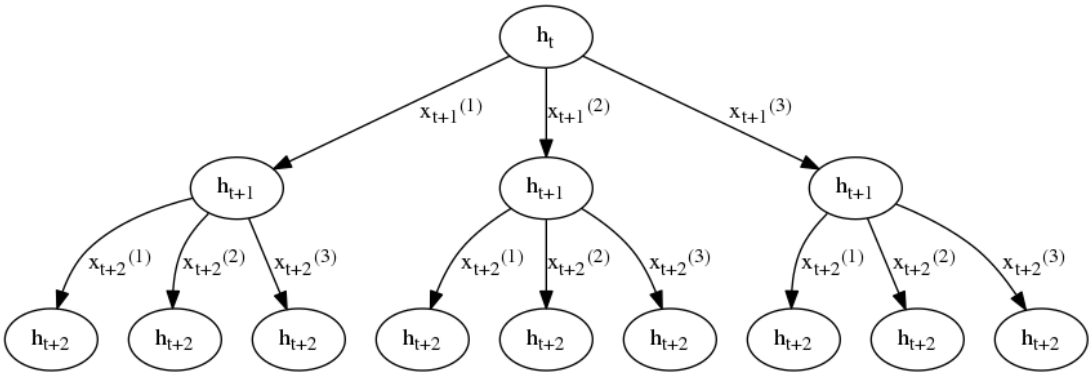


Figure 3.1: Diagram of hidden states of a generative RNN as a tree, where $x_t^{(n)}$ denotes which of N possible inputs is encountered at timestep t. Given h_t, the starting node of the tree, there will be a different possible h_{t+1} for every $x_{t+1}^{(n)}$. Similarly, for every h_{t+1} that can be reached from h_t, there is a different possible h_{t+2} for each $x_{t+2}^{(n)}$, and so on.

while limiting the overwriting of information should make an RNN more robust in its predictions after it encounters surprising inputs, as the hidden vector is less likely to get trapped in a bad numerical state for making future predictions.

An mRNN has the ability to have a very different transition function for each possible input. The effective hidden-to-hidden weight matrix for a particular input, $W_{hh}^{(x_t)}$, is given by

$$W_{hh}^{(x_t)} = W_{hm}\text{diag}(W_{mx}x_t)W_{mh} \tag{3.4}$$

in an mRNN. This architectural feature of an mRNN is a well motivated approach towards building models that are more adaptive to the input token.

3.3 Multiplicative LSTM

The LSTM (Section 2.3.3) and mRNN (Section 2.4.1) architectures both feature multiplicative units, but these units serve different purposes. An LSTM's gates are designed to give the network the ability to preserve information, whereas an mRNN's multiplicative units are designed to allow transition functions to vary across inputs. LSTM gates receive input from both the input units and hidden units, allowing multiplicative interactions between hidden units, but also potentially limiting the extent of input-hidden multiplicative interaction. LSTM gates are also squashed with a sigmoid, forcing them to take values between 0 and 1, which makes them easier to control, but gives them less power to scale hidden states as compared with mRNN's multiplicative units, which

can take any value. For language modeling problems, an mRNN's multiplicative units do not need to be controlled by the network because they are explicitly learned for each input. They are also placed in between a product of 2 dense matrices, giving more flexibility to the possible values of the final product of matrices.

Since the LSTM and mRNN architectures are complimentary, we propose the multiplicative LSTM (mLSTM), a hybrid architecture that combines the factorized hidden-to-hidden transition of mRNNs with the gating framework from LSTMs. The mRNN and LSTM architectures can be combined by adding connections from the mRNN's intermediate state m_t (which is redefined below for convenience) to each gating units in the LSTM, resulting in the following system:

$$m_t = (W_{mx}x_t) \odot (W_{mh}h_{t-1}) \tag{3.5}$$

$$\hat{h}_t = W_{hx}x_t + W_{hm}m_t \tag{3.6}$$

$$i_t = \sigma(W_{ix}x_t + W_{im}m_t) \tag{3.7}$$

$$o_t = \sigma(W_{ox}x_t + W_{om}m_t) \tag{3.8}$$

$$f_t = \sigma(W_{fx}x_t + W_{fm}m_t). \tag{3.9}$$

An mLSTM substitutes m_t in place of h_{t-1} in a normal LSTM. We set the dimensionality of m_t and h_t equal for all our experiments. We also chose to share m_t across all LSTM unit types, resulting in a model with 1.25 times the number of recurrent weights as LSTM for the same number of hidden units.

The goal of this architecture is to combine the flexible input-dependent transitions of mRNNs with the long time lag and information control of LSTMs. The gated units of LSTMs could make it easier to control (or bypass) the complex transitions that result from the factorized hidden weight matrix. The additional sigmoid input and forget gates featured in LSTM units allow even more flexible input-dependent transition functions than in regular mRNNs.

3.4 Related approaches

Many recently proposed RNN architectures use recurrent depth (Section 2.4.2), which is depth between recurrent steps. Recurrent depth allows more non-linearity in the combination of inputs and previous hidden states from every time step, which in turn allows for more flexible input-dependent transitions. Recurrent depth has been found to perform better than other kinds of non-recurrent depth for sequence modeling (Zhang

et al., 2016; Zilly et al., 2017). One problem with recurrent depth is it that it must be processed sequentially, making larger recurrent depths slower to run on a GPU. While adding recurrent depth could improve our model, we believe that maximizing the input-dependent flexibility of the transition function is more important for expressive sequence modeling. Recurrent depth can do this through non-linear layers combining hidden and input contributions, but mLSTM can do this independently of non-linear depth.

Another approach, multiplicative integration LSTMs (Wu et al., 2016, MI-LSTMs, Section 2.4.3), use Hadamard products instead of addition when combining contributions from input and hidden units. The hidden-to-hidden transition in MI-LSTM is given by the following equations

$$\hat{h}_t = W_{hx}x_t + W_{hm}h_t \tag{3.10}$$

$$i_t = \sigma(W_{ix}x_t + W_{im}h_t) \tag{3.11}$$

$$o_t = \sigma(W_{ox}x_t + W_{om}h_t) \tag{3.12}$$

$$f_t = \sigma(W_{fx}x_t + W_{fm}h_t). \tag{3.13}$$

Similarly to MI-LSTM, mLSTM also applies a Hadamard product between a contribution between the inputs and the previous hidden states, but this occurs between the multiplication of two matrices. In the case of LSTM, this allows for transition functions to vary more across inputs, without significantly increasing the size of the model. mLSTM also has normal additive connections from the input layer, which could improve its stability relative to MI-LSTM. mLSTM also has the representational power to model an LSTM exactly; for any LSTM it is possible to mathematically write an mLSTM will make equivalent predictions. This could for instance be done by setting W_{mx} to all ones, W_{mh} to the identity matrix, and weights W_{hm}, W_{im}, W_{om} and W_{fm} in the mLSTM to weights W_{hh}, W_{ih}, W_{oh} and W_{fh} in the LSTM, and setting all other weights equivalent. There is however no mathematical transformation to directly translate an LSTM into an MI-LSTM. The ability for an mLSTM to represent an LSTM may help mLSTM retain LSTM's advantages.

3.5 Experiments

3.5.1 System Setup

Our experiments measure the performance of mLSTM for character-level language modeling tasks of varying complexity[1]. Our initial experiments were mainly designed to compare the convergence and final performance of mLSTM vs LSTM and its deep variants. Our follow up experiments explored training and regularization of mLSTM in more detail, with goal of comparing more directly with the most competitive architectures in the literature.

Our initial and follow up experiments used slightly different set ups; initial experiments used a variant of RMSprop, (Tieleman and Hinton, 2012, Section 2.5.3), with normalized updates in place of a learning rate. All unnormalized update directions v_*, computed by RMSprop, were normalized to have length ℓ, where ℓ was decayed exponentially over training:

$$v \leftarrow \frac{\ell}{\sqrt{v_*^\top v_*}} v_*. \tag{3.14}$$

This update rule would be equivalent to applying gradient norm clipping (Pascanu et al., 2013b, Section 2.3.2), with a learning rate that approaches infinity balanced out by a gradient norm threshold that approaches zero. The initial experiments also used a slightly non-standard version of LSTM (and mLSTM) with the output gate inside of the final tanh of the LSTM cell. This gave us slightly better results in preliminary experiments with very small models, although we later changed the gate order to match what is traditionally used in LSTMs. We use LSTM (RMSprop) and mLSTM (RMSprop) in tables to distinguish results obtained by these initial set of experiments.

For our follow up experiments, we use more standard methodology to be more comparable to the literature. We used Adam (Kingma and Ba, 2014, Section 2.5.4), always starting with an initial learning rate of 0.001 and decaying this linearly to a minimum learning rate (which was always in the range 0.00005 to 0.0001). The mLSTMs used the standard LSTM cell with the output gate outside the tanh. These mLSTMs also used scaled orthogonal initialization (Saxe et al., 2013) for the hidden weights, an initial forget gate bias of 3, and truncated backpropagation lengths from 200 to 250.

We compared mLSTM to previously reported regular LSTM, stacked LSTM, and

[1]Code to replicate our experiments on the enwik8 dataset is available at `https://github.com/benkrause/mLSTM`.

RNN character-level language models. We run detailed experiments on the text8 and enwik8 datasets (Hutter, 2012) to test medium scale character-level language modeling. We test our best model from these experiments on the WikiText-2 dataset (Merity et al., 2017) to measure performance on smaller scale character level language modeling, and to compare with word level models.

3.5.2 Enwik8

We performed experiments using the enwik8 dataset, also known as the Hutter Prize dataset, originally used for the Hutter Prize compression benchmark (Hutter, 2012). This dataset consists mostly of English language text and mark-up language text, but also contains text in other languages, including non-Latin languages. The dataset is modelled using a UTF-8 encoding, and contains 205 unique bytes.

In our initial experiments, we compared mLSTMs and 2-layer stacked LSTMs for varying network sizes, ranging from about 3–20 million parameters. These results all used RMSprop with normalized updates, stopping after 4 epochs on the first 95 million characters, with test performance measured on the last 5 million bytes. Hyperparameters for each mLSTM and stacked LSTM were kept constant across all sizes. The results, shown in Figure 3.2, show that mLSTM gives an improvement across all network sizes.

We hypothesized that mLSTM's superior performance over stacked LSTM was in part due to its ability to recover from surprising inputs. To test this hypothesis, we looked at each network's performance after viewing inputs in the test set that were surprising to the model. We considered a set of the 10% characters with the largest average loss taken by mLSTM and stacked LSTM, and examined losses immediately after these characters. Both networks perform roughly equally on this set of surprising characters, with mLSTM and stacked LSTM taking losses of 6.27 bits/character and 6.29 bits/character respectively. However, stacked LSTM tended to take larger losses than mLSTM in the timesteps immediately following surprising inputs. One to four time-steps after a surprising input occurred, mLSTM and stacked LSTM took average losses of (2.26, 2.04, 1.61, 1.51) and (2.48, 2.25, 1.79, 1.67) bits per character respectively, as shown in Figure 3.3. mLSTM's overall advantage over stacked LSTM was 1.42 bits/char to 1.53 bits/char; mLSTM's advantage over stacked LSTM was greater after a surprising input than it is in general.

We also explore more standard training methodology and regularization methods on this dataset. These experiments all used ADAM, and the standard 90-5-5 training

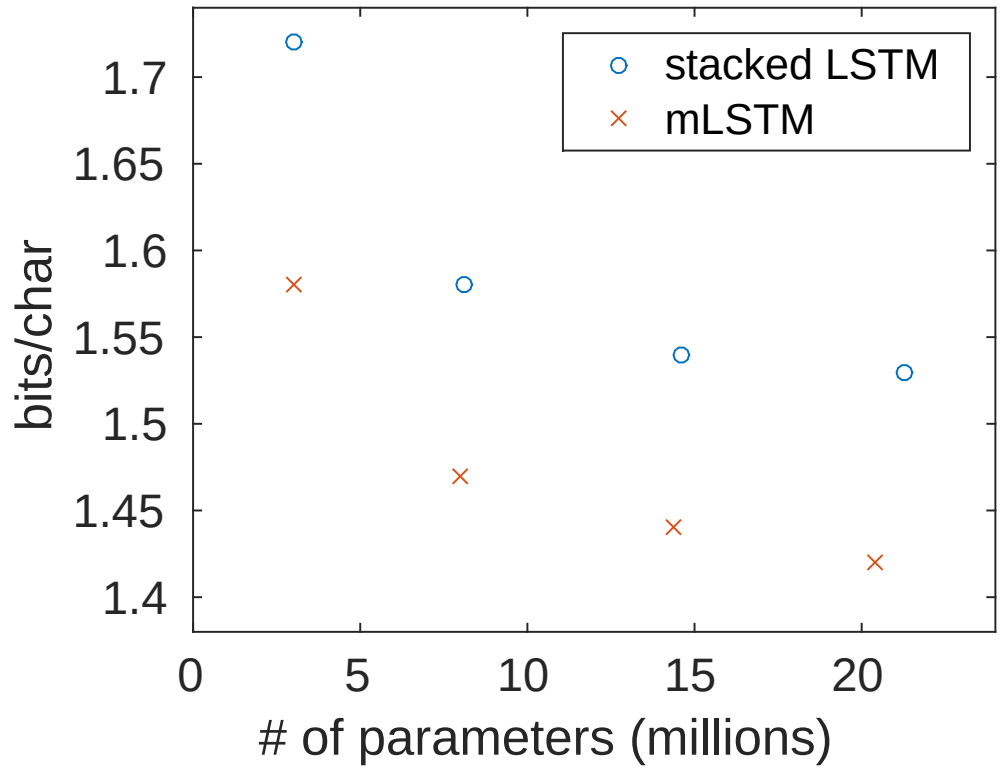


Figure 3.2: enwik8 validation performance in bits/char plotted against number of network parameters for mLSTM and stacked LSTM.

validation test split on this dataset. We firstly consider a standard unregularized mLSTM trained with this methodology. We then experiment with an mLSTM with a linear embedding layer and weight normalization (Salimans and Kingma, 2016) on recurrent weights (mLSTM +emb +WN), which is similar to the mLSTM architecture used in (Radford et al., 2017), which was built off our initial work. We also consider regularization of the later model with variational dropout (Gal and Ghahramani, 2016), which we refer to as mLSTM +emb +WN +VD.

The standard unregularized LSTM used 1900 hidden units and 20 million parameters. The weight normalized mLSTM used 1900 hidden units, and a linear embedding layer of 400, giving it 22 million parameters. The large embedding layer was used because it was found to work well with dropout. Since this embedding layer is linear, it could potentially be removed during test time by multiplying its incoming and outgoing weight matrices to reduce the number of parameters (however we report parameter numbers with the embedding layer). For the regularized weight normalized mLSTM, we apply a variational dropout of 0.2 to the hidden state and to the embedding layer (dropout masks

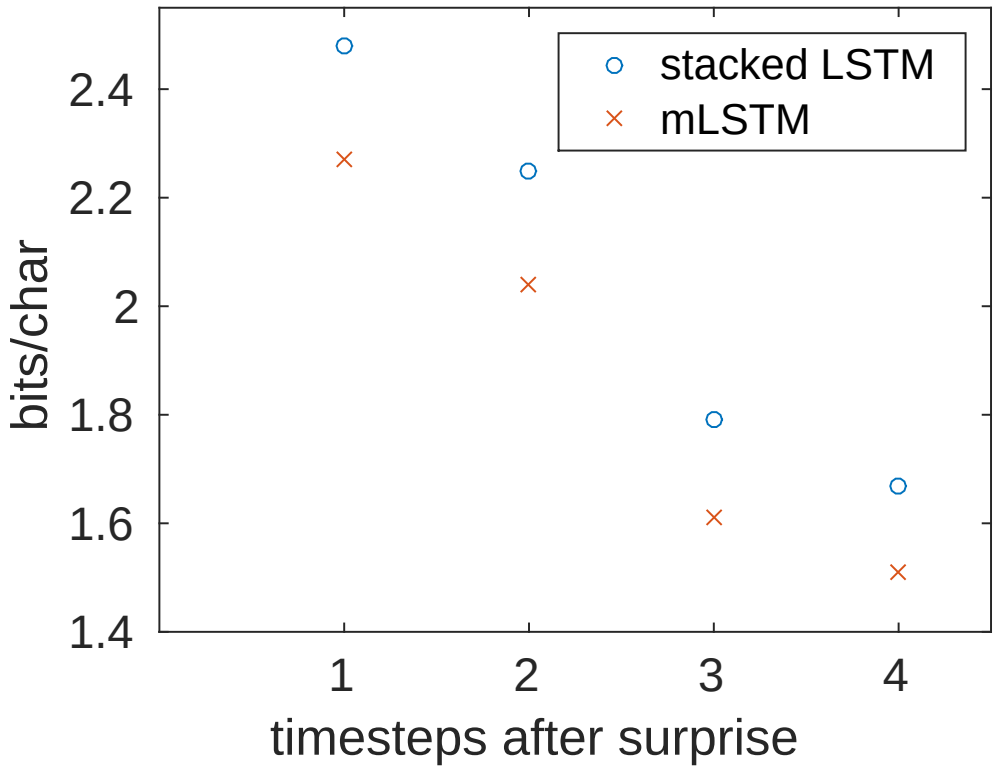


Figure 3.3: Cross entropy loss for mLSTM and stacked LSTM immediately after a surprising input

for both the hidden state and embedding layer were shared across a sequence). We also consider a larger version of the weight normalized mLSTM with 2800 hidden units and 46 million parameters. We increased the dropout in the embedding layer to 0.5 on this model. All results without variational dropout used early stopping on the validation error to reduce overfitting. The results for these experiments are given in Table 3.1.

Interestingly, adding weight normalization and an embedding layer hurt performance in the absence of regularization. However, when combined with variational dropout, this model outperformed all previous static single model neural network results on enwik8. It is worth noting that the LSTM from Melis et al. (2018) used similar regularization and very extensive and methodical hyper-parameter tuning, making it a strong LSTM baseline, which mLSTM is able to improve on.

We also tested an MI-LSTM (Wu et al., 2016, 2.4.3), mLSTM's nearest neighbor, with a slightly larger size (22M parameters) and a very similar hyperparameter configuration and initialization scheme[2] (compared with unregularized mLSTM with no

[2]The only difference in settings was the scale for the orthogonally initialized hidden weights; mLSTM

architecture	# of parameters	test set error
stacked LSTM (7-layer) (Graves, 2013)	21M	1.67
stacked LSTM (7-layer) + dynamic eval (Graves, 2013)	21M	1.33
MI-LSTM (Wu et al., 2016)	17M	1.44
recurrent memory array structures (Rocki, 2016a)		1.40
feedback LSTM + zoneout (Rocki, 2016b)		1.37
hyperLSTM (Ha et al., 2017)	27 M	1.34
hierarchical multiscale LSTM (Chung et al., 2017)		1.32
bytenet decoder (Kalchbrenner et al., 2016)		1.31
LSTM (4 layer) + VD + BB tuning (Melis et al., 2018)	46M	**1.30**
RHN (rec depth 10) + VD (Zilly et al., 2017)	46M	1.27
Fast-slow LSTM (rec depth 4) + zoneout (Mujika et al., 2017)	47M	1.25
unregularized mLSTM (RMS prop, 4 epoch)	20M	1.42
unregularized mLSTM	20M	1.40
mLSTM +emb +WN	22M	1.44
mLSTM +emb +WN +VD	22M	1.28
large mLSTM +emb +WN +VD	46M	**1.24**

Table 3.1: enwik8 dataset test error in bits/char. emb indicates the use of a linear embedding layer, WN indicates weight normalization, and VD indicates variational dropout.

WN). MI-LSTM achieved a relatively poor test set performance of 1.53 bits/char, as compared with 1.40 bits/char for mLSTM under the same settings. The MI-LSTM also converged more slowly, although eventually did require early stopping like the mLSTM. While this particular experiment cannot conclusively prove anything about the relative utility of mLSTM vs. MI-LSTM on this task, it does show that the two architectures are sufficiently different to obtain very different results under the same hyper-parameter settings.

used 0.7 and MI-LSTM used 0.5. We believed this was justified because mLSTM uses a product of two matrices, resulting in a spectral radius of 0.49 for this product. Additionally, reducing the scale to 0.5 improved MI-LSTM's initial convergence rate. Downscaling the orthogonal initializations was necessary in general because an initial forget gate bias of 3 was used.

3.5.3 Text8

Text8 contains 100 million characters of English text taken from Wikipedia in 2006, consisting of just the 26 characters of the English alphabet plus spaces. This dataset can be found at `http://mattmahoney.net/dc/textdata`. This corpus has been widely used to benchmark RNN character level language models, with the first 90 million characters used for training, the next 5 million used for validation, and the final 5 million used for testing. The results of these experiments are shown in Table 3.2.

The first set of experiments we performed were designed to be comparable to those of Zhang et al. (2016), who benchmarked several types of deep LSTMs against shallow LSTMs on this dataset. The shallow LSTM had a hidden state dimensionality of 512, and the deep versions had reduced dimensionality to give them roughly the same number of parameters. Our experiment used an mLSTM with a hidden dimensionality of 450, giving it slightly fewer parameters than the past work, and our own LSTM baseline with hidden dimensionality 512. mLSTM showed an improvement over our baseline and the previously reported best deep LSTM variant.

We also ran experiments to compare a large mLSTM with other reported experiments. We trained an mLSTM with hidden dimensionality of 1900 on the text8 dataset. Unregularized mLSTM was able to fit the training data well and achieved a competitive performance; however it was outperformed by other architectures that are less prone to over-fitting.

We later considered our best training setup from the enwik8 dataset, reusing the exact same architecture and hyper-parameters from this task, with the only difference being the number of input characters (27 for text8), which reduces the number of parameters to around 45 million. This well regularized mLSTM was able to achieve a much stronger performance on text8, tying recurrent highway networks (RHNs) with a recurrent depth of 10 for the best result on this dataset.

3.5.4 WikiText-2

The WikiText-2 dataset (Merity et al., 2017) has been a common benchmark for very recent advances in word-level language modeling. This dataset contains 2 million training tokens and a vocab size of 33k. Documents are given in non-shuffled order, causing the data to contain more long-range dependencies. We use this dataset to benchmark how our advances in character-level language modeling stack up against word level language models. Character language models generally perform worse than word-level language

architecture	test set error
mRNN (Mikolov et al., 2012a)	1.54
MI-LSTM (Wu et al., 2016)	1.44
LSTM (Cooijmans et al., 2017)	1.43
batch normalized LSTM (Cooijmans et al., 2017)	**1.36**
layer-norm hierarchical multiscale LSTM (Chung et al., 2017)	1.29
Recurrent highway networks, rec. depth 10 +VD (Zilly et al., 2017)	1.27
small LSTM (Zhang et al., 2016)	1.65
small deep LSTM (best) (Zhang et al., 2016)	1.63
small LSTM (RMSprop)	**1.64**
small mLSTM (RMSprop)	**1.59**
unregularized mLSTM (RMSprop)	1.40
large mLSTM +emb +WN +VD	**1.27**

Table 3.2: Text8 dataset test set error in bits/char. Architectures labeled with small used a highly restrictive hidden dimensionality (512 for LSTM, 450 for mLSTM).

models on standard word-level English text benchmarks with limited vocabulary sets. One reason for this is word level language models know in advance that every word in the test set will come from a limited vocabulary, whereas character level models model a distribution over all possible words, including out of vocabulary words, making the task inherently more difficult from character level view. Furthermore, very rare words, which character level models are more equipped to handle than word level models, are mapped to an unknown token, making the task artificially biased in a way that benefits word-level language models. From the perspective of training, character level language models must model longer range dependencies, and must learn a more complex non-linear fit to capture joint dependencies between characters. Character level models do have an inherent advantage of being able to capture subword language information, motivating their use on traditionally word-level tasks.

Character level language models can be compared with word level language models by converting bits per character to perplexity. In this case, we model the data in the WikiText-2 train, validation, and test files as raw UTF-8 bytes. The bits per word can be

computed as

$$bits/word = bits/symbol \times \frac{symbols/file}{words/file} \tag{3.15}$$

where in this case, symbols are UTF-8 bytes. The perplexity is then $2^{bits/word}$. The WikiText-2 test set is 245,569 words long, and 1,256,449 bytes long, so each word is on average 5.12 UTF-8 bytes long. A character level model can also assign word level probabilities directly by taking the product of the probabilities of the characters in a word, including the probability of the character ending the word (either a space or a newline). A byte level model is likely at a slight disadvantage compared with word-level because it must predict some information that gets removed during tokenization (such as spaces vs. newlines), but the perplexity given by the conversion above could at least be seen as an upper bound of the word level perplexity such a model could achieve predicting byte by byte. This is because the entropy of the file after tokenization (which word level models measure) will always be less than or equal to the entropy of the file before tokenization (which byte level models measure).

We trained the best mLSTM configuration from the enwik8 dataset, using an embedding layer, weight normalization, and a variational dropout of 0.5 in both the hidden and embedding layer, to model WikiText-2 at the byte level. This model contained 46 million parameters, which is larger than most word level models that use tied input and output embeddings (Press and Wolf, 2017; Inan et al., 2017) to share parameters, but similar in size to untied word level models on this dataset. The results are given in Table 3.3.

architecture	valid	test
LSTM (Grave et al., 2017b)	104.2	99.3
LSTM + VD (untied)(Inan et al., 2017)	98.8	**93.1**
LSTM + VD (tied)(Inan et al., 2017)	91.5	**87.0**
Pointer Sentinel LSTM (Merity et al., 2017)	84.8	80.8
LSTM (tied) + VD + BB tuning (Melis et al., 2018)	69.1	65.9
AWD-LSTM (tied) (Merity et al., 2018b)	68.6	65.8
byte mLSTM +emb +WN +VD	92.8	**88.8**

Table 3.3: WikiText-2 perplexity. The mLSTM operates on UTF-8 bytes, and all previous results are word-level.

Byte mLSTM achieves a byte-level test set cross entropy of 1.2649 bits/char, corresponding to a perplexity of 88.8. Despite all the disadvantages faced by character level models, byte level mLSTM achieves similar word level perplexity to previous word-level LSTM baselines that also use variational dropout for regularization. Byte mLSTM does not perform as well as word-level models that use adaptive add-on methods or recent advances in regularization/hyper-parameter tuning by Merity et al. (2018b) and Melis et al. (2018), however it could likely benefit from these advances as well.

3.6 Conclusion

This chapter developed a new variant of LSTM meant to be more adaptive at the token level, by combining LSTM with mRNN's factorized hidden weights. This mLSTM architecture was motivated by its ability to have both controlled and flexible input-dependent transitions, allowing it to make larger changes to its predictions as a function of a new input.

mLSTM's adaptive ability helped it make better predictions immediately after a surprising input and improved its overall character-level language modeling performance. In a series of character-level language modeling experiments, mLSTM showed improvements over LSTM and its deep variants. mLSTM regularized with variational dropout performed favorably compared with baselines in the literature, outperforming all previous neural models on enwik8 and tying the best previous result on text8. Byte-level mLSTM was also able to perform competitively with word-level language models on WikiText-2. Experiments showed that mLSTM's advantage vs. LSTM was greater after a surprising input.

Unlike many previous architectures for character level language modeling, mLSTM does not use non-linear recurrent depth. All mLSTMs considered in this work only had 2 linear recurrent transition matrices, whereas comparable works such as recurrent highway networks use a recurrent depth of up to 10 to achieve best results. This makes mLSTM more easily parallelizable than these approaches, since far less sequential computation is required per timestep. Additionally, our work suggests that a large depth is not necessary to achieve competitive results on character level language modeling. We hypothesize that mLSTM's ability to have very different transition functions for each possible input, and thus adapt to each input, is what makes it successful at this task. While recurrent depth can accomplish this too, mLSTM can achieve this more efficiently. Our work motivates exploration of adaptive sequence modeling architectures

that use multiplicative interaction and flexible input-dependent transition functions, and suggests application of mLSTM to language generation tasks.

Chapter 4

Dynamic evaluation of recurrent neural networks

This chapter explores dynamic evaluation as a way of making models more adaptive at the sequence level. Dynamic evaluation adapts auto-regressive sequence models to their own predictions using gradient descent. We show that this gives large log-likelihood improvements to language modeling and polyphonic music modeling. Dynamic evaluation is motivated by its ability to assign higher probabilities to re-occurring sequential patterns, making it more robust to surprising patterns that occur in test sequences. Dynamic evaluation's ability to improve LSTMs and mLSTMs supports our claim that language models that can adapt to their inputs can make better predictions (and follow up work in Chapter 5 supports that improvements largely comes from the ability to recover from surprising sequences). The present chapter motivates and develops dynamic evaluation methods that can improve predictions on language modeling and music modeling benchmarks, whereas Chapter 5 analyzes in more depth why these methods work well. Much of the work in this chapter was published in Krause et al. (2018).

4.1 Introduction

Sequence generation and prediction tasks span many modes of data, ranging from audio and language modeling, to more general timeseries prediction tasks. Applications of such models include speech recognition, machine translation, dialogue generation, speech synthesis, forecasting, and music generation. Neural networks can be applied to these tasks by predicting sequence elements one-by-one, conditioning on the history,

thus forming an autoregressive model. RNNs and LSTMs in particular achieved many successes at these tasks. However, in their basic form, these models have a limited ability to adapt to recently observed parts of a sequence.

Many sequences contain repetition; once a pattern occurs in a sequence it is more likely to occur again (Kuhn, 1988; Jelinek et al., 1991; Kuhn and De Mori, 1992). In text, a particular document may tend to repeat certain words, and may contain a particular style associated with the author and topic of the document. In other domains, a sequence of handwriting will generally stay in the same handwriting style and a sequence of speech will generally stay in the same voice, and a sequence of music will tend to have repeating patterns associated with melodies or rhythms. Although RNNs have a hidden state that can summarize the recent past, they have been shown to have problems learning to reproduce sequence elements (Marcus, 2001; Prickett, 2017). In the case of RNN language modeling, augmenting a model with a simple unigram cache improves perplexity (Grave et al., 2017b), demonstrating that RNNs have difficulty using the recent frequency of words in a sequence. Models such as pointer networks (Vinyals et al., 2015), copy nets (Gu et al., 2016), pointer-sentinel RNNs (Merity et al., 2017) and the neural cache method (Grave et al., 2017b) allows models to "point" to inputs in the sequence history use them directly as outputs, thus enabling them to more naturally handle "direct repetitions" in a sequence, where a symbol repeats itself. However, such approaches do not model "indirect repetitions", when a synonym, inflectional variant, or word otherwise related to a recently-occurring word appears. More broadly, it is desirable for an adaptive model to be able to capture deeper regularities such as topic or style.

This chapter examines *dynamic evaluation* (Mikolov et al., 2010; Mikolov, 2012; Graves, 2013), which adapts models to recent sequences using gradient descent, as a way to model re-occurring sequential patterns. Previous work using dynamic evaluation did not explore or describe its methodology in depth, and had mixed results. In contrast, our work develops a method and tuning procedure to consistently obtain strong results (Section 4.5), and uses this approach to outperform previously reported results in word and character-level language modeling (Section 4.7). Furthermore, we design a method to dramatically reduce the number of adaptation parameters in dynamic evaluation, making it practical in a wider range of settings (Section 4.6). We also show dynamic evaluation has a scope beyond text, applying it to improve music prediction (Section 4.8).

4.2 Motivation

As reviewed in Section 2.1.1, generative models can assign a probability to a sequence $x_{1:T} = \{x_1, \ldots, x_T\}$ by factorizing it as

$$P(x_{1:T}) = P(x_1) \prod_{t=2}^{T} P(x_t | x_{1:t-1}). \tag{4.1}$$

Methods that apply this factorization either use a fixed context when predicting $P(x_t|x_{1:t-1})$, or use a recurrent hidden state to summarize the context, as in an RNN. However, for longer sequences, the history $x_{1:t-1}$ often contains re-occurring patterns that are difficult to capture using static models with fixed parameters. RNNs have a memory capacity that is limited by their hidden state, which can make remembering these regularities difficult. Furthermore, RNNs have no inherent inductive bias that patterns in sequences tend to re-occur; for instance, randomly initialized RNN would not be able to predict even very repetitive sequences. The ability to model re-occurring sequential patterns would need to learned from using re-occurring sequential patterns in training to help reduce the training loss. This could be difficult to generalize to unseen data, and the re-occurring sequential patterns in held out sequences will not always have occurred in the training set.

In a dataset of sequences $\{x^1_{1:T}, x^2_{1:T}, ..., x^M_{1:T}\}$, each sequence $x^i_{1:T}$ could be viewed as being generated from a slightly different local distribution $P(x^i_{1:T})$. At any point in time t, the history of a sequence $x^i_{1:t-1}$ contains useful information about the generating distribution for that specific sequence. Ideally, we would like to assign higher probabilities to sequences that have a consistent local distribution, and lower probabilities to sequences that do not. Therefore we aim to adapt the global model parameters θ_g learned during training, by inferring a set of local model parameters θ_l from $x^i_{1:t-1}$ that will better approximate $P(x^i_t | x^i_{1:t-1})$.

The generating distribution may change continuously across a single sequence; for instance, a text excerpt may change topic. Furthermore, many sequence modeling problems do not distinguish between sequence boundaries, and concatenate all sequences into one continuous sequence. Thus, many sequence modeling tasks can be viewed as having a local distribution $P_l(x)$ as well as a global distribution $P_g(x) := \int P(l) P_l(x)\, dl$. When training, the goal is to find the best fixed model possible for $P_g(x)$. However, at evaluation, a model that can infer the current $P_l(x)$ from the recent history has an advantage.

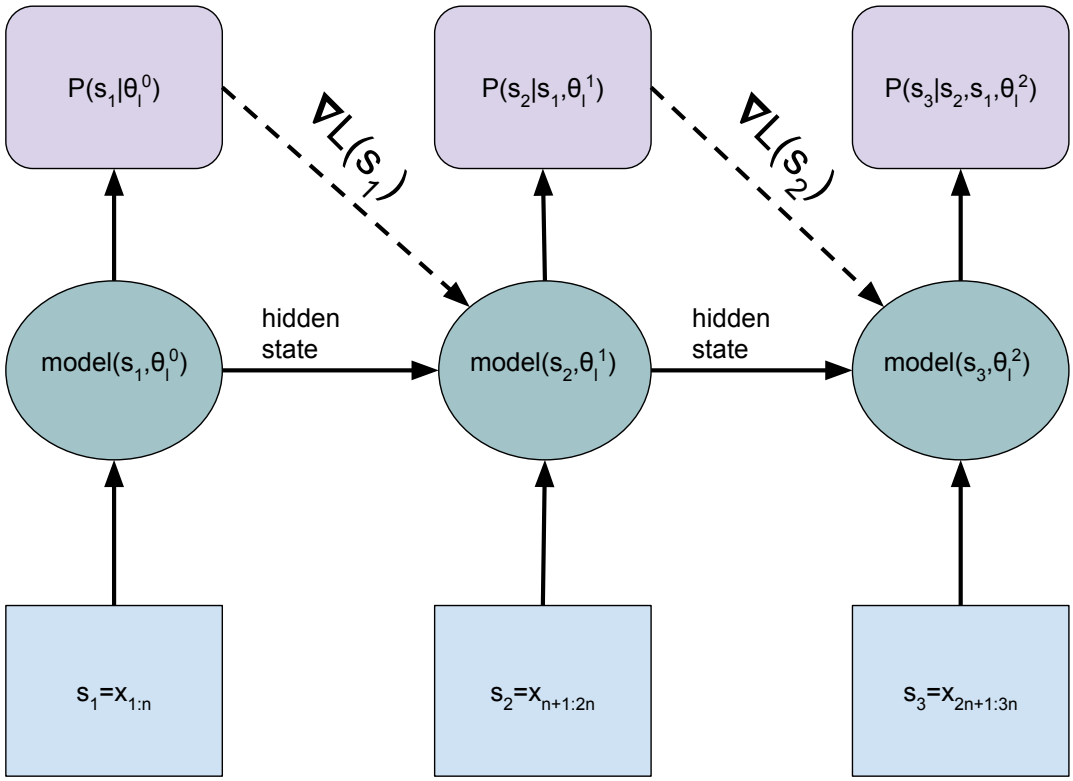


Figure 4.1: Illustration of dynamic evaluation. The model evaluates the probability of sequence segments s_i. The gradient $\nabla\mathcal{L}(s_i)$ with respect to the log probability of s_i is used to update the model parameters θ_l^{i-1} to θ_l^i before the model progresses to the next sequence segment. Dashed edges are what distinguish dynamic evaluation from static (normal) evaluation.

4.3 Dynamic evaluation

Dynamic evaluation is a test time modification to auto-regressive sequence models adapts the model parameters learned at training time, θ_g, to the models predictions. When assigning probabilities to sequences, the model is adapted to the recent sequence history. The goal is to learn adapted parameters θ_l that provide a better model of the local sequence distribution, $P_l(x)$. In this work, we apply dynamic evaluation by splitting a long test sequence $x_{1:T}$ into a sequence, $s_{1:M}$, of shorter sequence segments s_i of length n:

$$s_{1:M} = \{s_1 = x_{1:n}, s_2 = x_{n+1:2n}, \ldots, s_M\}. \tag{4.2}$$

The initial adapted parameters θ_l^0 are set to θ_g, and used to compute the probability of the first segment, $P(s_1|\theta_l^0)$. This probability gives a cross entropy loss $\mathcal{L}(s_1)$, with gradient $\nabla\mathcal{L}(s_1)$, which is computed using the truncated back-propagation through time (we

use the variant in Algorithm 1 which reuses the hidden state from previous segments). The gradient $\nabla\mathcal{L}(s_1)$ is used to update the model, resulting in adapted parameters θ_l^1, before evaluating $P(s_2|\theta_l^1)$. The same procedure is then repeated for s_2, and for each s_i (Figure 4.1). Gradients for each loss $\mathcal{L}(s_i)$ are only backpropagated to the beginning of s_i, so the computation is linear in the sequence length. Each update applies one maximum likelihood training step to approximate the current local distribution $P_l(x)$. The computational cost of dynamic evaluation is thus one forward pass and one gradient computation through the data, with an additional small overhead to apply the update rule for every sequence segment.

As in all autoregressive models, dynamic evaluation only conditions on sequence elements that it has already predicted, and so evaluates a valid log-probability for each sequence. Dynamic evaluation will assign higher probabilities to sequences where adapting to the sequences history helps, and lower probabilities to sequences where it does not help, but the sum of probabilities over all sequences will always sum to one. Dynamic evaluation can also be used while generating sequences (which we explore in a preliminary way in the next chapter). In this case, the model generates each sequence segment s_i using fixed weights, and performs a gradient descent based update step on $\mathcal{L}(s_i)$. Applying dynamic evaluation for sequence generation could result in generated sequences with more consistent regularities, meaning that patterns that occur in the generated sequence are more likely to occur again.

4.4 Background

Prior to this work, adaptive n-grams, neural cache methods, and earlier versions of dynamic evaluation had been considered for adaptive language modeling (see Section 2.2 for a review). At the time of publication of this work, the neural cache as applied by Merity et al. (2018b) to AWD-LSTMs was the state of the art method for adaptive language models. As mentioned in Section 2.2.2, dynamic evaluation was proposed by Mikolov et al. (2010), but had been only been considered as an aside and not explored in depth prior to this work.

4.5 Dynamic evaluation methodology

We propose several changes to Mikolov et al. (2010)'s dynamic evaluation update rule with SGD and fully truncated backpropagation, which we refer to as traditional dynamic

evaluation. The first modification reduces the update frequency, so that gradients are backpropagated over more timesteps. This change provides more accurate gradient information, and also improves the computational efficiency of dynamic evaluation, since the update rule is applied much less often. We use sequence segments of length $n = 5$ for word-level tasks and $n = 20$ for character-level tasks (see Equation 4.2), whereas the method of Mikolov et al. (2010) was equivalent to always setting $n = 1$.

Next, we add a decay prior to bias the model towards the parameters θ_g learned during training. Our motivation for dynamic evaluation assumes that the local generating distribution $P_l(x)$ is constantly changing, so it is potentially desirable to weight recent sequence history higher in adaptation. Adding a decay prior accomplishes this by causing previous adaptation updates to decay exponentially over time. The use of a decay prior for dynamic evaluation relates to an update rule used for fast weights (Ba et al., 2016a), which decayed fast weights towards zero. For SGD with a decay prior, learning rate η, and decay rate λ we form the update rule

$$\theta_l^i \leftarrow \theta_l^{i-1} - \eta \nabla \mathcal{L}(s_i) + \lambda(\theta_g - \theta_l^{i-1}). \tag{4.3}$$

We then consider using an update rule related to RMSprop (Tieleman and Hinton, 2012, Section 2.5.3) in place of SGD. RMSprop uses a moving average of recent squared gradients to scale learning rates for each weight. In our proposed dynamic evaluation update rule, we collect mean squared gradients, MS_g, on the training data rather than on recent test data (for a justification of why we do this, see Section 5.4). We refer to this approach as global RMS to distinguish it from RMSprop. MS_g[1] is given by

$$MS_g = \frac{1}{N_b} \sum_{k=1}^{N_b} (\nabla \mathcal{L}_k)^2, \tag{4.4}$$

where N_b is the number of training batches and $\nabla \mathcal{L}_k$ is the gradient on the kth training batch. The mini-batch size for this computation becomes a hyper-parameter, as larger mini-batches will result in smaller mean squared gradients. The update rule, which we call global RMS with a decay prior in our experiments, is then

$$\theta_l^i \leftarrow \theta_l^{i-1} - \eta \frac{\nabla \mathcal{L}(s_i)}{\sqrt{MS_g} + \varepsilon} + \lambda(\theta_g - \theta_l^{i-1}), \tag{4.5}$$

where ε is a stabilization parameter. For the decay step of our update rule, we also scale the decay rate for each parameter proportionally to $\sqrt{MS_g}$, since parameters with a high

[1] The exact scaling of MS_g (for instance, whether it is a sum or average) affects the hyperparameters but do not change the optimizer in principle. Some publicly available dynamic evaluation implementations scale MS_g differently from Equation 4.4

RMS gradient affect the dynamics of the network more. RMS_{norm} is $\sqrt{MS_g}$ divided by its mean, resulting in a normalized version of $\sqrt{MS_g}$ with a mean of 1:

$$RMS_{\text{norm}} = \frac{\sqrt{MS_g}}{\text{avg}(\sqrt{MS_g})}. \tag{4.6}$$

We clip the values of RMS_{norm} to be no greater than $1/\lambda$ to ensure that the decay rate does not exceed 1 for any parameter. Combining the learning component and the regularization component results in the final update equation, which we refer to as global RMS with an RMS decay prior

$$\theta_l^i \leftarrow \theta_l^{i-1} - \eta \frac{\nabla \mathcal{L}(s_i)}{\sqrt{MS_g} + \varepsilon} + \lambda(\theta_g - \theta_l^{i-1}) \odot RMS_{\text{norm}}. \tag{4.7}$$

This chapter contains experiments with each of the proposed modifications to the update rule, whereas Chapter 5 (Section 5.4) presents more rigorous experimental justifications for some of the design choices for the optimizer presented here, in comparison with more general deep learning optimizers such as RMSprop and Adam (Kingma and Ba, 2014, Section 2.5.4).

Hyper-parameter tuning: Regardless of update rule, we found it was important to properly tune the hyper-parameters of dynamic evaluation. As in the neural cache model, this hyper-parameter tuning procedure applied a post training step in which the model was dynamically evaluated over different hyper-parameter settings. We found tuning the learning rate was by far the most important, however we also found a small benefit to tuning the decay parameter. Hyper-parameter tuning for dynamic evaluation is much faster than hyper-parameter tuning for general training, because it requires a single pass through the validation set per setting. We also found that it is possible to use a small subset of the validation set to tune hyper-parameters and achieve a similar performance. We suspect that poor hyper-parameter tuning is why past dynamic evaluation results have been mixed. For instance, Sprechmann et al. (2018) reported using dynamic evaluation with optimization parameters obtained during training, and achieved minimal test time improvements.

4.6 Sparse dynamic evaluation

Mini-batching over sequences is desirable for some test-time applications because it allows faster processing of multiple sequences in parallel. Dynamic evaluation has a high memory cost for mini-batching because it is necessary to store a different set

of parameters for each sequence in the mini-batch. Therefore, we consider a sparse dynamic evaluation variant that updates a smaller number of parameters. We introduce a new adaptation matrix $\mathcal{M}$ which is initialized to zeros. $\mathcal{M}$ multiplies hidden state vector h_t of an RNN at every time-step to get a new hidden state h'_t, via

$$h'_t = h_t + \mathcal{M}h_t. \tag{4.8}$$

h'_t then replaces h_t and is propagated throughout the network via both recurrent and feed-forward connections. In a stacked RNN, this formulation could be applied to every layer or just one layer. Applying dynamic evaluation to $\mathcal{M}$ avoids the need to apply dynamic evaluation to the original parameters of the network, reduces the number of adaptation parameters, and makes mini-batching less memory intensive. We reduce the number of adaptation parameters further by only using $\mathcal{M}$ to transform an arbitrary subset of H hidden units. This results in $\mathcal{M}$ being an $H \times H$ matrix with $d = H^2$ adaptation parameters. If H is chosen to be much smaller than the number of hidden units, this reduces the number of adaptation parameters dramatically.

4.7 Language modeling experiments

We applied dynamic evaluation to word- and character-level language modeling[2]. After training the base model, we tune hyper-parameters for dynamic evaluation on the validation set, and evaluate both the static and dynamic versions of the model on the test set. We also analyze the sequence lengths for which dynamic evaluation is useful, and investigate how dynamic evaluation can generalize to related words.

4.7.1 Small scale word-level language modeling

We performed word-level language modeling experiments on the Penn Treebank (PTB, Marcus et al., 1993) and WikiText-2 (Merity et al., 2017) datasets. These experiments compared the performance of static and dynamic evaluation, different dynamic evaluation variants, and the neural cache.

The PTB language modeling dataset, which is derived from Wall Street Journal articles, contains 929k training tokens with a vocabulary limited to 10k words. WikiText-2 is roughly twice the size of PTB, with 2 million training tokens and a vocabulary size

[2]code available at `https://github.com/benkrause/dynamic-evaluation`

model	parameters	valid	test
RNN+LDA+kN-5+cache (Mikolov and Zweig, 2012)			92.0
CharCNN (Kim et al., 2016)	19M		78.9
LSTM (Zaremba et al., 2014)	66M	82.2	78.4
Variational LSTM (Gal and Ghahramani, 2016)	66M		73.4
Pointer sentinel-LSTM (Merity et al., 2017)	21M	72.4	70.9
Variational LSTM + augmented loss (Inan et al., 2017)	51M	71.1	68.5
Variational RHN (Zilly et al., 2017)	23M	67.9	65.4
NAS cell (Zoph and Le, 2017)	54M		62.4
Variational LSTM + gradual learning (Aharoni et al., 2017)	105M		61.7
LSTM + BB tuning (Melis et al., 2018)	24M	60.9	58.3
LSTM (Grave et al., 2017b)		86.9	82.3
LSTM + neural cache (Grave et al., 2017b)		74.6	72.1
AWD-LSTM (Merity et al., 2018b)	24M	60.0	57.3
AWD-LSTM + neural cache (Merity et al., 2018b)	24M	53.9	**52.8**
AWD-LSTM (rerun)	24M	59.8	**57.7**
AWD-LSTM + traditional dynamic eval (sgd, bptt=1)	24M	54.9	53.5
AWD-LSTM + dynamic eval (sgd, bptt=5)	24M	54.7	53.3
AWD-LSTM + dynamic eval (sgd, bptt=5, decay)	24M	54.0	52.4
AWD-LSTM + dynamic eval (global RMS, bptt=5, decay)	24M	52.7	52.0
AWD-LSTM + dynamic eval (global RMS, bptt=5, RMS decay)	24M	51.6	**51.1**
AWD-QRNN (rerun)	24M	59.2	**56.7**
AWD-QRNN + dynamic eval (global RMS, bptt=20, RMS decay)	24M	51.4	**50.5**

Table 4.1: Penn Treebank perplexities. bptt refers to sequence segment lengths. The bold lines show that dynamic evaluation gives a large improvement over a state-of-the-art static model. Each of our other contributions leads to further improvements.

of 33k. It features articles in a non-shuffled order, with dependencies across articles that adaptive methods should be able to exploit.

For our baseline model, we use the previous state-of-the-art averaged SGD (ASGD) weight-dropped LSTM (AWD-LSTM, Merity et al., 2018b). We wanted to ensure a strong starting baseline to increase the likelihood that any improvements given by dynamic evaluation would generalize to other models. Furthermore, Merity et al. (2018b) implemented the neural cache on top of their approach, making it easy for us to compare dynamic evaluation with the neural cache using the same starting settings. The AWD-LSTM is a vanilla LSTM that combines the use of drop-connect (Wan et al., 2013) on recurrent weights for regularization, and a variant of ASGD (Polyak and Juditsky, 1992) for optimization. Our model used 3 layers and tied input and output embeddings (Press and Wolf, 2017; Inan et al., 2017, Section 2.4.6), and was intended to be a direct replication of AWD-LSTM. Merity et al. (2018b) later added results using an AWD quasi-recurrent neural network (AWD-QRNN, Bradbury et al., 2017), so we (later) applied dynamic evaluation to this model as well to demonstrate that dynamic evaluation could work using a different starting model.

We experiment with traditional dynamic evaluation, as well as each modification we make building up to our final update rule as described in Section 4.5. We also apply our proposed hyper-parameter tuning scheme to all dynamic evaluation methods. Results for PTB are given in Table 4.1, and results for WikiText-2 are given in Table 4.2. As our final update rule (global RMS + RMS decay, Equation 4.7) worked best, we use this for future experiments and use "dynamic eval" by default to refer to this update rule in tables.

All dynamic evaluation variants give large improvements to both datasets. We demonstrate much larger improvements on PTB even with traditional dynamic evaluation than some past work (Mikolov, 2012), highlighting the importance of using our proposed hyper-parameter tuning scheme. Our most advanced dynamic evaluation variant achieves better final results than the neural cache, improving the state-of-the-art on PTB and WikiText-2. This improvement is especially pronounced on WikiText-2, suggesting that dynamic evaluation is exploiting the rich vocabulary or the non-shuffled order of documents. Since publishing our code, the state-of-the-art on PTB and WikiText-2 has been further improved by applying our dynamic evaluation implementation on top of an AWD-LSTM with a mixture of softmaxes output layer (Yang et al., 2018).

model	parameters	valid	test
Byte mLSTM (Chapter 3)	46M	92.8	88.8
Variational LSTM (Inan et al., 2017)	28M	91.5	87.0
Pointer sentinel-LSTM (Merity et al., 2017)		84.8	80.8
LSTM + BB tuning (Melis et al., 2018)	24M	69.1	65.9
LSTM (Grave et al., 2017b)		104.2	99.3
LSTM + neural cache (Grave et al., 2017b)		72.1	68.9
AWD-LSTM (Merity et al., 2018b)	33M	68.6	65.8
AWD-LSTM + neural cache (Merity et al., 2018b)	33M	53.8	**52.0**
AWD-LSTM (rerun)	33M	68.9	**66.1**
AWD-LSTM + traditional dynamic eval (sgd, bptt=1)	33M	51.6	49.0
AWD-LSTM + dynamic eval (sgd, bptt=5)	33M	51.5	48.8
AWD-LSTM + dynamic eval (sgd, bptt=5, decay)	33M	49.8	47.4
AWD-LSTM + dynamic eval (global RMS, bptt=5, decay)	33M	46.9	44.7
AWD-LSTM + dynamic eval (global RMS, bptt=5, RMS decay)	33M	46.4	**44.3**
AWD-QRNN (rerun)	33M	68.3	**66.0**
AWD-QRNN + dynamic eval (global RMS, bptt=20, RMS decay)	33M	45.9	**44.0**

Table 4.2: WikiText-2 perplexities.

4.7.2 Medium scale word-level language modeling

We benchmarked the performance of dynamic evaluation against static evaluation and the neural cache on the larger word-level text8 dataset. Text8 is often used as a character-level language modeling benchmark (as it is throughout this thesis). A word-level version of the dataset was introduced by Mikolov et al. (2014), who preprocessed the data by mapping rare words to an "unknown" token, resulting in a vocabulary of 44k and 17M training tokens. We use the same test set as in Mikolov et al. (2014), but also hold out the final 100k training tokens as a validation set to allow for fair hyper-parameter tuning (the original task did not have a validation set). We trained an AWD-LSTM with 52M parameters using the implementation from Merity et al. (2018b), and compared the performance of static evaluation, dynamic evaluation, and neural caching at test time.

We used the hyper-parameter settings for dynamic evaluation found on PTB, and only tuned the learning rate (to 2 significant figures). The neural cache uses 3 hyper-parameters: the cache length, a mixing parameter and a flatness parameter. Starting from a cache size of 3000, we used a series of grid searches to find optimal values for

the mixing parameter and flatness parameter (to 2 significant figures). We found that the affect of varying the cache size in the range 2000–4000 was negligible, so we kept the cache size at 3000. Results are given in Table 4.3, with the results from Grave et al. (2017b) that used the same test set given for context.

model	valid	test
LSTM (Grave et al., 2017b)		121.8
LSTM + neural cache (Grave et al., 2017b)	-	99.9
AWD-LSTM	80.0	**87.5**
AWD-LSTM + neural cache	67.5	**75.1**
AWD-LSTM + dynamic eval	63.3	**70.3**

Table 4.3: text8 (word-level) perplexities

Dynamic evaluation soundly outperforms static evaluation and the neural cache method, demonstrating that the benefits of dynamic evaluation are maintained when using a stronger model with more training data.

4.7.3 Character-level language modeling

We consider dynamic evaluation for character-level language modeling using the (character-level) text8 and enwik8 data sets. Both of these datasets are 100M characters long, with a 90:5:5 split for training, validation, and testing. enwik8 is comprised of raw Wikipedia and contains XML and special characters, whereas text8 contains preprocessed Wikipedia that is lowercased and limited to 26 characters of English text plus spaces. We used the mLSTM from Chapter 3 as our base model for both datasets, as it was the strongest baseline on these datasets at the time of writing. More details about the base model and data sets can be found in Chapter 3. As in the word level experiments, we tune the hyperparameters of dynamic evaluation on the validation set before evaluating on the test set.

We also used sparse dynamic evaluation (Section 4.6) on the enwik8 dataset. In this case, we adapted a subset of 500 hidden units, resulting in a 500×500 adaptation matrix and 250k adaptation parameters. Our mLSTM only contained one recurrent layer, so only one adaptation matrix was needed. All of our dynamic evaluation results in this section use the final update rule given in Section 4.5. Results for enwik8 are given in Table 4.4, and results for text8 are given in Table 4.5.

model	parameters	test
Stacked LSTM (Graves, 2013)	21M	1.67
Stacked LSTM + traditional dynamic eval (Graves, 2013)	21M	1.33
Multiplicative integration LSTM (Wu et al., 2016)	17M	1.44
HyperLSTM (Ha et al., 2017)	27M	1.34
Hierarchical multiscale LSTM (Chung et al., 2017)		1.32
Bytenet decoder (Kalchbrenner et al., 2016)		1.31
LSTM + BB tuning (Melis et al., 2018)	46M	1.30
Recurrent highway networks (Zilly et al., 2017)	46M	1.27
Fast-slow LSTM (Mujika et al., 2017)	47M	1.25
mLSTM (Section 3.5.2)	46M	**1.24**
mLSTM + sparse dynamic eval ($d = 250k$)	46M	1.13
mLSTM + dynamic eval	46M	**1.08**

Table 4.4: enwik8 test set error in bits/char.

Dynamic evaluation achieves large improvements to our base models and state-of-the-art results on both datasets. Sparse dynamic evaluation also achieves significant improvements on enwik8 using only 0.5% of the adaptation parameters of regular dynamic evaluation.

4.8 Music modeling experiments

Our next set of experiments examine dynamic evaluation in the setting of probabilistic models of polyphonic music. While evaulatue our models exclusively using log-likelihood, on these models can be applied to music generation, where the model generates sequences of musical notes, and polyphonic transcription, where the model can act as a symbolic prior for a system that transcribes musical notation from audio. Data are represented as 88 dimensional binary vectors at each timestep, with one element for each possible piano note from A0 to C8. This task can be framed as high dimensional sequence modeling (Section 4.8), since multiple notes can be present in a single timestep. Our task and evaluation followed Boulanger-Lewandowski et al. (2012),

model	parameters	test
Multiplicative RNN (Mikolov et al., 2012b)	5M	1.54
Multiplicative integration LSTM (Wu et al., 2016)	4M	1.44
LSTM (Cooijmans et al., 2017)		1.43
Batch normalized LSTM (Cooijmans et al., 2017)		1.36
Hierarchical multiscale LSTM (Chung et al., 2017)		1.29
Recurrent highway networks (Zilly et al., 2017)	45M	1.27
mLSTM (Section 3.5.3)	45M	**1.27**
mLSTM + dynamic eval	45M	**1.19**

Table 4.5: text8 (char-level) test set error in bits/char.

evaluating on the preprocessed versions of four MIDI datasets[3]:

- Piano-midi.de is a classical piano MIDI archive that was split according to Poliner and Ellis (2006)
- Nottingham is a collection of 1200 folk tunes with chords instantiated from the ABC format.
- MuseData is an electronic library of orchestral and piano classical music from CCARH4.
- JSB chorales is a corpus of 382 fourpart harmonized chorales by J. S. Bach with the split of Allan and Williams (2005).

Our experiments are designed to compare our baseline model with static evaluation vs. dynamic evaluation on the 4 music datasets. All experiments use an LSTM-NADE hybrid as the baseline model. RNN-NADEs are described in Chapter 2.6.2, and an LSTM-NADE substitutes LSTMs for vanilla RNNs in this architecture. SGD with gradient norm clipping was used for optimization on the training sets. Recurrent dropout in the style of Zaremba et al. (2014) was applied in the LSTM layers, and dropout was also applied to NADE units separately at each timestep. Our network for all tasks consisted of a linear embedding layer, 2 stacked layers of 650 LSTM units, and a NADE

[3] http://www-etud.iro.umontreal.ca/~boulanni/icml2012

of 400 units. This resulted in a network with 7.2M total parameters. Hyperparameters for training were tuned to give the best validation performance on each task.

After the baseline models were trained, dynamic evaluation was applied. We applied SGD dynamic evaluation, as well as the global RMS dynamic evaluation variant presented in Equation 4.7. Unlike in language modeling, where the validation and test sets are made up of groups of articles concatenated together, the songs in the validation and test sets of these music corpora contain discrete boundaries. Therefore, the weights of dynamic evaluation (and hidden states of the model) were reset at the end of each song. Since we would expect local dependencies present in a song to generally be present throughout the song, we set the decay rate to zero for these experiments. Other hyperparameters for dynamic evaluation are tuned on the validation set. The results of the baseline models with static evaluation, SGD dynamic evaluation, and global RMS dynamic evaluation are given in Table 4.6.

model	Piano-midi	Nottingham	Muse Data	JSB Chorales
NADE (1 frame only) (Boulanger-Lewandowski et al., 2012)	-10.28	-5.48	-10.06	-7.19
RNN (HF) (Boulanger-Lewandowski et al., 2012)	-7.66	-3.89	-7.19	-8.58
RNN-NADE (HF) (Boulanger-Lewandowski et al., 2012)	-7.05	-2.31	-5.60	-5.56
RNN-RBM (HF) (Boulanger-Lewandowski et al., 2012)	-7.09	-2.39	-6.01	-6.27
LSTM-NADE (Johnson, 2017)	-7.39	-2.06	-5.03	-6.10
TP-LSTM-NADE (Johnson, 2017)	-5.49	-1.64	-4.34	-5.92
BA-LSTM (Johnson, 2017)	-5.00	-1.62	-4.41	-5.86
LSTM-NADE	**-6.84**	**-1.96**	**-5.35**	**-5.28**
LSTM-NADE + dynamic eval (SGD)	-5.57	-1.30	-4.58	-5.14
LSTM-NADE + dynamic eval (global RMS)	**-5.47**	**-1.27**	**-4.48**	**-5.11**

Table 4.6: Performance of models in log-likelihood per frame on four music prediction tasks.

Overall, dynamic evaluation gave large improvements on three out of four datasets. The sequences in JSB Chorales were shorter than in the other three datasets, meaning that dynamic evaluation had less data to adapt to, which may partially explain why it did not perform as well. Dynamic evaluation performed especially well on the Nottingham dataset, which contains sequences with many repeating motifs. The biaxial LSTM (BA-LSTM) and tied-parallel LSTM-NADE (TP LSTM-NADE) of Johnson (2017) are designed to give the model inductive biases to allow them to better capture the relative relationships between musical notes. Our models do not have these inductive biases, and still perform better on some datasets. Dynamic evaluation could be applied directly

to the BA-LSTM and TP LSTM-NADE as well, potentially resulting in larger gains in performance.

4.9 Conclusion

This chapter motivates dynamic evaluation as way of adapting auto-regressive sequence models to their own predictions to assign higher probabilities to sequences with re-occurring sequential patterns, thus allowing models to adapt to longer sequences. We develop a novel dynamic evaluation approach, and perform experiments showing that the proposed approach gives large test time improvements across character and word level language modeling, as well as polyphonic music prediction. This chapter shows how making strong RNN language modeling baselines more adaptive at the sequence level can improve their predictions. The following chapter explores in more depth how and why dynamic evaluation works well, and provides evidence that dynamically evaluated models achieve better recovery from surprising sequences.

Chapter 5

Understanding the generalization ability of adaptive language models

The previous chapter presented an approach to dynamic evaluation, motivated by its ability to adapt to the recent sequence history, that could give large improvements to language modeling. This chapter explores why this adaptation works well, and in what context it gains an advantage. Supporting our thesis claim, we find evidence that the adaptation ability of dynamic evaluation makes it especially robust to surprising sequences. For instance, Section 5.1 shows that dynamic evaluation is able to gain a large advantage when the test sequence is in a different language from the training data by exploiting re-occurring statistical patterns in the language. Further analysis in Section 5.2 that draws conditional samples using dynamic evaluation shows that it allows models to generate text with statistical regularities present in the conditioning text, supporting the hypothesis the dynamic evaluation can better model surprising patterns after being exposed to them earlier in the sequence. Section 5.3 further explores what kinds of repeating patterns dynamic evaluation can exploit, and demonstrates a mechanism for how this could be occurring. Section 5.4 gives a more detailed look at optimizers in a dynamic evaluation setting, and justifies many of the design decisions made in the dynamic evaluation optimizer from Equation 4.7 in the previous chapter. Lastly, Section 5.5 benchmarks how well the models developed by advances from chapters 3 and 4 compare with humans at the task of text prediction, and finds that a dynamically evaluated mLSTM can perform text prediction on par with the best human predictors, but still worse than an ensemble of human predictors, leaving room for improvement. Some of the work in Sections 5.1, 5.2, and 5.3 was published in Krause et al. (2018).

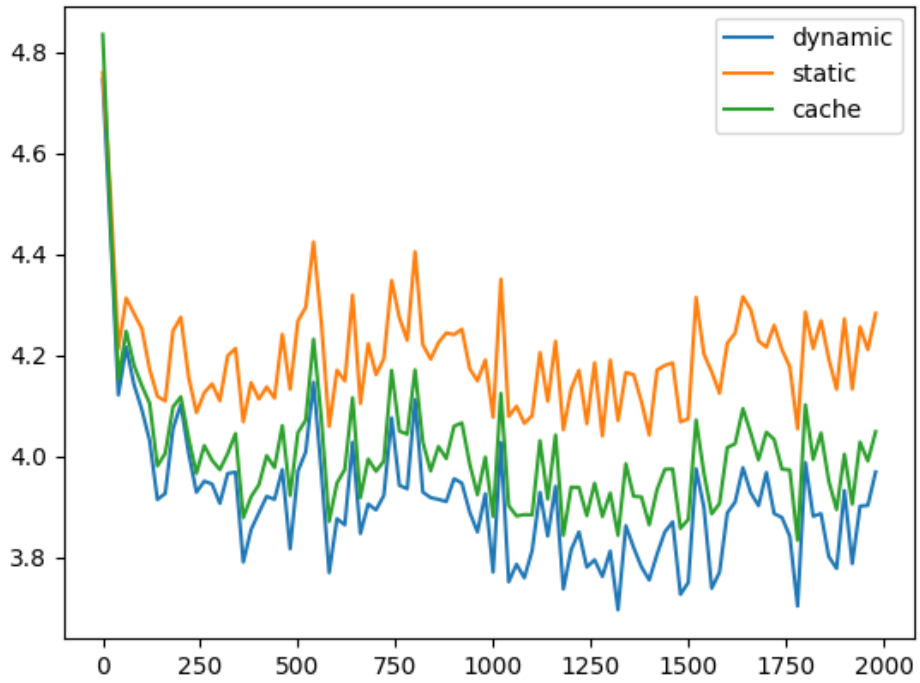
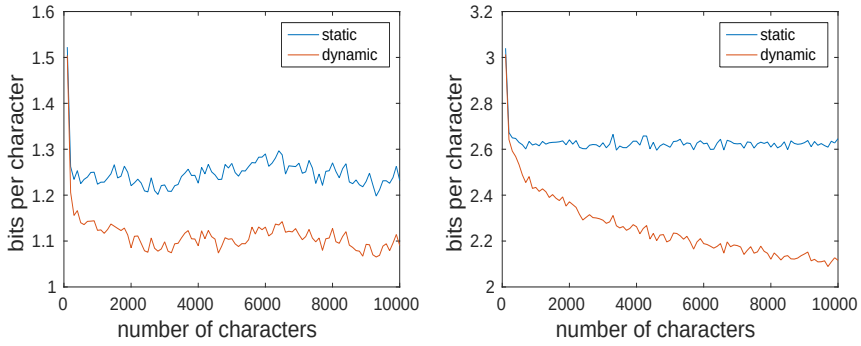


Figure 5.1: Average losses in nats/word of dynamic evaluation, the neural cache, and static evaluation plotted against number of words processed; on sequences from the WikiText-2 test set, averaged over 122 trials for each.

5.1 Time-scales of adaptive sequence modeling

We examined the timescales at which dynamic evaluation gains an advantage over static evaluation at language modeling. To observe this effect for word level-language modeling, we plotted the performance of static vs. dynamic evaluation vs. neural cache against the number of words processed on sequences from the WikiText-2 test set (using the settings from Section 4.7.1). We divided up the WikiText-2 test set up into 122 sequences of length 2000, and measured the average performance vs. number of words processed. Losses were averaged across these 122 sequences to obtain average losses at each time step. The results are given in Figure 5.1.

We also measured the time-scales at which dynamic evaluation is useful for character level language modeling. For these experiments, we also considered a domain adaptation scenario where the test sequences came from a different distribution from the training sequences. We plotted the performance of static and dynamic evaluation against the number of characters processed on sequences from the enwik8 test set, and sequences



(a) enwik8 data (b) Spanish data

Figure 5.2: Average losses in bits/char of dynamic and static evaluation plotted against number of characters processed; on sequences from the enwik8 test set (left) and European Parliament dataset in Spanish (right), averaged over 500 trials for each. Losses at each data point are averaged over sequence segments of length 100, and are not cumulative. Note the different y-axis scales in the two plots.

in Spanish from the European Parliament dataset (Koehn, 2005).

The enwik8 data experiments show the timescales at which dynamic evaluation gained the advantage observed in Table 4.4 from the previous chapter. We divided the enwik8 test set into 500 sequences of length 10000, and applied static and dynamic evaluation to these sequences using the same model and methodology used to obtain results in Table 4.4. Losses were averaged across these 500 sequences to obtain average losses at each time step. Plots of the average cross-entropy errors against the number of enwik8 characters sequenced are given in Figure 5.2a.

The Spanish experiments measure how dynamic evaluation handles large distribution shifts between training and test time, as enwik8 contains very little Spanish. This setting can be seen as similar to "finetuning", where pretrained models are adapted to new datasets. The main difference is that the model is evaluated online, where the model is evaluated on every prediction, as would be the case if a language model where to be used for compression in out-of-domain data. We used the first 5 million characters of the Spanish European Parliament data for this experiment. The Spanish experiments used the same base model and dynamic evaluation settings as the enwik8 experiments. Plots of the average cross-entropy errors against the number of Spanish characters sequenced are given in Figure 5.2b.

Dynamic evaluation gave a very noticeable advantage after a few hundred characters. For Spanish, this advantage continued to grow as more of the sequence was processed.

For enwik8, this advantage was maximized after viewing around 2–3k characters, demonstrating that the adaptation effect was local rather than global. The ability to use thousands of characters to improve its predictions on enwik8 also demonstrates that dynamic evaluation can use very long-range statistical dependencies in its predictions. This means that dynamic evaluation is able to adapt on the scale of very long sequences, since tokens from thousands of timesteps back are influencing the model's predictions enough to improve its performance.

5.2 Conditional samples with dynamic evaluation

Following up on the previous section, we drew 300 character conditional samples from the static and dynamic models after viewing 10k characters of Spanish. For the dynamic model, we continued to apply dynamic evaluation during sampling as well. The static samples quickly switched to English that resembled enwik8 data. The dynamic model generated data with some Spanish words and a number of made up words with characteristics of Spanish words for the entirety of the sample.

Below we present 300 character samples generated from the static and dynamic versions of the model trained on enwik8, conditioned on 10k of Spanish characters. The final sentence fragment of the 10k conditioning characters is given to the reader, with the generated text given in **bold**.

STATIC:
*Tiene importancia este compromiso en la medida en que la Comisión es un organismo que tiene el mon**de,***
***
There is a secret act in the world except Cape Town, seen in now flat comalo and ball market and has seen the closure of the eagle as imprints in a dallas within the country." Is a topic for an increasingly small contract saying Allan Roth acquired the government in [[1916]].***

===

DYNAMIC:
*Tiene importancia este compromiso en la medida en que la Comisión es un organismo que tiene el mon**tembre tas procedíns la conscriptione se ha Tesalo del Pómienda que et hanemos que Pe la Siemina.***

De la Pedrera Orden es Señora Presidente civil, Orden de siemin presente relevante frónmida que esculdad pludiore e formidad President de la Presidenta Antidorne Adamirmidad i ciemano de el 200'. Fo

These samples illustrate the kinds of features that dynamic evaluation was able to learn to model on the fly. For instance, the model was able to repeat real Spanish words and phrases it had never seen in training from the conditioning text. The model also made up some words with Spanish-like features. While this is an over generalization in this case, the ability to predict to unseen or out-of-vocabulary words with related features to the conditioning text could help with language prediction, especially when the language is out-of-domain.

5.3 Generalizing to unseen words

Mikolov et al. (2010) hypothesized that dynamic evaluation updates generalize not only to the direct re-occurrence of words, but also to the re-occurrence of related words. For an example of this consider the following sequence:

The Gambia won the first match 3 - 0 in Banjul , the Gambia 's capital . The return match was delayed in for 24 hours and played in Makeni. The Gambia beat Sierra Leone 4 - 3 to qualify for the final round. The Gambia then beat Tunisia 1 - 0 at home and won 2 - 1 in Tunisia .

This sequence has certain words, such as the highlighted names of African countries, that repeat themselves, and should be more predictable by their second occurrence. However, it also contains re-occurrences of related words; all the words highlighted in blue relate to competition. Observing some words that relate to competition should make words later in the sequence that relate to competition more predictable. The neural cache method can only increase the probability of words that it has previously seen in a sequence. As a result, the neural cache can never improve on unseen words. In Table 5.1, we breakdown the performance of static eval vs. dynamic eval vs. neural cache on words that have already occurred at least once in a sequence (seen words) vs. words occurring for the first time in a sequence (unseen words). We measure this

model	seen words (nats/word)	unseen words (nats/word)
static eval	**3.24**	6.92
dynamic eval	**2.78**	**6.99**
neural cache	2.91	**7.06**

Table 5.1: Negative log-likelihood loss on words that have previously occurred in the sequence (seen words) and words occurring for the first time in a sequence (unseen words) of static evaluation vs. dynamic evaluation vs. neural cache on WikiText-2.

in the second half of held out WikiText-2 sequences of length 2000, with the first 1000 tokens only used as conditioning text. As would be expected, dynamic evaluation and the neural cache are both much more effective than static evaluation at predicting words that re-occur in a sequence. Dynamic evaluation gains its advantage over the neural cache on both seen and unseen words. Static evaluation performs better than both methods on previously unseen words. We also measured the ratio of how often dynamic evaluation and the neural cache outperform static evaluation on unseen words. Dynamic evaluation achieves a lower prediction error than static evaluation on 42.6% of unseen words, as opposed to the neural cache, which as expected, always performs worse than static evaluation on unseen words. This means that unlike the neural cache, dynamic evaluation is capable of generalizing to words that have not previously occurred in the sequence.

We attempt to analyze the mechanism by which dynamic evaluation can generalize to unseen words. The change to probabilities of symbols other than the observed symbols can be measured by doing a second forward pass after each dynamic evaluation update. We generally found that dynamic evaluation can increase the probability of related words, and we demonstrate this for a specific point in the WikiText-2 test set. We analyze the output log probabilities before and after applying dynamic evaluation to the word “production”, which occurred in the following context:

“He appeared on a 2006 episode of the television series , Doctors, followed by a role in the 2007 theatre **production**”

After applying a dynamic evaluation update to the sequence segment containing the word “production”, we recomputed the output probabilities at this time step with the updated network weights. We measure the change in log probabilities to words with a similar word embedding to “production”. The results of this experiment is given in

Word	emb. cos distance	Δ log prob
“production”	1.00	+2.42
“development”	0.55	+1.14
“construction”	0.53	+1.29
“filming”	0.52	+0.80
“recording”	0.50	+1.27
“photography”	0.46	+0.15
“release”	0.45	+1.25
“performance”	0.45	+1.01
“design”	0.44	+0.90
“work”	0.44	+1.23
“productions”	0.44	+0.50
median	−0.02	−0.66

Table 5.2: Effect of dynamic evaluation on probabilities of related words. We measure how updating on the target word “production” changes the probabilities of words most related to production, measured by cosine distance of word embeddings learned during training time (note that input and output embeddings are tied). The median values across the entire vocabulary are given in the bottom entry.

Table 5.2.

Updating on the observation of the word “production” also increases the log probability of words with similar word embeddings. This experiment is meant to simulate the situation where the model sees the exact same context twice. If the same context were to occur again in sequence, the model would likely assign a higher probability to the word “production” as well the related words in Table 5.2. It is more likely that the model would need to predict these words in similar but slightly different context, but this generalization to related words could still apply in a slightly different context.

To analyze this effect, we consider the more general case where a model with parameters θ conditions on a sequence $x_{1:t}$ to predict a distribution over the sequence token at position $t+1$, $p(x_{t+1}|x_{1:t},\theta)$. After predicting, the model performs a gradient descent update with learning rate η using

$$\Delta\theta = \eta\nabla_\theta \log p(x_{t+1}|x_{1:t},\theta) \tag{5.1}$$

(to avoid double negatives in some of our equations, we write gradients as being with

respect to positive log likelihoods and assume we are updating in the direction of the positive gradient), resulting in adapted parameters $\theta + \Delta\theta$. This corresponds to a dynamic evaluation update with a simple SGD optimizer. Then, the updated model predicts $p(x_{t+1}|x_{1:t}, \theta + \Delta\theta)$ again to see how the distribution over x_{t+1} changed as a result of the update. We would obviously expect the token observed at $x+1$ to have a higher probability after the update, but if our model is generalizing well, we might also expect words related to the observed word to have a higher probability after the update. We consider the scenario where there are two words i and j, which we will assume are semantically similar words with similar word embeddings (if we are using tied embeddings, we do not need to worry about if they are input or output embeddings). We are interested in the effect that updating on word i has on the probability predicted for word j after the update in the above scenario. For an infinitesimally small learning rate, the change in log probability of word j (given as a log ratio of probabilities for compactness) as a function of an update $\Delta\theta$ is given by a first order Taylor series approximation

$$\log\left(\frac{p(x_{t+1}=j|x_{1:t},\theta+\Delta\theta)}{p(x_{t+1}=j|x_{1:t},\theta)}\right) \approx (\nabla_\theta \log p(x_{t+1}=j|x_{1:t},\theta))^\top \Delta\theta. \tag{5.2}$$

Under the condition that

$$(\nabla_\theta \log p(x_{t+1}=j|x_{1:t},\theta))^\top (\nabla_\theta \log p(x_{t+1}=i|x_{1:t},\theta)) > 0, \tag{5.3}$$

an update in the direction of the gradient that results from observing word i will increase the probability of word j under the approximation. We make the argument that word embeddings contribute significantly to the gradient that results from observing a word, so words with similar word embeddings will often have respective gradients with positive inner products.

Given embedding matrix $W_{yh} \in \mathbb{R}^{d_h, d_y}$, where d_y is the vocab size, and d_h is the hidden state size, we denote the word embedding vector for word i as W_{yh}^i, and the word embedding vector for word j as W_{yh}^j. We are assuming that the inner product $(W_{yh}^i)^\top (W_{yh}^j) > 0$. The gradients of the pre-softmax output activation $\hat{y}_t$ are given by the difference between a one hot encoding of the target token i, e_i, and the post-softmax output probabilities y_t.

$$\frac{\partial \log p(x_{t+1}=i|x_{1:t},\theta)}{\partial \hat{y}_t} = e_i - y_t \tag{5.4}$$

For language modeling d_y is normally large (it is 33k for WikiText-2 for instance). Unless the model confidently predicts the correct answer, a large portion of the mass in

$\frac{\partial \log p(x_{t+1}=i|x_{1:t},\theta)}{\partial \hat{y}_t}$ will be in the index corresponding to the observed target token i. This portion of the gradient is equal to the scalar $1-y_t^i$, where y_t^i is the output probability predicted for word i. y_t^i will usually be closer to 0 than to 1 for most predictions from most word-level language models. If we approximate the hidden state gradient $\frac{\partial \log p(x_{t+1}=i|x_{1:t},\theta)}{\partial h_t}$ by ignoring contributions from output units other than i , we would get

$$\frac{\partial \log p(x_{t+1}=i|x_{1:t},\theta)}{\partial h_t} \approx \frac{\partial \log p(x_{t+1}=i|x_{1:t},\theta)}{\partial \hat{y}_t^i}\frac{\partial \hat{y}_t^i}{\partial h_t}, \tag{5.5}$$

which simplifies to

$$\frac{\partial \log p(x_{t+1}=i|x_{1:t},\theta)}{\partial h_t} \approx (W_{yh}^i)(1-y_t^i). \tag{5.6}$$

If we were to use the same approximation for the hidden state gradient with respect to $\log p(x_{t+1}=j|x_{1:t},\theta)$, we would get

$$\frac{\partial \log p(x_{t+1}=j|x_{1:t},\theta)}{\partial h_t} \approx (W_{yh}^j)(1-y_t^j). \tag{5.7}$$

Based on the above approximations, the hidden state gradients that result from observing i and j will have an inner product (Equation 5.3) given by

$$(1-y_t^i)(1-y_t^j)(W_{yh}^j)^\top(W_{yh}^i). \tag{5.8}$$

Since the $(1-y_t^i)$ and $(1-y_t^j)$ terms are always positive and generally close to 1, the second term, which is the inner product of the word embeddings, is proportional to the inner product of the hidden state gradients under our approximation. While this approximation accounts for a significant portion of the hidden state gradient for most language model predictions, it does ignore the derivatives with respect to output units other than the target output unit. Furthermore, even when the gradients of the hidden states have positive inner products, the gradients of the weights will not always have positive inner products. Even if the gradients of the weights have positive inner products, for a non-infinitesimal step size, it is still possible for a gradient descent update on word i to decrease the probability of observing word j. However, this analysis does demonstrate a mechanism by which dynamic evaluation updates could and might be expected to generalize to words with similar word embeddings.

We examined this effect at the level of individual token updates, but in general dynamic evaluation is likely to be effective at generalizing to sequences with similar gradients. We would expect applying dynamic evaluation with a small enough learning rate applied to sequence $x_{1:t}$ to help with predicting sequence $y_{1:t}$ when

$$(\nabla_\theta \log p(x_{1:t}|\theta))^\top(\nabla_\theta \log p(y_{1:t}|\theta)) > 0. \tag{5.9}$$

We observed that test sequences on WikiText-2 that occur in close proximity do tend to have gradients under the static model with higher inner products with each other than random test sequences in general. It seems from our analysis in this section at the token level that during training the model learns a representation where stylistically similar sequences result in similar gradients. While this is easier to analyze at the level of individual tokens, it could potentially apply more broadly to stylistic regularities. The exploration of gradients under a language model as a similarity metric between sequences could be interesting future work.

5.4 Optimizers for dynamic evaluation

The previous chapter on dynamic evaluation presented an optimizer that worked particularly well in an online setting. This optimizer had a number of unique features, including the use of RMS gradients from the training data (or source task, as opposed to the target task as in normal RMSprop (Tieleman and Hinton, 2012, Section 2.5.3)) for inverse learning rates, and the use of a decay prior towards the initially learned parameters. While this optimizer gave strong results relative to past approaches, only a small space of potential optimizers was considered. This section explores a number of commonly used deep learning optimizers as additional baselines, and tests design decisions such as momentum and adaptive learning rates in a dynamic evaluation setting. We find that features of optimizers that work well for training do not always work well in a dynamic evaluation setting, further justifying the optimizer presented in the previous chapter.

5.4.1 Incorporating momentum

Momentum (Polyak, 1964; Nesterov, 1983; Sutskever et al., 2013, Section 2.5.2) is a modification to gradient descent that is commonly used to help with convergence in deep learning. Momentum delays the effect of gradient based updates on the weights of the network, potentially giving the optimizer a chance to correct for mistakes before stepping too far. While this is often helpful in the traditional optimization setting where there is a separate training and testing phase, it could be undesirable in an online setting where the network is penalized for every mistake.

This subsection explores the introduction of a momentum parameter in a dynamic evaluation optimizer with SGD. The dynamic evaluation update rule is then given by

momentum	PTB validation perplexity
Static	59.7
0.0	**54.7**
0.5	54.8
0.75	55.0
0.875	55.3
0.9375	55.7
0.96875	56.2
0.984375	56.7

Table 5.3: Effect of momentum in dynamic evaluation on PTB perplexity

this series of equations using the notation from the previous chapter:

$$v^i \leftarrow \mu v^{i-1} - \eta \nabla \mathcal{L}(s_i) \tag{5.10}$$

$$\theta_l^i \leftarrow \theta_l^{i-1} + v^i. \tag{5.11}$$

The μ parameter sets the momentum, with $\mu = 0$ equivalent to plain SGD. We ran experiments to measure the effect of momentum in dynamic evaluation on PTB validation loss, using the same setup and base model as in Section 4.7.1 in the previous chapter. We varied μ values from the series $1 - \frac{1}{2^n}$, so starting from $\mu = 0$ and increasing towards one. For each μ value, we found the optimal learning rate by starting with a sufficiently low value for η, and repeatedly increasing η by 20% until validation performance peaked. It is important to tune the learning rate separately for each μ value, because larger μ values result in more aggressive momentum and thus require smaller learning rates. The results are given in Table 5.3. Increasing momentum always hurt dynamic evaluation performance, with $\mu = 0$ (vanilla SGD) giving the best validation error.

5.4.2 Adaptive learning rates

One might consider applying optimizers that work well for training neural networks from scratch in a dynamic evaluation settings. In this section, we consider applying adaptive learning rate optimizers RMSprop (Tieleman and Hinton, 2012, Section 2.5.3)

model	PTB validation perplexity
Static	59.7
RMSprop ($\alpha = 0.9$)	59.1
RMSprop ($\alpha = 0.99$ (default in PyTorch))	58.6
RMSprop ($\alpha = 0.999$)	**58.1**
Adam ($\alpha = 0.999$, $\mu = 0.9$ (default in PyTorch))	58.1
Adam ($\alpha = 0.999$, $\mu = 0.0$)	**58.0**
SGD	**54.7**

Table 5.4: Dynamic evaluation PTB perplexity using RMSprop and Adam optimizers

and Adam (Kingma and Ba, 2014, Section 2.5.4). The global RMS (Equation 4.7) optimizer presented in the previous chapter is related to RMSprop, but uses RMS gradients on the training data rather than recent test data. This allows the global RMS optimizer to use larger batch sizes to compute gradient statistics, and also results in fixed learning rates for each weight rather than adaptive learning rates. This subsection explores the differences in performance of using RMSprop vs. RMS training gradients. It also re-examines the effect of momentum by applying an ADAM optimizer directly in a dynamic evaluations setting, following up on the previous subsection which showed that momentum hurt performance in a vanilla SGD dynamic evaluation. Adam, RMSprop, and SGD with momentum traditionally use different variable names for very similar hyperparameters, so to make things consistent, we use μ as momentum (used in SGD with momentum), and Adam, and α as the weighted averaging parameter for RMS gradients (as in Section 2.5).

We initially considered RMSprop and ADAM style dynamic evaluation using the default settings for PyTorch (Paszke et al., 2017), only tuning the learning rate. We the considered varying the α parameter for RMSprop and ADAM, and ran ADAM with and without momentum. Learning rates for all experiment were found by starting with a sufficiently low value, and repeatedly increasing the learning rate by 20% until validation performance peaked. The results are given in Table 5.4, with SGD from the previous experiment included as a baseline.

These experiments showed that momentum still hurt performance with adaptive learning rates, and that the bias correction of Adam helped slightly (Adam without momentum could be seen as bias corrected RMSprop). The most important overall finding

from these experiments is that RMSprop and Adam perform much worse than even vanilla SGD in a dynamic evaluation setting. This could be seen as surprising, because the optimizer originally presented for dynamic evaluation that worked noticeably better than SGD had an update rule similar to RMSprop.

5.4.3 Hybridizing RMSprop and SGD

Following from the previous subsection, we explore why RMSprop as implemented in deep learning frameworks performs so poorly on dynamic evaluation. We found that this could be largely attributed to the ϵ parameter in RMSprop that controls numerical stability. The update rule for RMSprop is given by

$$\theta_l^i \leftarrow \theta_l^{i-1} - \eta \frac{\nabla \mathcal{L}(s_i)}{\sqrt{MS} + \epsilon} \tag{5.12}$$

Where MS is the running average of the mean squared gradients. The ϵ parameter is generally set to be very small and is mainly for numerical stability purposes (10^{-8} is the default setting in many frameworks, and was used for Adam and RMSprop in the previous experiments). In word-level language modeling, the running average for MS can be near zero for weights associated with rare words. In this case, ϵ can have a large effect on the update magnitude the first time a new word occurs. As a result, dynamic evaluation in word level language modeling is very sensitive to the ϵ parameter. Another view of the ϵ parameter is to determine how SGD-like the optimizer is. As $\epsilon \to \infty$, the optimizer becomes equivalent to SGD under a transformation of the learning rate (this transformation would involve scaling the learning rate by $1/\epsilon$ so that the update does go to zero). For very small ϵ values, the optimizer can behave very differently from SGD, and acts more like RMSprop in its intended form, with ϵ only used for numerical stability. We measured dynamic evaluation performance of an RMSprop optimizer on PTB with varying ϵ values (with the learning rate tuned to each ϵ value). For comparison, we also include the dynamic evaluation optimizer used in the previous chapter that uses RMS gradient statistics from the training data (equation 4.7), which we refer to as "global RMS", but leave out the decay prior for a fairer comparison with RMSprop. The results are in Table 5.5.

The performance of RMSprop peaked at slightly better than SGD for an ϵ value of 0.01, suggesting that there was a slight benefit to using adaptive learning rates if carefully hybridized with SGD. As expected, RMSprop behaved identically to SGD as the value of ϵ become very large. While these experiments show that if carefully tuned,

model	PTB validation perplexity
Static	59.7
RMSprop $\varepsilon = 0.000001$	57.5
RMSprop $\varepsilon = 0.00001$	56.3
RMSprop $\varepsilon = 0.0001$	55.2
RMSprop $\varepsilon = 0.001$	54.6
RMSprop $\varepsilon = 0.01$	54.4
RMSprop $\varepsilon = 0.1$	54.6
RMSprop $\varepsilon = 1.0$	54.7
RMSprop $\varepsilon = 10$	54.7
RMSprop best, $\varepsilon = 0.005$	**54.4**
SGD	54.7
Global RMS	**52.7**

Table 5.5: Dynamic evaluation PTB perplexity for varying ε values for RMSprop

RMSprop can gain a very slight advantage over SGD, the gains relative to SGD are much smaller than achieved by the global RMS optimizer.

5.4.4 Global RMS vs. RMSprop

We perform follow up experiments to investigate why standard RMSprop performs worse than using a Global RMS optimizer.

5.4.4.1 Amount of data for gradient statics

One hypothesis is that since the global RMS optimizer is able to use the entire training set to compute gradient statistics, the gradient statistics should be more accurate, and this could help the optimizer take better step sizes. A standard RMSprop optimizer can only use the sequence history to compute gradient statistics, so it will have leverage less data to compute RMS gradients than a global RMS optimizer. To test the theory that this could be having some effect on performance, we ran a global RMS optimizer using smaller portions of the training data (10% and 1%) to collect gradient statistics, and measured performance vs. the global RMS optimizer that uses all of the training data to collect gradient statistics in Table 5.6.

model	PTB validation perplexity
Static	59.7
RMSprop (with $\varepsilon = 0.01$)	54.4
Global RMS (1% for gradient statistics)	**52.5**
Global RMS (10% for gradient statistics)	52.5
Global RMS (100% for gradient statistics)	**52.7**

Table 5.6: Dynamic evaluation PTB perplexity using RMSprop and Adam optimizers

Using smaller portions of the training data to collect gradient statistics had a negligible effect on performance, which contradicts the hypothesis that the amount of data used for gradient statistics could explain the performance difference between RMSprop and global RMS.

5.4.4.2 Local vs. global gradient statistics

Another hypothesis for why global RMS may work better than RMSprop is that there is an advantage to collecting gradient statistics from sequences from the global distribution of text rather than the local, sequence specific, distribution of text. We have observed that gradients of static neural language models are locally correlated, meaning that the gradients of sub-sequences in close proximity to each other tend to have higher inner products with each other than random sub-sequences in general. So we would expect updating on the gradients of these sub-sequences earlier in a long sequence to reduce the loss on predictions later in the sequence, which is likely why dynamic evaluation works (see Section 5.3). Collecting gradient statistics from the recent history would have the effect of dampening update directions that have been occurring locally, even if these update directions are rare globally. So for instance, if a rare word occurs multiple times in a sequence, the learning rates for the weights associated with this rare word would get smaller after each occurrence with RMSprop, but would stay constant (and likely large, since the word is rare) for global RMS. Since being able to model rare words well is important for dynamic evaluation, it may be undesirable to dampen the updates for these rare words, as RMSprop would do.

To test this hypothesis, we ran a version of RMSprop that we refer to as "non-local RMSprop", that collected gradient statistics from a source independent of the evaluation sequence. This procedure involved running two versions of the model in parallel, one

model	PTB validation perplexity
Static	59.7
RMSprop (with $\varepsilon = 0.01$)	54.4
Global RMS	**52.7**
Non-local RMSprop	**52.6**

Table 5.7: Dynamic evaluation PTB perplexity using RMSprop and ADAM optimizers

which was used for evaluation purposes, and one which was used to collect gradient statistics for RMSprop. The setup was exactly like dynamic evaluation with RMSprop, except that the sequence used for the recency weighted RMS gradients was different from the evaluation sequence. The results of non-local RMSprop along with the relevant baselines are given in Table 5.7. Non-local RMSprop achieved results on par with the Global RMS optimizer from the previous chapter. This result supports the hypothesis that using local gradient statistics for RMSprop has an undesirable effect for dynamic evaluation.

5.5 Comparing state of the art character models with human predictors

Measuring the entropy of the English language has long been of interest to information theorists (Shannon, 1951; Cover and King, 1978; Brown et al., 1992b). The true entropy of text is given by the generating distribution of the human writing the text. The entropy of a particular sample of English text can be upper bounded by the entropy of that text under a particular model, where model could be a human or a probabilistic algorithm. In the past, it was generally assumed that humans could evaluate the entropy of text better than other probabilistic models existing at the time; language is generated by humans after all. However, measuring the entropy of text under a human's language model is non-trivial, as humans do not just output probability distributions. A couple of well known past works have used text predictions from human subjects to yield estimates for the entropy of English that are widely cited today (Shannon, 1951; Cover and King, 1978).

The (binary) entropy of a discrete random variable is given by

$$H(X) = -\sum_i p(x_i) \log_2 p(x_i). \tag{5.13}$$

The entropy of English text generally refers to the conditional entropy of a character of text, measured in bits/character, given the history of text. The entropy of a distribution p can be upper bounded by using distribution q as an approximation to distribution p. The cross entropy of p approximated by q is given by

$$H(X) = -\sum_i p(x_i) \log_2 q(x_i) \tag{5.14}$$

While exactly measuring the entropy of English text is impossible without the generating process of the human writing the text, a tighter upper bound can be established with a better approximating distribution. In this section, we compare how well approaches developed in this thesis can upper bound the entropy of English text versus human subjects in past work.

Shannon (1951) attempted to measure the entropy of the English language using what is now known as the "Shannon game". Human subjects repeatedly guessed the next letter of text (limited to be 26 characters plus spaces) until they guessed correctly, and then moved onto the next letter. Shannon (1951) used the number of guesses for each symbol to estimate bounds to the human-level entropy of text prediction. If the model (or human) guesses the next symbol in descending order of probability, the entropy of the data under the model can be upper bounded using the entropy of the distribution over the number of guesses it takes the model to predict the next symbol. q_i^N is the frequency that the model playing the Shannon game guesses the symbol correctly on the ith guess. The entropy of this distribution, which is the upper bound for the entropy of the generating distribution of the data, is given by:

$$H(X) \leq -\sum_{i=1}^{27} q_i^N \log_2 q_i^N \tag{5.15}$$

This upper bound is achievable because the model could always assign probability q_i^N to its ith guess and it would achieve this entropy. However, doing this would not allow the model to update the relative confidence of its guesses depending on its context, which makes this a relatively loose upper bound. Shannon (1951) measured the entropy on several segments of text and estimated the upper bound of human prediction to be 1.3 bits/character.

The main drawback of the Shannon game is that models have no way to specify how confident they are in their predictions. Cover and King (1978) devised a more precise way to measure the human-level entropy of text, where subjects played a modified version of the Shannon game where they also waged bets on how confident they were in their guesses. Cover and King (1978) showed that an ideal gambler's wagers are proportional to the probability of the next symbol on their next bet, and thus established a direct relationship between the return of the gambler and the entropy of the data under the gambler's model. Cover and King (1978) also provided the exact excerpt of text they used, making direct comparison with future work possible. Cover and King (1978) used 12 human subjects to measure the entropy from an excerpt taken from the book Jefferson the Virginian (Malone, 1948) (which was the same book but a different excerpt from what was used by Shannon (1951)). The subjects predicted the 75 characters of bold text (given below) excluding the comma and case insensitive, and were given access to the entire book up to this excerpt to allow them to adjust to the author's style. The human predictors, made up of students and professors at Stanford, were apparently very dedicated to achieving the best performance possible, spending an average of 5 hours each to make predictions on just 75 characters of text.

The text excerpt is given here, with the evaluation text given in **bold**:

*She was not only a "pretty lady" but an accomplished one in the customary ways, and her love for music was a special bond with him. She played on the harpsichord and the pianoforte, as he did on the violin and the cello. The tradition is that music provided the accompaniment for his successful suit: his rivals are said to have departed in admitted defeat after hearing him play and sing with her . In later years he had the cheerful habit of singing and humming to himself as he went about his plantation. This is not proof in its****elf that he was a pleasing vocal performer, but with Martha in the parlor it***

We apply the advances in character level language modeling developed in the earlier chapters of this thesis to compare with the human prediction results from Cover and King (1978). We use the mLSTM model on text8 from Section 3.5.3 with dynamic evaluation applied using the same settings as in Table 4.5.

The distribution of the training set and test set are noticeably different, as text8 is taken from modern Wikipedia, whereas the excerpt is from a book written in 1948. To partially make up for this, we give the entire book up until the test sequence as conditioning text to allow the model to adapt to the author's style. Perhaps better results

model	evaluation error (bits/char)
Human (avg subject)	1.59
Human (based on avg gambling return)	1.34
Human (best subject out of 12)	**1.29**
Human (ensemble of 12)	1.25
text8 mLSTM	1.44
text8 mLSTM + dynamic eval	**1.31**

Table 5.8: Performance of mLSTM vs humans on predicting text from Jefferson the Virginian

could be achieved with a full retraining on a dataset of text from this author or time-period. The results, in comparison to the human level results given by Cover and King (1978), are presented in Table 5.8.

The mLSTM trained on text8 with dynamic evaluation applied performs on par with the best human text predictors. It is likely that deep learning models and human models make different kinds of errors. For instance, humans are much better at understanding the higher level structure and logic of the text. Deep learning models may be able to more accurately know their own confidence, and may capture certain statistical regularities that humans fail to capture. As these two types of models likely make different kinds of errors, the true entropy of the given sample of English text is probably lower than either model alone could achieve.

Other evaluation methods in this thesis compare one model with another, which is useful for comparing architectural features. These human experiments help put some of the model comparison experiments into context, and give a sense of how well these algorithms are actually doing at language prediction, since human performance is a strong baseline. From these results we can say that RNN based language models can achieve text prediction on par with the best human predictors, and that mLSTM and dynamic evaluation help close this gap. However, given that an ensemble of humans can still perform better, and that humans are likely still non-optimal, it seems likely that there is still plenty of room for improvement in text prediction.

5.6 Conclusion

This chapter aimed to better understand why dynamic evaluation improves the prediction ability of strong baseline language models. We demonstrated that dynamic evaluation was able to use about two thousand characters of context to improve its predictions on enwik8, meaning that it is able to exploit long range statistical dependencies between tokens on this task–an ability critical for achieving adaptation to long sequences. We demonstrate how this adaptive ability helps dynamic evaluation recover from surprising sequences by modeling re-occurring sequential patterns. We also explore optimization for dynamic evaluation, and show how using gradient statistics for RMSprop from the training data (instead of the adaptation data) can be beneficial in an adaptive setting. Future work could apply this to other adaptation settings as well, such as normal finetuning.

Chapter 6

Dynamic evaluation of Transformer language models

In the time frame of this thesis, major advances were made to the field of language modeling. The largest advance, was the introduction of transformer language models, which have replaced LSTM language models for most–but not all text applications at the time of writing of this thesis. As compared with LSTMs, Transformers are able to use much longer range statistical dependencies, potentially making them more adaptive at the sequence level. In this chapter, we explore the extension of dynamic evaluation to Transformers. Gains resulting from dynamic evaluation are smaller than gains observed in LSTMs in the previous chapters, suggesting that Transformers may have more inherent adaptation ability, and this may partially explain their advantage. While the improvements were smaller than in LSTMs, dynamic evaluation does still lead to significantly better language modeling results both in and out-of-domain, indicating that the improved adaptation ability from using dynamic evaluation is still useful for Transformers. These results suggest that, despite the Transfomer's ability to use long range statistical dependencies, they are not fully able to adapt to recently seen text. This further supports our thesis motivation of developing more adaptive methods for language modeling, showing that it is applicable to different models. Some of the work in this chapter is published in Krause et al. (2019).

6.1 Introduction

The previous two chapters focused on applying dynamic evaluation to LSTM and similar recurrent neural network architectures, which were the state of the art for language

modeling at the time that work was done. LSTMs conventionally used for language modeling have been shown to use relatively short contexts to make predictions. For instance, Khandelwal et al. (2018) showed that LSTMs trained on WikiText-2 and Penn Treebank can use up to about 200 tokens of context to make predictions, and that the token order only matters for the most recent 20 or so tokens. Dynamic evaluation helps LSTM based language models better exploit long range dependencies, as demonstrated in Section 5.1. A more recently proposed neural architecture, the Transformer (Vaswani et al., 2017, Section 2.4.5.2), has been shown to be able to use longer range dependencies in its predictions compared with LSTMs (Dai et al., 2019). Dynamic evaluation can be applied to any language model at test time, but to our knowledge, no previous work has applied dynamic evaluation to Transformers.

Transformers use a combination of a self-attention mechanism and positional embeddings to encode information about the sequence history (Vaswani et al., 2017, Section 2.4.5.2). The use of self-attention provides shorter paths for information to travel, which is conjectured to be one of the main reasons that Transformers achieve better results on common language modeling benchmarks, when compared to other models (Dai et al., 2019). Moreover, Transformers trained on very large datasets can generalize to other NLP tasks (Devlin et al., 2018; Radford et al., 2018, 2019), and generate samples over long time frames that are sometimes realistic enough to trick humans into thinking they are human generated (Radford et al., 2019; Zellers et al., 2019).

Dynamic evaluation adapts models to the recent sequence history via gradient descent in order to exploit re-occurring sequential patterns. Natural language tends to have long range dependencies associated with the style and word usage of particular passages of texts; and dynamic evaluation can exploit these dependencies via online model adaptation. Transformers with a large memory cache also potentially have the capability of adapting to the style of the recent sequence history. The self-attention mechanism could potentially learn to represent a similar algorithm to non-parametric adaptation methods such as the neural cache (Grave et al., 2017b) or pointer sentinel RNN (Merity et al., 2017). While Transformers in theory have this adaptive capability, although it is unclear to what extent they learn to do this in practice. An untrained Transformer has no inductive bias towards predicting repeating patterns in sequences, and would have to learn this capability from training data. Applying dynamic evaluation ensures that this ability will be present whether the model has learned to do this from training or not. Transformers also require memory that scales linearly with the context length used, whereas dynamic evaluation does not have this computational requirement.

Dynamic evaluation and Transformers have each shown their respective capabilities to over a thousand of timesteps of context to improve predictions (as demonstrated for dynamic evaluation in Section 5.1, where we showed that using up to two thousand characters of conditioning text continued to reduce prediction errors on Wikipedia data, and for Transformers by Dai et al. (2019)), but it is unclear how much overlap there is between the type of long-range dependencies exploited by Transformers and dynamic evaluation. If Transformers are able to fully adapt to the style of the recent sequence history, there should be little to no advantage of using dynamic evaluation. Therefore, in this work, we explore the utility of applying dynamic evaluation to Transformers.

6.2 In-domain language modeling

These experiments apply dynamic evaluation to Transformers on standard language modeling benchmarks, where the training and test set come from the same domain. A number of variants of Transformers have been suggested for language modeling (Al-Rfou et al., 2018; Liu et al., 2018; Baevski and Auli, 2019; Radford et al., 2018), but in this section, we focus on the Transformer-XL architecture of Dai et al. (2019), which has recently improved state-of-the-art results on a number of common language modeling benchmarks.

6.2.1 Transformer-XL

The Transformer-XL is a modified version of the vanilla Transformer presented in Section 2.4.5.2. The main differences are the use of relative positional encodings and a segment-level attention recurrence.

When evaluating a vanilla Transformer to language modeling over long sequences, the model uses some fixed context window based on its attention length. When evaluating the perplexity of sequences longer than the model's attention length, the sequence can be broken up and evaluated in segments. Using the series of equations presented in Section 2.4.5.2, Transformer decoders map an input sequence of length N $x_{1:N}$ to an output sequence $y_{1:N}$, where $y_{1:N}$ is used to make predictions about $x_{2:N+1}$ (as reviewed in Section 2.4.5.2, masked attention to prevent the model from having access to $x_{t+1:N}$ when predicting y_t). In the case where we evaluate a longer sequence using non-overlapping segments of length N (meaning we are using a context length of N and evaluation length of also N), then when predicting the first segment element y_1, the

model only has 1 token of context, resulting in a poor prediction. The other extreme would be to (after evaluating the first sequence segment) only predict the last segment element y_N conditioned on $x_{1:N}$, discarding the predictions $y_{1:N-1}$. This method can only predict 1 token at a time, making it very slow to evaluate the whole sequence.

Transformer-XL works around the problem of fixed length attention by using a segment level attention recurrence. Hidden states after processing a sequence segment are cached into a memory of length M, where typically $M \geq N$, and are maintained when processing subsequent sequence segments. This way, even when making predictions at the beginning of a sequence segment, the model will have at least M tokens of context to make predictions.

Applying the idea of segment level attention recurrence naively to a vanilla Transformer would lead to problems distinguishing positional information between hidden states in the current sequence segment and hidden states in the previously cached hidden sequence segment. For instance, when applying attention over the current sequence segment and the previously cached sequence segment, the positional encoding (given by Section 2.4.5.2, Equations 2.35 and 2.36) at position n in the current sequence segment and the previously cached sequence segment would be the same. As a result, the model would not be able to distinguish between the position of these previous states, resulting in a performance loss. To address this, Transformer-XL uses relative positional encodings in place of the absolute positional encodings used in the vanilla Transformer. In vanilla Transformers, when computing attention between position t and earlier position n, the tth and nth rows of the positional embedding matrix U are used to compute positional encodings used for the computation (Section 2.4.5.2, Equations 2.37, 2.38, and 2.39). In relative positional encodings used in Transformer-XL, the $(t-n)$th row of U is used to compute positional encodings that contribute to the attention score between t and n. Dai et al. (2019) shows that the unnormalized attention scores $\hat{A}$ in Transformers between the tth and nth position (for a given attention head) can be decomposed as

$$\hat{A}_{t,n} = X_t^\top W_{qx}^\top W_{kx} X_n + X_t^\top W_{qx}^\top W_{kx} U_n + U_t^\top W_{qx}^\top W_{kx} X_n + U_t^\top W_{qx}^\top W_{kx} U_n \tag{6.1}$$

where X_t and X_n are vectors correspond to the tth and nth rows of the matrix of embedding across the sequence X (this notation is slightly different from in Section 2.4.5.2, which used subscript to denote layer), U_t and U_n are vectors correspond to the tth and nth rows of positional encoding matrix U, and (as in Section 2.4.5.2), W_{qx} and W_{kx} are the learnable weight matrices for the queries and keys of the self attention. This equation can be derived from the equations in Section 2.4.5.2. Transformer-XL replaces

the absolute positional encodings U_n with relative positional encodings U_{t-n}. It also introduces two learnable parameter vectors $u \in \mathbb{R}^{d_x}$ and $v \in \mathbb{R}^{d_x}$, which replace the two instances of U_t. This results in a new equation for unnormalized attention scores.

$$\hat{A}_{t,n} = X_t^\top W_{qx}^\top W_{kx} X_n + X_t^\top W_{qx}^\top W_{kx} U_{t-n} + u^\top W_{qx}^\top W_{kx} X_n + v^\top W_{qx}^\top W_{kx} U_{t-n} \tag{6.2}$$

This relative encoding scheme gives the model the prior that only relative positional information should matter, which is desirable for language modeling. Furthermore, it allows the segment level attention recurrence to distinguish between positions in different cached sequence segments and the sequence segment it is currently processing. The relative encodings in Transformer-XL have also been shown to be able to generalize to longer sequence lengths than seen during training (Dai et al., 2019), allowing for more efficient training with shorter sequence segments to be an effective strategy.

6.2.2 Experimental set-up

We apply dynamic evaluation to pretrained Transformer-XL models from Dai et al. (2019) on two character-level datasets and one word-level dataset. We chose these 3 datasets because they all contain long-range dependencies that span across sentences and paragraphs, since they feature articles in unshuffled order. Details of the model training can be found in Dai et al. (2019), and we downloaded models using their code[1].

Following the work in Chapter 4, which applies dynamic evaluation to RNNs at the sequence segment level, we apply dynamic evaluation to Transformer-XL models at the sequence segment level. As noted in Section 6.2.1, Transformer-XL uses a segment level attention recurrence, proccesing sequences in segments of length N and storing the resulting embeddings in memory cache of length M. We align the sequence segments of length N used for Transformer-XL with the sequence segments used to compute the gradient for dynamic evaluation. The gradient is computed once for each sequence segment (after taking a loss on the segment), and backpropagation is truncated to be contained within a single sequence segment.

We measured the performance of two types of dynamic evaluation; one which used the best optimizer from Chapter 4, which we refer to as "global RMS dynamic eval + RMS decay" (update rule given in Equation 4.7), and one that used stochastic gradient descent, which we refer to as "SGD dynamic eval". Following the approach

[1] `https://github.com/kimiyoung/transformer-xl`

from Chapter 4, we tuned hyperparameters for dynamic evaluation on the validation sets before evaluating on the test sets.

6.2.3 Character-level experiments

As with RNNs in Chapter 4, we use the enwik8 (Hutter Prize) and text8 datasets to benchmark dynamically evaluated Transformer-XL on character level language modeling. In addition to applying dynamic evaluation to the largest pretrained models by Dai et al. (2019), we also re-train a smaller Transformer-XL on enwik8 with $42M$ parameters using code and hyperparameter settings from Dai et al. (2019). This model size is more comparable to results elsewhere on enwik8 for this thesis. We noticed a slight anomaly in the preprocessing of enwik8 in the code released by Dai et al. (2019) that ignored a very rarely occurring token, causing it to have 204 unique tokens (rather than the standard 205 tokens used in most results, for instance in Graves (2013)), and caused the datasets to be shorter by a nearly negligible amount. Our results also contain this anomaly since we use their implementation. Following Dai et al. (2019), we used sequence segments of 128 and a memory cache of length 3800 for evaluation for the large models on both datasets. For the smaller model on enwik8, we used sequence segments of 128 and a memory cache of length 3800 for evaluation (also following Dai et al. (2019)). Results for enwik8 and text8 are reported in Table 6.1 and Table 6.2 respectively. Applying Dynamic evaluation improves the Transformer-XL by a noticeable margin, achieving state of the art on both of these character-level datasets.

6.2.4 Word-level experiments

We evaluate dynamic evaluation on word-level Transformer-XL using the WikiText-103 dataset (Merity et al., 2017), which is derived from the same Wikipedia data source as WikiText-2 (first used in Section 4.7.1), but has a larger training set and vocabulary. WikiText-103 contains 103 million training tokens, and a vocabulary size of 268k. Given the large vocabulary size, the pretrained model we re-evaluate from Dai et al. (2019) used an adaptive softmax output layer (Grave et al., 2017a) to make training faster. Results for WikiText-103 are reported in Table 6.3. There was no noticeable validation advantage to using a decay rate, so we refer to the dynamic evaluation optimizer for this experiment as "global RMS dynamic eval", since the decay rate was set to zero. Dynamic evaluation gave a 9% perplexity improvement to Transformer-XL on WikiText-103.

Model	# of params	test bpc
Hyper LSTM (Ha et al., 2017)	25M	1.34
HM-LSTM (Chung et al., 2017)	35M	1.32
Recurrent highway networks (Zilly et al., 2017)	46M	1.27
FS-LSTM (Mujika et al., 2017)	47M	1.25
AWD-LSTM (Merity et al., 2018a)	47M	1.23
Transformer + aux losses (Al-Rfou et al., 2018)	235M	1.06
Multiplicative LSTM (Section 3.5.2)	46M	1.24
Multiplicative LSTM + dynamic eval (Section 4.7.3)	46M	1.08
Transformer-XL * (Dai et al., 2019)	42M	1.053
Transformer-XL + global RMS dynamic eval + RMS decay	42M	1.012
Transformer-XL (Dai et al., 2019)	277M	**0.993**
Transformer-XL + dynamic eval (SGD)	277M	0.946
Transformer-XL + dynamic eval (global RMS, RMS decay)	277M	**0.940**

Table 6.1: Character-level cross-entropy (bits/char) on enwik8. As noted in Section 6.2.3, there is a slight difference in the data used in Transformer-XL results and previous work. *This result was from retraining this model from a new random initialization rather than using the exact pretrained model from (Dai et al., 2019).

The results on WikiText-103 are the first to apply dynamic evaluation with an adaptive softmax output layer, to our knowledge. Adaptive softmax reduces the computational expense of the output layer at the cost of giving the model less expressiveness at modeling rare words. When training a network from scratch, such a trade-off is sensible, since it is difficult to learn a good representation of rare words. However, when dynamically adapting to the recent sequence history, the adaptive softmax layer may make adapting to recent rare words more challenging. There is potential for future work improving the combination of dynamic evaluation and adaptive softmax, for instance by hybridizing it with the neural cache method (Grave et al., 2017b). The neural cache learns a non-parametric output layer that is independent of the network's output layer, which may potentially allow for more expressive adaptation to rare words in models with an adaptive softmax.

Model	# of params	test bpc
HM-LSTM (Chung et al., 2017)	35M	1.29
Recurrent highway networks (Zilly et al., 2017)	45M	1.27
Transformer + aux losses (Al-Rfou et al., 2018)	235M	1.13
Multiplicative LSTM (Section 3.5.3)	45M	1.27
Multiplicative LSTM + dynamic eval (Section 4.7.3)	45M	1.19
Transformer-XL (Dai et al., 2019)	277M	**1.085**
Transformer-XL + dynamic eval (SGD)	277M	1.042
Transformer-XL + dynamic eval (global RMS, RMS decay)	277M	**1.038**

Table 6.2: Character-level cross-entropy (bits/char) on text8.

6.2.5 Discussion

Dynamic evaluation was able to give moderate improvements to strong Transformer network baselines, and improves the state of the art on all three datasets evaluated. These results demonstrate that the types of long range dependencies used by dynamic evaluation and Transformers are somewhat different, as applying dynamic evaluation to Transformers leads to further improvements. These improvements are not nearly as large in terms of cross-entropy reduction as when dynamic evaluation has been applied to weaker models, suggesting that Transformers are more capable of capturing and modeling re-occurring patterns in sequences than past architectures. However, Transformers still struggle to fully exploit these repetitions, even in these experiments where training and testing data came from the same domain.

6.3 Out-of-domain language modeling

These experiments apply dynamic evaluation to Transformers that are trained and evaluated on datasets from different domains. The ability to generalize to data from a distribution that is different from what was observed during training is important for many real word tasks where training on data from the true testing distribution may be impossible. The capability of dynamic evaluation to generalize out of domain was briefly explored in Section 5.1, and in this section we extend it to a much stronger baseline that already has some out of domain generalization capability to start with.

Model	# of params	valid	test
LSTM+ neural cache (Grave et al., 2017b)	-	-	40.8
GCNN-14 (Dauphin et al., 2017)	-	-	37.2
QRNN (Merity et al., 2018a)	151M	32.0	33.0
LSTM + hebbian + cache (Rae et al., 2018)	-	29.7	29.9
Transformer + adaptive input (Baevski and Auli, 2019)	247M	19.8	20.5
Transformer-XL* (Dai et al., 2019)	257M	17.3	**18.1**
Transformer-XL + dynamic eval (SGD)	257M	16.3	17.0
Transformer-XL + dynamic eval (global RMS)	257M	15.8	**16.4**

Table 6.3: Word-level perplexity on WikiText-103. *We report our results using the pretrained model from (Dai et al., 2019) using a batch size of 1, and achieved a slightly lower perplexity than in the original paper (18.1 vs 18.3).

6.3.1 Background

There has recently been an interest in high capacity language models trained on much larger data sets than standard language modeling benchmarks (Radford et al., 2018, 2019; Devlin et al., 2018). These models have been shown to be able to generalize to a wide variety of NLP and language prediction tasks. Radford et al. (2019) specifically looked at the task of out-of-domain language modeling, where a language model trained on a large dataset is evaluated on text datasets from domains it has previously not seen. The training set used for their GPT-2 model consisted of 40 GB of text crawled from the web, with all wikipedia text excluded. Their model was then evaluated on common Wikipedia based language modeling test sets, including enwik8, text8, and WikiText-2. The largest GPT-2 variant, with 1.5B parameters, could outperform state of the art language models on some Wikipedia based tasks, despite having never seen Wikipedia data in its training set. GPT-2 seems to be able model re-occurring patterns to an extent, as evidenced by the conditional samples that GPT-2 can generate that repeat patterns in the conditioning text. This may partially explain why GPT-2 is able to perform so well on text data that it has never seen before; it is able to learn to model patterns on the fly from conditioning text. However, GPT-2 is limited by its fixed length memory window of 1024 tokens (GPT-2 uses byte pair encoding tokens (Sennrich et al., 2015), which is a type of subword unit).

6.3.2 Experiments

In this section, we experiment with augmenting GPT-2 with dynamic evaluation in the task of out-of-domain language modeling. Since the largest version of GPT-2 was not released at the time of writing, we use a smaller version of GPT-2 with 345M parameters[2]. This version of GPT-2 still obtained results on out-of-domain language modeling that were comparable to previous state of the art models trained in domain. Unlike the previous section, which uses Transformer-XL, GPT-2 uses a vanilla Transformer, meaning that segment level attention recurrence cannot be used. Near the beginning of the context, GPT-2 has very little context to make predictions, and therefore performs much worse. It is possible to only evaluate the last token of the sequence segment and discard the other predictions, making predictions one-by-one, using the full context window. This is very slow however, because the model re-processes a long context each time just to predict one token. As a compromise between these two extremes, when applying both static and dynamic evaluation, we used a context window length of 1024 tokens and evaluated on segments of the last 32 tokens of the context. This way, the model gets access to at least 992 tokens of context to make each prediction (except for on the start of the test set, when it starts predicting form the beginning), but it is 32 times faster than predicting one token at a time.

When applying dynamic evaluation, we tune the hyper-parameters for dynamic evaluation on held out webtext data[3], and evaluate on each task with the same hyper-parameters. This preserves the out-of-domain nature of the tasks (as opposed to tuning dynamic evaluation hyper-parameters separately for each task). Both for perplexity and memory efficiency reasons, we only applied dynamic evaluation to a subset of the parameters of the network. We found that applying dynamic evaluation to the input-output tied embedding matrix of GPT-2 required a much smaller learning rate and resulted in a worse perplexity overall than only adapting other parameters. Furthermore, adapting all the parameters of the network would not fit in memory on a single 11 GB GPU. Therefore, we only apply dynamic evaluation to the later layers in the network, although not the tied input-output layer. This requires less computation and memory because gradients do not need to back propagated as far. As in the section on Transformer-XL, we evaluate our model using SGD and global RMS style dynamic evaluation. The global RMS style dynamic evaluation did not include a decay rate, as this did not help performance. Bits per character and perplexities are calculated by

[2]GPT-2 345M is publicly available here `https://github.com/openai/gpt-2`

[3]This subset was released here: `https://github.com/openai/gpt-2-output-dataset`

Model	enwik8 bpc	text8 bpc	WikiText-103 ppl
GPT-2 345M	**1.017**	**1.066**	**26.6**
GPT-2 345M + dynamic eval (SGD)	0.893	1.017	23.0
GPT-2 345M + dynamic eval (global RMS)	**0.869**	**1.003**	**21.9**

Table 6.4: Out-of-domain language modeling

computing the entropy of the full test file under GPT-2's tokenization, and dividing this by the number of tokens used in the benchmark, to give numbers comparable to other models tokenized and trained directly on these data sets. The results are given in Table 6.4.

Dynamic evaluation improves GPT-2 by a significant margin on all tasks, especially enwik8, which contains a mix of markup and text. The result of 0.87 bits/char on enwik8 is also significantly better than the largest GPT-2 (GPT-2 1.5B), which achieved 0.93 bits/char without dynamic evaluation (Radford et al., 2019).

6.3.3 Discussion

These experiments show that dynamic evaluation's capability to help even very strong and multi-modal language models perform well on out of domain tasks. The GPT-2 baseline model was pretrained across many different data domains and can achieve a competitive performance on out of domain data when statically evaluated. However, applying dynamic evaluation to this model still greatly improves performance. This may be partially due to the limited context window in the self-attention; GPT-2 has access to the previous 1024 tokens when making predictions, but the test sequence on some tasks is over a million tokens long. GPT-2 may be able to adapt to the sequence within its context window, but dynamic evaluation allows the model to adapt to the entire sequence history, which contains much more information. Some recent work has focused on extending the context windows of Transformers by having more efficient memory. For instance Child et al. (2019) proposed a sparse attention mechanism that is $O(n\sqrt{n})$ computation in the length of the sequence, as opposed to standard self attention, which is $O(n^2)$. However, like standard self-attention, this approach is still $O(n)$ memory in the length of the sequence, which would be a problem for generalizing to very long sequences. Furthermore, explicitly learning the ability to use very long range dependencies in predictions would likely be difficult even for a model with a very

long attention window. Dynamic evaluation has the advantage of being able use very long range dependencies implicitly without being explicitly trained to do so; even a completely untrained model with dynamic evaluation would be able to benefit from long contexts due to the gradient descent updates.

6.4 Conclusion

This chapter applied dynamic evaluation to Transformer language models in two settings; the conventional language modeling setting, where the model is trained on evaluated on the same data, and out-of-domain language modeling, where an especially strong pretrained model is evaluated on data from a different distribution from what it saw during training. Transformers are able to use longer range statistical dependencies than the models from previous chapters, and may to some extent learn to adapt to what they have seen recently. Therefore, it was well motivated to examine whether dynamic evaluation could still be helpful in this setting. Dynamic evaluation gave significant improvements in both the in domain and out-of-domain language modeling. Thus, the work in this chapter further highlights the importance of adaptation ability in achieving robustness to surprise and better overall language modeling results.

Chapter 7

Conclusion

This thesis supports the claim that **giving language models additional flexibility to adapt to their inputs can improve their predictions by helping them recover from surprising tokens or sequences**. Multiplicative LSTM made LSTMs more adaptive at the token level by allowing them to have a different hidden-to-hidden transition function for each possible input token. Dynamic evaluation enabled models to adapt more effectively to sequences by using gradient descent to fit to the recent sequence history. Both of these adaptive methods led to improvements in language modeling, and in both cases, these improvements were at least partially explained by the ability to recover from surprising inputs. These results highlight the importance of adaptability in language modeling.

7.1 Thesis contributions

The main contributions of this thesis are:

- presenting multiplicative LSTM as an architectural enhancement to LSTMs for language modeling that allow them to have a different recurrent transition function for each possible input token. The improved adaptation ability resulting from this modification gave empirical improvements to language modeling in our experiments, which showed
  - mLSTM could outperform well tuned LSTMs and previously existing neural architectures at character level language modeling, using significantly less depth (Section 3.5.2). While depth can give models a greater ability to adapt to their inputs by making the model more complex, mLSTM can do this in

a simpler way, requiring fewer sequential computation steps (thus allowing faster computation via parallelization), and achieving better overall results.

  - a byte-level mLSTM could match perplexities of similarly trained word-level LSTMs (Section 3.5.4). Word level language models are unable to model out-of-vocabulary words, which is a major drawback for language prediction tasks that require this ability. Showing that a byte-level mLSTM with the flexibility to model almost any type of text can perform competitively with a word-level LSTM in a limited vocabulary setting is another demonstration of the advantage of mLSTM vs. LSTM. Showing that a byte-level model can achieve strong results also motivates using benchmarks and models with broader tokenizations to be able to model open vocabulary more effectively.
  - experiments showing mLSTM can make better predictions vs. LSTM immediately after an unexpected token (Section 3.5.2). This contributes to the understanding that adaptation ability helps models recover from surprise.

- exploring dynamic evaluation, a previously existing but not well understood and sparingly used method, as a way of adapting auto-regressive sequence models to their predictions. We found that through this adaptation, dynamic evaluation could give large improvements to language modeling to many different neural architectures in several different settings. The specific contributions to dynamic evaluation include:

  - developing a dynamic evaluation method to improve a variety of state of the art models at character and word-level language modeling (Sections 4.5 4.7 and 6.2). These results present evidence that adaptation ability helps language models make better predictions.
  - demonstration of dynamic evaluation's ability to generalize to out of domain text prediction (Sections 5.1 and 6.3). These results show that dynamic evaluation's adaptation ability make it especially robust to surprising sequences. Normal models struggle when evaluated on sequences from a different language from the training language, whereas dynamic evaluation greatly improves both LSTM and Transformer language models in this scenario.
  - showing the qualitative ability of dynamic evaluation to generate samples that repeat patterns in the conditioning text (Section 5.2). The ability to model repeating patterns gives a mechanism by which dynamic evaluation

can make models more robust to surprising sequences. If the model observes text with statistical regularities it was not expecting, it will always fail at predicting it the first time. However, if it is able to adapt, if it sees text with the same statistical regularities later in the sequence, it will be able to predict it better.

- analysis demonstrating that dynamic evaluation gains a large advantage on repeating words, and can also generalize from related words (Section 5.3). This analysis gives further evidence that dynamic evaluation gains its advantage from adapting to re-occurring patterns in sequences. We also hypothesized a specific mechanism by which dynamic evaluation generalizes to related words; related words tend to occur in similar contexts, and as a result, learn similar output embeddings during training. This means that prediction errors on related words will result in similar gradient signals, meaning that for a small enough learning rate, an update to one word will generalize to words with similar word embeddings.
- in depth exploration of dynamic evaluation optimizers (Section 5.4), and development of novel optimization algorithms that improve performance in a dynamic evaluation setting, and may have applications to other adaptation settings. We introduce a decay prior to prevent parameters from straying too far from what they learned during training. We also develop a new RMSprop-like method for dynamic evaluation that benefits from using gradient statistics from the training set rather than from the adaptation data.
- dynamic evaluation methods that are more computationally and memory efficient (Section 4.6). Applying dynamic evaluation to update all of a model's parameters uses a large amount of memory when combined with test time mini-batching, giving it some computational disadvantages compared to other models. Achieving strong adaptation by only adapting a small subset of the weights of the model makes dynamic evaluation practical to settings that require mini-batching.
- analysis of the sequence lengths at which dynamic evaluation gains its advantage (Section 5.1). The ability to exploit long range statistical dependencies is important for fully adapting to long sequences, and of general interest to researchers working on language modeling and sequence modeling problems. We show that dynamic evaluation can use long contexts

especially well.

- comparison of language models that use dynamic evaluation (and mLSTM) with human language predictors (Section 5.5), showing that they could perform on par with the best human predictors, but worse than an ensemble of human predictors. This result helps put other results in this thesis into context, as humans would be expected to be a strong baseline.
- application of dynamic evaluation to polyphonic music prediction (Section 4.8). While this thesis mainly focuses on text language modeling, other types of sequences contain re-occurring patterns too. We show in these results that making music sequence models more adaptive with dynamic evaluation can also improve their predictions.

7.2 Future work

Two main categories of future work that could extend the claim in this thesis are:

1. directly applying the methods proposed in this thesis to other sequence prediction and generation problems. This thesis demonstrated that dynamic evaluation and multiplicative LSTM give improvements to next token prediction in text. However, improvements to language modeling often generalize to text generation, and other more general problems in NLP and sequence modeling.

2. developing new neural architectures that are flexible and adaptive. This thesis highlights the utility of neural architectures in language modeling with greater flexibility to adapt to their inputs. This principle may be extendable to create better architectures for language modeling or other problems in deep learning.

Applying mLSTM and dynamic evaluation directly to other problems is the most straightforward way for this work to be extended. Fortunately, at the time of writing of this thesis, many extensions of mLSTM to other problems have already been carried out in other work. mLSTM has found applications to problems that require text generation rather than just ground truth prediction, including machine translation (Pinnis and Kalnins, 2018), conversational AI (Krause et al., 2017a), and abstractive summarization (Chu and Liu, 2018). Work by Radford et al. (2017) demonstrated that large mLSTMs pretrained as language models could learn strong representations for sentiment, and used these representations to achieve state of the art sentiment analysis results. Outside

of text modeling, mLSTM was also found to learn useful representations for proteins by modeling them as strings of amino acids (Alley et al., 2019).

Dynamic evaluation's most direct applications are to settings where ground truth is available. This includes predictive keyboard and predictive search engine, where dynamic evaluation could be used to adapt to recent data. Dynamic evaluation demonstrated the ability to generate samples that repeated patterns from the conditioning text in Chapter 5, which could be useful for conditional sequence generation. Dynamic evaluation could be applied to conditional language and music generation, where the ground truth of the conditioning text is available. This is the case in dialogue generation for instance.

Dynamic evaluation may also have applications to abstractive summarization, where the ground truth source sequence is available. Summarization entails mapping a longer passage of text to a shorter summary. The distribution of text in the passage and the summary are not exactly the same, but are closely related as they will cover the same topics. The distributions are related enough that pure language models (which use the same parameters to process source text and generate target text) have had success at this task (Radford et al., 2019; Liu et al., 2018). In summarization models that use a language model (with a shared encoder and decoder), dynamic evaluation could be applied to fit the model to the passage before generating the summary. This would bias the model towards words and topics found in the passage, which could be especially useful for summarizing longer passages, when it may be difficult for standard models to utilize the full context.

While dynamic evaluation has clearer benefits to tasks such as language modeling where the ground truth is available, future work will be required to determine how these improvements generalize to tasks that would require fitting to generated tokens. Dynamic evaluation could potentially be beneficial to tasks such as speech recognition and machine translation over longer contexts, as similarly motivated adaptation approaches have given improvements in these settings. For instance, in past work adaptive n-grams have been used to improve speech recognition (Jelinek et al., 1991) and neural caching has been used to improve machine translation (Tu et al., 2017; Kuang et al., 2017). Dynamic evaluation would likely assign a higher probability to correct transcriptions for these tasks, since it would be able to capture repeating patterns and style. However, there are potential problems with decoding dynamically evaluated models, since they may assign higher probabilities to incorrect transcriptions that repeat incorrect words many times. This problem could result from fitting models to repeat patterns from

incorrect transcriptions from earlier in the sequence. One conservative approach to address this might be to use dynamic evaluation to re-weight an n-best list generated by another model, as constraining dynamic evaluation to limited options would prevent it from repeating the same tokens over and over. The success of dynamic evaluation for language modeling indicates that the architectures used for speech recognition and machine translation systems struggle to correctly predict repeating patterns over long contexts, and approaches that use dynamic evaluation or other related methods to do this are well motivated.

Dynamic evaluation also appears to applications outside of text modeling. This thesis demonstrated that dynamic evaluation could help with music prediction, and there may be applications of dynamic evaluation in other auto-regressive sequence modeling tasks such as video prediction. Furthermore, dynamic evaluation and approaches that extend on dynamic evaluation with online meta-learning have proven useful in model based reinforcement learning in a dynamically changing environment (Nagabandi et al., 2019).

Beyond direct applications to tasks, this thesis suggests future work in developing new models with added flexibility to adapt to their inputs. The architectural enhancements in mLSTM and dynamic evaluation could both be useful starting points for achieving this. The input dependent hidden-to-hidden transition matrix in mLSTM could be used in other architectures; for instance, in Transformers, the weight matrices in the feed forward layers could use a similar input dependent factorization.

Dynamic evaluation builds gradients into the architecture in a way that could potentially be extended to other adaptive methods. Gradient descent is typically used as a method for learning global information about a distribution i.e. by training a neural network. However, the idea of using gradient descent as a memory that can exploit local information is still under-explored, and may have applications beyond dynamic evaluation. It is clear that gradient descent is capable of encoding knowledge that can be useful for tasks like question answering. For instance, the GPT-2 model from Radford et al. (2019) was able to answer questions that never occurred in the training data such as “who wrote the book the origin of the species?” purely as a function of the learned weights of the network, without any context. Presumably, at some point during training the network encountered a passage about Charles Darwin and learned to associate him with the “origin of the species” via a gradient descent update on a language modeling objective function. While the non-parametric memory found in Transformers is powerful for encoding memories for relatively short contexts, gradient descent could

potentially encode memories over entire data sets, as it appears to successfully do in the above Charles Darwin example. The high level idea of exploring language models with gradient descent built into the model, which could be trained via meta-learning approaches such as MAML (Finn et al., 2017), is an interesting future direction.

Bibliography

Aharoni, Z., Rattner, G., and Permuter, H. (2017). Gradual learning of deep recurrent neural networks. *arXiv:1708.08863*.

Al-Rfou, R., Choe, D., Constant, N., Guo, M., and Jones, L. (2018). Character-level language modeling with deeper self-attention. *arXiv:1808.04444*.

Allan, M. and Williams, C. (2005). Harmonising chorales by probabilistic inference. In *NeurIPS*, pages 25–32.

Alley, E., Khimulya, G., Biswas, S., AlQuraishi, M., and Church, G. M. (2019). Unified rational protein engineering with sequence-based deep representation learning. *Nature methods*, 16(12):1315–1322.

Ba, J., Hinton, G. E., Mnih, V., Leibo, J. Z., and Ionescu, C. (2016a). Using fast weights to attend to the recent past. In *NeurIPS*, pages 4331–4339.

Ba, J., Kiros, J. R., and Hinton, G. E. (2016b). Layer normalization. *arXiv:1607.06450*.

Baevski, A. and Auli, M. (2019). Adaptive input representations for neural language modeling. *ICLR*.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. *ICLR*.

Bellegarda, J. R. (2004). Statistical language model adaptation: review and perspectives. *Speech Communication*, 42(1):93–108.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

Bengio, Y., Simard, P., and Frasconi, P. (1994a). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166.

Bengio, Y., Simard, P., and Frasconi, P. (1994b). Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166.

Boulanger-Lewandowski, N., Bengio, Y., and Vincent, P. (2012). Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *ICML*.

Bradbury, J., Merity, S., Xiong, C., and Socher, R. (2017). Quasi-recurrent neural networks. *ICLR*.

Bridle, J. S. (1990). Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing*, pages 227–236. Springer.

Brown, P. F., Desouza, P. V., Mercer, R. L., Pietra, V. D., and Lai, J. C. (1992a). Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479.

Brown, P. F., Pietra, V. D., Mercer, R. L., Pietra, S. D., and Lai, J. C. (1992b). An estimate of an upper bound for the entropy of english. *Computational Linguistics*, 18(1):31–40.

Child, R., Gray, S., Radford, A., and Sutskever, I. (2019). Generating long sequences with sparse transformers. *arXiv:1904.10509*.

Chu, E. and Liu, P. J. (2018). Unsupervised neural multi-document abstractive summarization of reviews.

Chung, J., Ahn, S., and Bengio, Y. (2017). Hierarchical multiscale recurrent neural networks. *ICLR*.

Cooijmans, T., Ballas, N., Laurent, C., and Courville, A. (2017). Recurrent batch normalization. *ICLR*.

Cover, T. and King, R. (1978). A convergent gambling estimate of the entropy of english. *IEEE Transactions on Information Theory*, 24(4):413–421.

Dai, Z., Yang, Z., Yang, Y., Cohen, W. W., Carbonell, J., Le, Q. V., and Salakhutdinov, R. (2019). Transformer-XL: Attentive language models beyond a fixed-length context. *arXiv:1901.02860*.

Dauphin, Y. N., Fan, A., Auli, M., and Grangier, D. (2017). Language modeling with gated convolutional networks. In *ICML*.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, 14(2):179–211.

Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*, pages 1126–1135.

Fortunato, M., Blundell, C., and Vinyals, O. (2017). Bayesian recurrent neural networks. *arXiv:1704.02798*.

Gal, Y. and Ghahramani, Z. (2016). A theoretically grounded application of dropout in recurrent neural networks. In *NeurIPS*, pages 1019–1027.

Gers, F. A., Schmidhuber, J., and Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12:2451–2471.

Grave, E., Joulin, A., Cissé, M., and Jégou, H. (2017a). Efficient softmax approximation for gpus. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1302–1310. JMLR. org.

Grave, E., Joulin, A., and Usunier, N. (2017b). Improving neural language models with a continuous cache. *ICLR*.

Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv:1308.0850*.

Graves, A., Mohamed, A., and Hinton, G. E. (2013). Speech recognition with deep recurrent neural networks. In *Proceddings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649. IEEE.

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. *arXiv:1410.5401*.

Gu, J., Lu, Z., Li, H., and Li, V. O. K. (2016). Incorporating copying mechanism in sequence-to-sequence learning. *arXiv:1603.06393*.

Ha, D., Dai, A., and Lee, Q. (2017). Hypernetworks. *ICLR*.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Hermans, M. and Schrauwen, B. (2013). Training and analysing deep recurrent neural networks. In *NeurIPS*, pages 190–198.

Hochreiter, S., Bengio, Y., Frasconi, P., and Schmidhuber, J. (2001). Gradient flow in recurrent nets: The difficulty of learning long-term dependencies. In Kremer and Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9:1735–1780.

Hutter, M. (2012). The human knowledge compression contest. *URL http://prize.hutter1.net*.

Inan, H., Khosravi, K., and Socher, R. (2017). Tying word vectors and word classifiers: A loss framework for language modeling. *ICLR*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv:1502.03167*.

Jelinek, F., Merialdo, B., Roukos, S., and Strauss, M. (1991). A dynamic language model for speech recognition. In *HLT*, volume 91, pages 293–295.

Johnson, D. D. (2017). Generating polyphonic music using tied parallel networks. In *International conference on evolutionary and biologically inspired music and art*, pages 128–143. Springer.

Kalchbrenner, N., Espeholt, L., Simonyan, K., Oord, A., Graves, A., and Kavukcuoglu, K. (2016). Neural machine translation in linear time. *arXiv:1610.10099*.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv:1609.04836*.

Khandelwal, U., He, H., Qi, P., and Jurafsky, D. (2018). Sharp nearby, fuzzy far away: How neural language models use context. *arXiv:1805.04623*.

Kim, Y., Jernite, Y., Sontag, D., and Rush, A. M. (2016). Character-aware neural language models. In *Thirtieth AAAI Conference on Artificial Intelligence*.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv:1412.6980*.

Koehn, P. (2005). Europarl: A parallel corpus for statistical machine translation. In *MT Summit*, volume 5, pages 79–86.

Krause, B. (2015). Optimizing and contrasting recurrent neural network architectures. Master's thesis, The University of Edinburgh. `https://arxiv.org/abs/1510.04953`.

Krause, B., Damonte, M., Dobre, M., Duma, D., Fainberg, J., Fancellu, F., Kahembwe, E., Cheng, J., and Webber, B. (2017a). Edina: Building an open domain socialbot with self-dialogues. *Alexa Prize Proceedings*.

Krause, B., Kahembwe, E., Murray, I., and Renals, S. (2018). Dynamic evaluation of neural sequence models. *ICML*.

Krause, B., Kahembwe, E., Murray, I., and Renals, S. (2019). Dynamic evaluation of transformer language models. *arXiv:1904.08378*.

Krause, B., Lu, L., Murray, I., and Renals, S. (2016). Multiplicative LSTM for sequence modelling. *arXiv:1609.07959*.

Krause, B., Murray, I., Renals, S., and Lu, L. (2017b). Multiplicative LSTM for sequence modelling. *ICLR Workshop track*.

Kuang, S., Xiong, D., Luo, W., and Zhou, G. (2017). Cache-based document-level neural machine translation. *arXiv:1711.11221*.

Kuhn, R. (1988). Speech recognition and the frequency of recently used words: A modified Markov model for natural language. In *Proceedings of the 12th conference on Computational linguistics-Volume 1*, pages 348–350. Association for Computational Linguistics.

Kuhn, R. and De Mori, R. (1992). Corrections to "a cache-based language model for speech recognition". *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (6):691–692.

Larochelle, H. and Murray, I. (2011). The neural autoregressive distribution estimator. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 29–37.

Lin, C. and Och, F. J. (2004). Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 605. Association for Computational Linguistics.

Liu, H., Simonyan, K., and Yang, Y. (2018). Darts: Differentiable architecture search. *arXiv:1806.09055*.

Malone, D. (1948). *Jefferson the Virginian*, volume 1. St. Martin's Press.

Marcus, G. (2001). The algebraic mind.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):313–330.

Martens, J. (2010). Deep learning via Hessian-free optimization. In *ICML*, pages 735–742.

Melis, G., Dyer, C., and Blunsom, P. (2018). On the state of the art of evaluation in neural language models. *ICLR*.

Merity, S., Keskar, N. S., and Socher, R. (2018a). An analysis of neural language modeling at multiple scales. *arXiv:1803.08240*.

Merity, S., Keskar, N. S., and Socher, R. (2018b). Regularizing and optimizing LSTM language models. *ICLR*.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. (2017). Pointer sentinel mixture models. *ICLR*.

Mikolov, T. (2012). *Statistical language models based on neural networks*. PhD thesis, Brno University of Technology.

Mikolov, T., Joulin, A., Chopra, S., Mathieu, M., and Ranzato, M. (2014). Learning longer memory in recurrent neural networks. *arXiv:1412.7753*.

Mikolov, T., Karafiát, M., Burget, L., Cernockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Interspeech*, volume 2, page 3.

Mikolov, T., Sutskever, I., Deoras, A., Le, H., Kombrink, S., and Cernocky, J. (2012a). Subword language modeling with neural networks. *preprint (http://www.fit.vutbr.cz/ imikolov/rnnlm/char.pdf)*.

Mikolov, T., Sutskever, I., Deoras, A., Le, H., Kombrink, S., and Cernocky, J. (2012b). Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)*.

Mikolov, T. and Zweig, G. (2012). Context dependent recurrent neural network language model. *SLT*, 12:234–239.

Mujika, A., Meier, F., and Steger, A. (2017). Fast-slow recurrent neural networks. *arXiv:1705.08639*.

Nagabandi, A., Finn, C., and Levine, S. (2019). Deep online learning via meta-learning: Continual adaptation for model-based rl. *ICLR*.

Nallapati, R., Zhou, B., Gulcehre, C., Xiang, B., et al. (2016). Abstractive text summarization using sequence-to-sequence rnns and beyond. *arXiv preprint arXiv:1602.06023*.

Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate o(1/sqr(k)). *Soviet Mathematics Doklady*, 27:372–376.

Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. *arXiv:1609.03499*.

Ororbia II, A. G., Mikolov, T., and Reitter, D. (2017). Learning simpler language models with the differential state framework. *Neural Computation*.

Papineni, K., Roukos, S., Ward, T., and Zhu, W. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (2013a). How to construct deep recurrent neural networks. *arXiv:1312.6026*.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013b). On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch. *NeurIPS*.

Pinnis, M. and Kalnins, R. (2018). Developing a neural machine translation service for the 2017-2018 european union presidency. In *Proceedings of the 13th Conference of the Association for Machine Translation in the Americas (Volume 2: User Papers)*, pages 72–83.

Poliner, G. E. and Ellis, D. (2006). A discriminative model for polyphonic piano transcription. *EURASIP Journal on Advances in Signal Processing*, 2007(1):048317.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17.

Polyak, B. T. and Juditsky, A. B. (1992). Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855.

Press, O. and Wolf, L. (2017). Using the output embedding to improve language models. *EACL 2017*, page 157.

Prickett, B. (2017). Vanilla sequence-to-sequence neural nets cannot model reduplication.

Radford, A., Jozefowicz, R., and Sutskever, I. (2017). Learning to generate reviews and discovering sentiment. *arXiv:1704.01444*.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training. *URL https://openai.com/blog/language-unsupervised/*.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. *URL https://openai.com/blog/better-language-models/*.

Rae, J. W., Dyer, C., Dayan, P., and Lillicrap, T. P. (2018). Fast parametric learning with activation memorization. *ICML*.

Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Proceedings of the IEEE international conference on neural networks*, volume 1993, pages 586–591. San Francisco.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407.

Robinson, T. (1994). An application of recurrent nets to phone probability estimation. *IEEE transactions on Neural Networks*, 5(2):298–305.

Rocki, K. (2016a). Recurrent memory array structures. *arXiv:1607.03085*.

Rocki, K. (2016b). Surprisal-driven feedback in recurrent networks. *arXiv:1608.06027*.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533.

Salimans, T. and Kingma, D. P. (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *NeurIPS*, pages 901–909.

Saxe, A. M., McClelland, J. L., and Ganguli, S. (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv:1312.6120*.

Schmidhuber, J. (1992). Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139.

Sennrich, R., Haddow, B., and Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv:1508.07909*.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423.

Shannon, C. E. (1951). Prediction and entropy of printed english. *Bell system technical journal*, 30(1):50–64.

Siegelmann, H. T. and Sontag, E. D. (1995). On the computational power of neural nets. *Journal of computer and system sciences*, 50(1):132–150.

Sprechmann, P., Jayakumar, S., Rae, J., Pritzel, A., Badia, A. P., Uria, B., Vinyals, O., Hassabis, D., Pascanu, R., and Blundell, C. (2018). Memory-based parameter adaptation. *ICLR*.

Sukhbaatar, S., Weston, J., Fergus, R., et al. (2015). End-to-end memory networks. In *NeurIPS*, pages 2431–2439.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. E. (2013). On the importance of initialization and momentum in deep learning. In *ICML*, pages 1139–1147.

Sutskever, I., Martens, J., and Hinton, G. E. (2011). Generating text with recurrent neural networks. In *ICML*, pages 1017–1024.

Sutskever, I., Vinyals, O., and Le, Q. V. V. (2014). Sequence to sequence learning with neural networks. In *NeurIPS*, pages 3104–3112.

Tieleman, T. and Hinton, G. E. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2).

Tu, Z., Liu, Y., Shi, S., and Zhang, T. (2017). Learning to remember translation history with a continuous cache. *arXiv:1711.09367*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *NeurIPS*, pages 5998–6008.

Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer networks. In *NeurIPS*, pages 2692–2700.

Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In *ICML*, pages 1058–1066.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78:1550–1560.

Williams, R. J. and Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, 2(4):490–501.

Witten, I., Neal, R., and Cleary, J. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540.

Wu, Y., Zhang, S., Zhang, Y., Bengio, Y., and Salakhutdinov, R. (2016). On multiplicative integration with recurrent neural networks. In *NeurIPS*.

Yang, Z., Dai, Z., Salakhutdinov, R., and Cohen, W. W. (2018). Breaking the softmax bottleneck: a high-rank RNN language model. *ICLR*.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., and Le, Q. V. (2019). Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*, pages 5754–5764.

Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *arXiv:1409.2329*.

Zellers, R., Holtzman, A., Rashkin, H., Bisk, Y., Farhadi, A., Roesner, F., and Choi, Y. (2019). Defending against neural fake news. In *Advances in Neural Information Processing Systems*, pages 9051–9062.

Zhang, S., Wu, Y., Che, T., Lin, Z., Memisevic, R., Salakhutdinov, R., and Bengio, Y. (2016). Architectural complexity measures of recurrent neural networks. In *NeurIPS*. arXiv:1602.08210.

Zilly, J. G., Srivastava, R. K., Koutník, J., and Schmidhuber, J. (2017). Recurrent highway networks. *ICLR*.

Zoph, B. and Le, Q. V. (2017). Neural architecture search with reinforcement learning. *ICLR*.