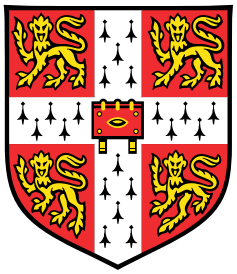
Memory Networks for Language Modelling



Oscar Chen

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Master of Philosophy

St Edmund's College

August 2017

This dissertation is dedicated to my mother, and the loving support she has provided me my entire life. Thanks Mom.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains 14,897 words excluding bibliography, photographs and diagrams but including tables, footnotes, and appendices.

Oscar Chen
August 2017

Acknowledgements

Firstly, I would like to acknowledge my supervisor, Professor Mark Gales, for providing me with the invaluable opportunity to work on this project, and his continued support and guidance throughout my time at Cambridge. His objective criticism of my work has been invaluable for me towards understanding my own work at a deeper level, and I am tremendously thankful for the opportunity to work with such a brilliant advisor.

Secondly, I would like to acknowledge both Dr. Anton Ragni and Dr. Jeff Chen, for their invaluable support, provision of tools and helpful advice, to which was key for the development of this project. The constructive feedback I have received from Anton has been crucial towards improving the quality of my work, and it has been a wonderful experience working with him on developing the ideas presented in this dissertation.

I would also like to give thanks to all of those not mentioned here who have provided their expertise and support throughout the duration of this project. Special thanks to my fellow course-mates in the MLSALT group for having made this such an enjoyable experience.

I would also like to thank my college roommates Callum Talbot-Cooper, Stefan Hosein and Goeric Huybrechts for reading and providing helpful feedback on my dissertation. Additionally, their friendship throughout this year has provided me with many fond memories and helped make Cambridge an unforgettable experience.

Finally, I would like to give a special thanks to my family whose continued love and care provided the foundations for me succeed. Their persistent belief in my ability, has, and always will be a huge factor in achieving my goals.

Abstract

Recurrent neural networks (RNNs) have achieved state-of-the-art results in problems such as language modeling and speech recognition. This is primarily due to their ability to encode a long word history into a compact hidden state. Furthermore, RNNs can be contextualized with topics by utilizing the features learned by a topic model (such as Latent Dirichlet Allocation) as additional inputs to the model. This dissertation proposes an alternative method that make use of attention and memory to learn the global context given by the topics – this approach is encapsulated in a model referred to as an Active Memory Network (AMN). A topic can be viewed as a specific interpretation of the word history, which suggests that a model can perform *implicit* topic modeling by simply tracking multiple word history representations. This can be performed by storing those representations within memory cells, and using attention to select the most likely representation at each time-step. Experiments were conducted on the Penn Tree Bank and BBC Multi-Genre Broadcast News corpora, with AMN significantly outperforming comparable GRU and LSTM models when evaluated in terms of perplexity. An in-depth analysis of the learned attention mechanism was then performed, which showed that AMN is capable of learning a plausible attention mechanism for performing implicit topic modeling. Finally, AMN was applied to N-best hypotheses rescoring for speech recognition, where it obtained word error rate improvements over comparable RNN and GRU models.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Motivation

The use of context is fundamental to human language understanding. A simple example to illustrate this phenomenon is provided with the following example:

> the markets were open for most of the morning but closed in the afternoon due to the [BLANK]

It can be quite challenging to predict a list of hypotheses for [BLANK] given the limited sentence context. Generic guesses such as "snow-storm" or "car-accident" can be made, but there is inevitably a high-degree of uncertainty associated with each of these guesses. However, suppose the following sentence had been observed instead:

> the dow jones industrial average fell rapidly today after the nasdaq stock market suffered from a computer network shutdown <eos> the markets were open for most of the morning but closed in the afternoon due to the [BLANK]

With this additional context from the previous sentence, it can be assumed that [BLANK] is a word such as "crash" or "outage" to a relatively high degree of confidence. This however is based on the assumption that the second sentence ("the markets were open ...") directly follows from the first sentence ("the dow jones industrial ..."), which may not be necessarily true. This therefore has significant implications in the prediction of [BLANK], since different sets of word hypotheses are generated depending on the interpretation of the word history. Crucially, the interpretation of the word-history provides an *implicit topic* for the understanding of the text (generic topic versus finance).

1.1.1 Language Models for Corpora with Diverse-Topics

For language models (LMs) based on neural networks, practitioners would typically incorporate global context understanding by appending the topic-features extracted from a topic model (such as Latent Dirichlet Allocation) to the word inputs [2, 3]. This would result in modest performance gains in downstream tasks like speech recognition. But ideally, the language model should be able to learn such context-information without additional supervision from another model.

Motivated by the simple example above, the goal is to construct a language model that is capable of maintaining multiple interpretations of the word-history after reading a set of sentences. This then allows the model to capture the implicit topics which exists in the various word history representations.

To this end, a novel neural network architecture, referred to as an Active Memory Network, was applied. This model uses a recurrent attention mechanism to actively attend to multiple, dynamic memory cells. Each of these memory cells holds a distinct context representation of the word history, where the most likely context representation is selected by the model using soft-attention.

1.1.2 LM Applications to Speech Recognition

Although language modeling is of interest in its own right, they also have direct applications for downstream tasks such as ASR. Thus, improved language modeling performance is of great interest given their potential for improving ASR performance.

One commonly used application for an LM in an ASR pipeline is to use it for Viterbi decoding with an acoustic model [4]. N-best hypotheses can be generated for rescoring using another separate LM. This is a relatively simple approach that has been shown to yield significant ASR performance gains.

1.2 Organization of Dissertation

The contents of this dissertation is organized as follows. Chapter 2 provides an overview of the neural networks literature. Chapter 3 provides an overview of the related language modeling literature. Chapter 4 presents the model architecture of AMN, including an overview of relevant training techniques. Chapter 5 discusses the relevant implementation details for the experiments conducted. Chapter 6 presents the experimental results and analysis from language modeling experiments on the PTB and MGB datasets. Chapter 7 presents the results from the speech recognition experiments on MGB3. And finally, Chapter 8 provides

a summary of the main results and conclusions of this dissertation, including a discussion on future work.

Chapter 2

Neural Networks

2.1 Feedforward Neural Networks (FNNs)

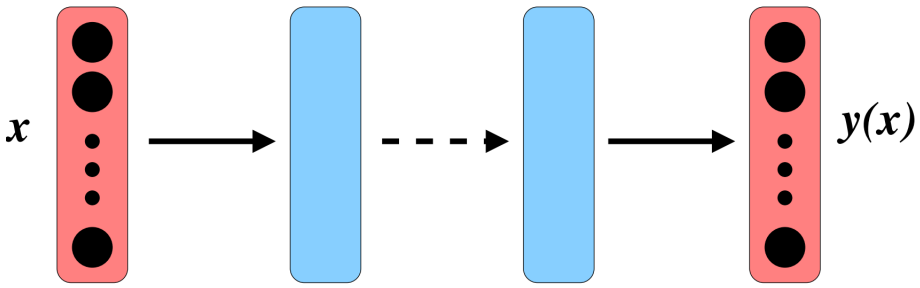


Fig. 2.1 A feedforward neural net mapping input x to output y.

Classically, a neural network is a highly nonlinear function that maps an input vector x to an output vector y. Typically, neural networks are referred to in their feedforward variant, where one or more hidden layers are applied successively to the input vector in a compositional manner. For multi-class classification problems, y is typically an one-hot vector with the k^{th} index (representing the k^{th} class) set to 1 and the rest set to 0.

The output of a hidden layer – which is the hidden-state vector h – is computed by performing matrix multiplication with the input x, followed by a nonlinear, continuous and differentiable activation function f applied element-wise.

$$h = f(W_x x + b_x) \tag{2.1}$$

where W_x is the weight-matrix and b_x is the bias. Sometimes it is implicitly assumed that $x_0 = 1$ and the first column of W_x is the bias b_x, in which case b_x is omitted. Note that the value $W_x x + b_x$ is commonly referred to as the *pre-activation* vector, whereas $f(W_x x + b_x)$ is referred to as the *post-activation* vector.

Another activation function can then be applied on the output of the hidden layer to compute the pre-activation for the final output layer.

$$q = W_h h + b_h \tag{2.2}$$

For multi-category classification, the output layer is usually implemented in the form of a softmax layer, which returns a vector where the elements sum to 1.

$$\hat{y} = \text{Softmax}(q) = \frac{\exp(q)}{\sum_j \exp(q_j)} \tag{2.3}$$

where $\hat{y}$ is the vector of predicted outputs. Sometimes the softmax function is defined in terms of scalar-inputs with the notation:

$$\hat{y}_k = \text{Softmax}(q_k) \tag{2.4}$$

but it should be clear from context whether this function should be returning a scalar or a vector.

2.2 Activation Functions

The linear, logistic sigmoid, tanh, and rectified-linear units (ReLu) are the most commonly used activation functions for neural network models.

- The linear function simply returns the result from multiplying with a weight matrix W.
- The sigmoid function has mean 0.5 and a range of $[0, 1]$.
- The tanh function has mean 0.0 and a range of $[-1, 1]$.
- The ReLu activation function is specified by $\max(x, 0)$ and always returns a positive value.

Note that the activation values for both sigmoid and tanh may "saturate" near either end of their ranges when the magnitude of the input is huge, which results in gradients with extreme values. This can sometimes be problematic for training [5].

On the other hand, ReLu activation functions have been found to perform exceptionally well in feed-forward networks concerning both convergence speed and performance [6]. The derivative for the positive parts of their activation range is constant, which allows ReLu-based models to effectively deal with the problem of saturating gradients when training deep networks [6].

2.3 Recurrent Neural Networks (RNNs)

For sequential inputs of the form $\mathbf{x} = \{x_1, ..., x_T\}$, one can implement recurrent neural networks (RNN) that reads the sequence one input at a time. At each time-step t, the next hidden-state h_t is computed by passing the input x_t and the hidden-state at the current time-step h_t through a non-linearity f.

$$h_{t+1} = f(h_t, x_t) \tag{2.5}$$

Typically, this reduces to:

$$h_{t+1} = f(W_x x_t + W_h h_t + b) \tag{2.6}$$

where W_x is the input-to-hidden weight matrix, W_h is the hidden-to-hidden weight matrix, and b is the bias. In the research literature, this is sometimes explicitly referred to as an RNN with an Elman-architecture [7], although this is typically implied by the model equations.

2.3.1 Long-term Short-term Memory

RNNs are in theory capable of learning extremely long-term word dependencies across time – but in practice, training RNNs to learn such dependencies is exceptionally difficult. This is primarily due to the phenomenon of vanishing/exploding gradients [8], where the magnitude of the gradients becomes exponentially large or small during training with gradient descent (discussed in 2.5).

The Long-term Short-term Memory (LSTM) unit [9], introduced almost two decades ago, presents an alternative approach towards solving the vanishing/exploding gradient problem. Within the LSTM architecture, a set of gates is used to control the flow of information. Since then, researchers have developed many LSTM variants, with the most commonly-used one being the LSTM model described in [10], which is described here.

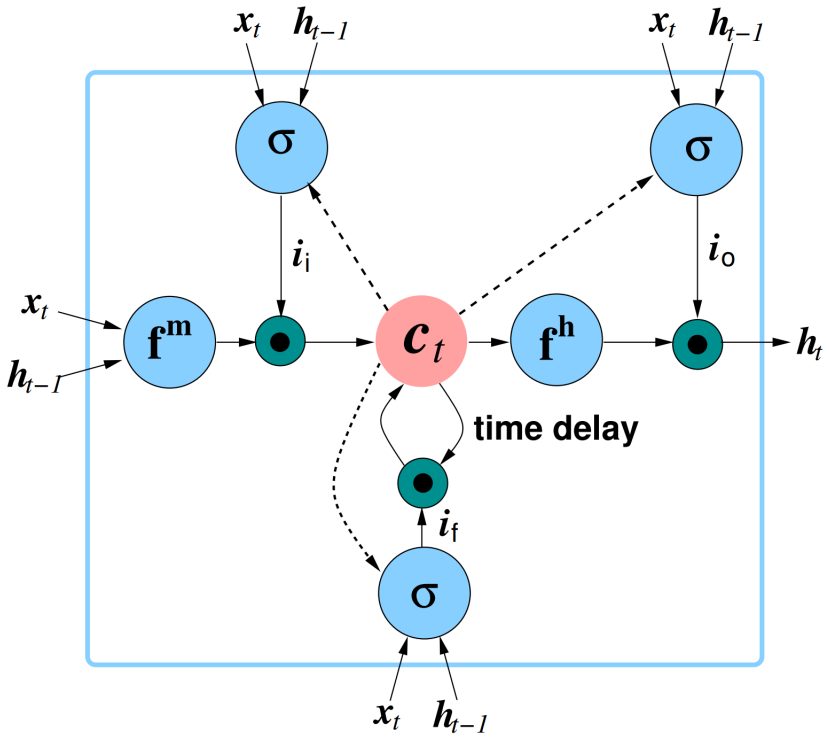


Fig. 2.2 High-level diagram of the LSTM architecture. i_i, i_f and i_o correspond to the input gate, forget gate and output gate respectively. c is the cell activation vector. $\mathbf{f^m}$ and $\mathbf{f^h}$ are the activation functions, which are usually set to the tanh function.

$$i_t = \text{sigmoid}(W_{x^i} x_t + W_{h^i} h_{t-1} + W_{c^i} c_{t-1} + b_i) \tag{2.7}$$
$$f_t = \text{sigmoid}(W_{x^f} x_t + W_{h^f} h_{t-1} + W_{c^f} c_{t-1} + b_f) \tag{2.8}$$
$$c_t = f_t c_{t-1} + i_t \tanh(W_{x^c} x_t + W_{h^c} h_{t-1} + b_c) \tag{2.9}$$
$$o_t = \text{sigmoid}(W_{x^o} x_t + W_{h^o} h_{t-1} + W_{c^o} c_t + b_o) \tag{2.10}$$
$$h_t = o_t \tanh(c_t) \tag{2.11}$$

i, f, o and c are the input gate, forget gate, output gate and cell activation vector respectively. c holds the information content contained in the LSTM cell, and o controls the amount of information flowing out of the LSTM cell. i controls the amount of new information remembered in c whereas f controls the amount of existing information forgotten in c. This can be seen more clearly in the memory cell update formula in Equation 2.9. Each of these vectors is the same size as the hidden vector h. Additionally, each of the weight matrices from the cell to gate vectors is a diagonal matrix (e.g. the W_c matrices for the gates).

By using these various gates to control the flow of information, the inputs can be carouselled in memory by the gates indefinitely [11]. This is achieved by having the input and forget gate saturated near 0, which allows an LSTM to remember important inputs from many time-steps ago.

Despite their rather complicated structure compared with a vanilla RNN, LSTMs have been shown to work remarkably well across a wide range of tasks [12, 10, 13, 14].

2.3.2 Gated Recurrent Units

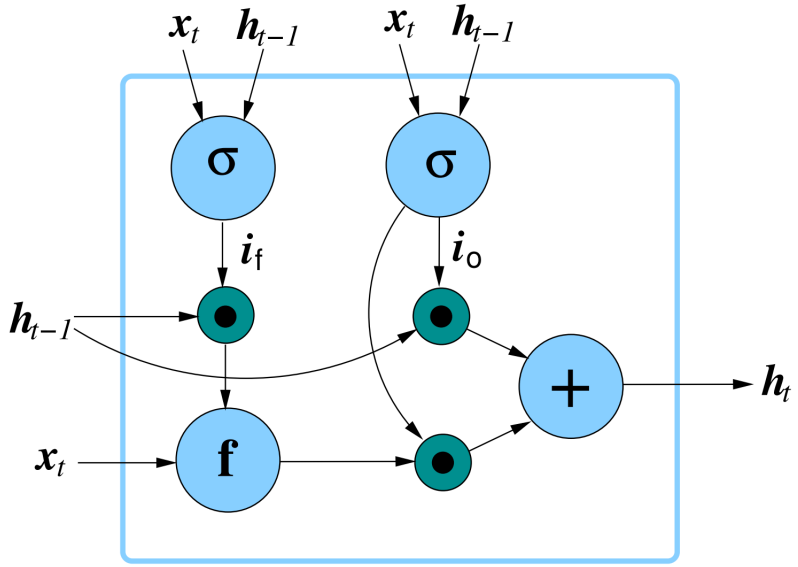


Fig. 2.3 High-level diagram of the GRU architecture. i_f and i_o correspond to the reset gate (a.k.a. forget gate) and update gate (a.k.a. output gate) respectively. **f** is the activation function, which is usually set to the tanh function.

The complex architecture of the LSTM unit motivated the desire for a simpler design that can still achieve the same functions. This led to the introduction of the Gate Recurrent Unit (GRU) by [15]. Similar to the LSTM unit, a GRU learns long-term time-dependencies through the use of gates for modulating the information flow. The equations for describing a GRU is given by:

$$r_t = \text{sigmoid}(W_{x^r} x_t + W_{h^r} h_{t-1} + b_r) \tag{2.12}$$

$$\hat{h}_t = \tanh(W_h x_t + W_h (r_t \odot h_{t-1}) + b_h) \tag{2.13}$$

$$z_t = \text{sigmoid}(W_{x^z} x_t + W_{h^z} h_{t-1} + b_z) \tag{2.14}$$

$$h_t = (1 - z_t) h_{t-1} + z_t \hat{h}_t \tag{2.15}$$

r and z are the reset/forget and update/output gate respectively, and $\odot$ is the element-multiplication operator. r controls the degree to which the previous hidden-state is reset to 0 when computing the candidate activation. z is used to interpolate between the previous activation h_{t-1} and the candidate activation $\hat{h}_t$ when computing the new hidden-state h_t. Consequently, a GRU can "carousel" an input in memory by having the reset gate saturated near 0.

Despite the similarities between a GRU and an LSTM unit, a crucial difference between the two units lies with how the hidden-state h_t is computed. Unlike an LSTM, there is no internal memory cell maintained in a GRU for storing information. This means that the hidden-state is always completely exposed to the rest of the network, whereas an LSTM unit has an output gate that can modulate the hidden-state (equiv. memory) exposure [11].

2.3.3 Comparison of GRUs and LSTMs

Practically speaking, it is hard to determine which unit is better given their similar behaviors. An obvious benefit of using GRUs is the fewer number of gates required, which reduces the number of parameters that need to be trained and stored in computer memory.

[11] performed an empirical evaluation of GRUs and LSTMs and found that gated units always performed better than vanilla RNN units. But their analysis for establishing which of the two gated units were better was inconclusive. On the other hand, [16] performed a comparison of GRUs and LSTMs based on an exhaustive architecture search over many different hyperparameter configurations for multiple sequence-based tasks. They found that GRU outperformed LSTM on all tasks except language modeling. However, an LSTM will perform better when its forget gate bias is initialized to a large value such as 1 or 2.

2.4 Memory Networks

Similar to the original motivation for the LSTM [9], the architecture of memory networks were designed specifically for addressing the problem of learning long-term dependencies in the input sequence [17, 18, 1].

However, these models differ from "traditional" memory-based models like GRU/LSTM in that the addition of memory occurs at the level of the *hidden-state*, rather than within the hidden-unit. These networks also tend to have a more clearly defined interpretation of what is held within the memory cells [1]. The following description of a memory network architecture will focus on the MemN2N model in [1], since this model is the one that most closely aligns with the work presented in this dissertation.

2.4.1 Model Architecture

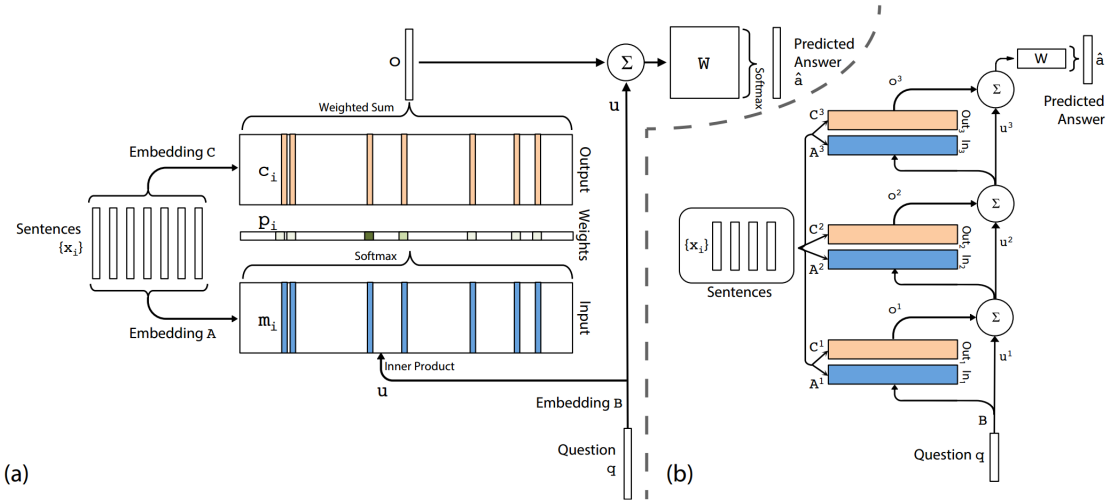


Fig. 2.4 Depiction of the end-to-end memory network. (a) shows a single-layer model, whereas (b) shows a multi-layer model. Diagram taken from [1].

In the simplest case of a single-layer memory network [1], the input sequence $\{x_1, ..., x_T\}$ (corresponding to a set of sentences) is first converted into a set of memory vectors $\{m_1, ..., m_T\}$ by applying an embedding matrix A on each sentence. A separate set of output vectors $\{c_i\}$ is also obtained from $\{x_i\}$ by applying a separate embedding matrix C. An embedding matrix B is then applied to the query sentence q to obtain a query vector u. In the simplest case, each of these embedding matrices can be treated as another linear layer learned by the network.

Matches between the query vector u and the memory vectors m_i are computed in the embedding space by taking the softmax over the inner product:

$$v = [u^T m_1 \; .. \; u^T m_T]^T \tag{2.16}$$

$$p = \text{Softmax}(v) \tag{2.17}$$

which returns a probability vector p over the memory vectors $\{m_i\}$. This can be interpreted as a soft-attention mechanism over the memory cells.

The response vector o is computed by taking the sum over the set of output vectors c_i for each sentence x_i, weighted by the probability p_i:

$$o = \sum_i p_i c_i \tag{2.18}$$

The output probabilities $\hat{y}$ are then generated by adding the response vector o and the embedded query u, and passing it through a weight matrix W and an output softmax layer:

$$\hat{y} = \text{Softmax}(W(o+u)) \tag{2.19}$$

All parameters in the model, including the embedding matrices and the weight matrix W, can be learned using standard training algorithms for feedforward neural nets.

[1] also discussed how MemN2N can be extended with multiple "computational hops" by adding more layers, which is shown in Figure 2.4(b). That discussion is omitted here, but can be found in [1] for the interested reader.

2.5 Network Training

2.5.1 Cross-Entropy Loss Function

In practice, neural networks are almost trained exclusively using the cross-entropy loss function for learning problems with discrete inputs/output pairs:

$$L(\theta) = -\frac{1}{N}\sum_{i=1}^{N} y^{(i)} \log P(y^{(i)}|x^{(i)}) \tag{2.20}$$

$$= -\frac{1}{N}\sum_{i=1}^{N} y^{(i)} \log \hat{y} \tag{2.21}$$

where θ are the network parameters, i is the index of the training pair $(x^{(i)}, y^{(i)})$, $y^{(i)}$ is the target one-hot vector, and $P(y^{(i)}|x^{(i)}) = \hat{y}$ is the predicted class-probability vector. The class-probability $\hat{y}$ is typically computed by applying a softmax on the pre-activation z.

2.5.2 Error Backpropagation

Feed-forward neural networks are typically trained using the back-propagation (BP) algorithm [19] – a gradient-descent procedure that computes the weight-gradients in each layer through recursive applications of the chain rule. The update for the weights in the k^{th} hidden layer is given by:

$$w_{i+1}^{(k)} = w_i^{(k)} - \gamma \frac{\partial L(\theta)}{\partial w_i^{(k)}} \tag{2.22}$$

where γ is the learning rate. A well-known issue with training neural networks – particularly those that are very deep (i.e. containing many hidden layers) – is the problem of vanishing/exploding gradients [8]. At each layer of the network, the error gradients need to be multiplied with the error gradients from the layer below it. For deep networks, the successive application of the chain rule leads to the error gradients growing exponentially large or small as the BP algorithm moves up towards the input layer.

In practice, researchers have developed many techniques for addressing this optimization issue. For example, the method of gradient clipping [20] is one simple but effective technique for training deep networks. Whenever the norm of the gradients exceeds some threshold value, the gradients gets re-scaled. The method of pre-training is another commonly used technique – it initializes the network weights in a favorable state such that the gradient computation is more well-behaved [21, 22, 23]. Alternatively, second-order approaches based on utilizing the Hessian-Free optimizer have also been shown to work well [24].

2.5.3 (Truncated) Backpropagation Through Time

It is possible to train an RNN using the back-propagation algorithm by feeding the input-output pairs (x_t, y_t) in a sequence $\{(x_1, y_1), ..., (x_T, y_T)\}$. However, an RNN trained in such a way is unlikely to make use of its hidden-state effectively. This is because it is trained to predict the target y_t using only the current input x_t.

Instead, RNN models are trained using the back-propagation-through-time (BPTT) algorithm [25]. The BPTT algorithm is a simple extension of the standard back-propagation algorithm that works by "un-rolling" the hidden-state h_t across time, which returns a sequence of hidden-states $\{h_1, ..., h_t\}$. This allows it to backpropagate the error gradients from y_t to all the previously seen inputs $\{x_1, ..., x_t\}$. By training the model in this way, an explicit training signal is provided to the model to remember particular inputs that were useful for predicting y_t.

In effect, the BPTT algorithm can be viewed as training a feedforward network that is deep in time (and has fixed weight matrices). As mentioned earlier, the training of deep feedforward neural networks is problematic due to the issue of vanishing/exploding gradients. Unfortunately, this problem also emerges for training with RNNs, where the gradients vanish or explode when the error gradients are back-propagated across time.

2.6 Regularization

2.6.1 Dropout

Dropout [26] is a commonly used regularization technique for preventing overfitting in neural nets. Dropout is particularly useful when training large networks on a relatively small dataset.

The basic idea behind dropout is simple: at each time-step t, a subset of the hidden-state activation values is "turned off" by setting them to 0. The intuition is that, by switching off the activation of a subset of the neurons during training, the network is encouraged to train using the other neurons that are activated [26].

Mathematically speaking, dropout for feed-forward neural networks is implemented by sampling a dropout mask z_j consisting of 0s and 1s for the j^{th} hidden layer. This mask is then applied over the hidden-state activations h_j to produce a masked hidden-state $\hat{h}_j$ for input to the next layer:

$$\hat{h}_j = h_j \odot z_j \tag{2.23}$$

$$h_{j+1} = W_{j+1}\hat{h}_j + b_{j+1} \tag{2.24}$$

$\odot$ is the element-wise multiplication operator. Typically, the dropout mask z_j will set half of the hidden-state activations to 0, which is equivalent to using a dropout probability of 0.5. Training networks with dropout is performed using BP as usual, but with the addition of the masked hidden-states $\hat{h}_j$.

However, at test time, all hidden activations are left untouched (e.g. z_t is simply a vector of ones) [26]. This essentially has the effect of doubling the magnitude of the inputs to the next layer (assuming that a dropout probability of 0.5 was used). Thus, it is common to multiply the pre-activation for the next layer by the dropout probability to compensate for the doubling effect [26].

2.6.2 Dropout for RNN

The application of dropout for an RNN is similar to the case of feed-forward networks. As mentioned earlier, a recurrent neural network can be viewed as a very deep feed-forward network when unrolled across time. So when the dropout mask is sampled, it should be done in a time-dependent manner. This suggests the following dropout equation:

$$\hat{h}_t = W \begin{pmatrix} x_t \odot z_{x_t} \\ h_t \odot z_{h_t} \end{pmatrix} \tag{2.25}$$

where z_{x_t} and z_{h_t} are the dropout masks sampled at time t for the input x_t and hidden-state h_t respectively. However, empirical results suggests that removing dropout from the recurrent connections yields better performance [27]. The most commonly cited explanation was that the added noise in the hidden-to-hidden connections dropout tend to get amplified across time [27, 28].

This leads to an update in the dropout equations where a dropout mask is only sampled for the input-to-hidden direction [27].

$$\hat{h}_t = W \begin{pmatrix} x_t \odot z_{x_t} \\ h_t \end{pmatrix} \tag{2.26}$$

LSTM models trained with this form of dropout were able to obtain significant performance gains compared with their non-dropout counterparts [27, 29].

Motivated by a Bayesian interpretation of dropout, [28] developed an alternative approach where the sampled masks are kept constant across time:

$$\hat{h}_t = W \begin{pmatrix} x_t \odot z_x \\ h_t \odot z_h \end{pmatrix} \tag{2.27}$$

where the dropout masks z_x and z_h are repeated across all time-steps. In their results, they demonstrated significant performance gains over the approach described in [27]. Further details can be found within their paper [28].

Chapter 3

Statistical Language Modeling

Language models (LM) are commonly used in speech recognition pipelines to incorporate an understanding of language into the system. This is done by assigning probabilities to word sequences.

$$P(\mathbf{w}) = \prod_{t=1}^{T} P(w_t | w_{t-1}, ..., w_1) \tag{3.1}$$

It is also common to express this in the log-domain for improved compatibility.

$$\log P(\mathbf{w}) = \sum_{t=1}^{T} \log P(w_t | w_{t-1}, ..., w_1) \tag{3.2}$$

The goal of language modeling is to train a model that can robustly model this probability distribution. In this chapter, an overview is provided for discrete-space language models (based on computing n-grams) and continuous-space language models (based on training neural nets). The latter class of models can be further divided between ones that operate with truncated word-histories (FNNs) versus complete word-histories (RNNs).

3.1 N-gram Language Models

N-gram models are commonly used in ASR pipelines with great success [4]. In an n-gram model [30], the probability for a word sequence $\mathbf{w} = \{w_1, ..., w_T\}$ is approximated using a

truncated word-history:

$$P(\mathbf{w}) = \prod_{t=1}^{T} P(w_t|w_{t-1},...,w_1) \tag{3.3}$$

$$\approx \prod_{t=1}^{T} P(w_t|w_{t-1},...,w_{t-n}) \tag{3.4}$$

where $n > 0$ is the n-gram order. N-gram models can be trained using maximum likelihood estimation [30], which essentially counts the number of times a particular sequence of words appears in a corpus and divides that number by the total number of word sequences with the same length.

There are two main drawbacks to using n-gram models [31]:

1. data-sparsity issues from low frequency counts for uncommon-words (Zipf's law).
2. poor probability estimates from limited word-history (Markov assumption).

Both problems can lead to unreliable probability estimates. A lower n-gram order helps to alleviate the first problem but at the cost of compounding the second problem. Thus, the setting of the n-gram order has a significant impact on the quality of the n-gram model.

Alternatively, smoothing can be applied to the n-gram probability distribution to improve robustness of probability estimates. A detailed treatment on smoothing techniques is provided in [32]; a discussion on the commonly used Kneser-Ney smoothing technique can be found in [33].

3.2 Feedforward Neural Network Language Models

Similar to n-gram models, a feedforward neural network language model (FNNLM) also approximates $P(\mathbf{w})$ by using a truncated word-history $\{w_{t-1},...,w_{t-n}\}$, where $n > 0$ is length of the history.

$$P(\mathbf{w}) = \prod_{t=1}^{T} P(w_t|w_{t-1},...,w_1) \tag{3.5}$$

$$\approx \prod_{t=1}^{T} P(w_t|w_{t-1},...,w_{t-n}) \tag{3.6}$$

However, a key difference lies in how that word-history is represented. Whereas an n-gram model represents it using discrete n-grams tokens, a FNNLM represents the word-history using a continuous hidden-state vector h [34].

Firstly, a word vector x_i is obtained by projecting with a word-embedding matrix C learned by the network.

$$x_i = Cw_i \tag{3.7}$$

where w_i is the i^{th} word represented using an one-hot encoding. Note that C contains a row-vector corresponding to every word in the vocabulary.

Each word-embedded vector x_i can then be concatenated to form a single, large vector x that is used to form the input to the hidden layer.

$$\hat{x} = (x_{t-1}, ..., x_{t-n+1}) \tag{3.8}$$

The hidden-state h – which approximates the complete word-history – can then be computed by:

$$h = W_x\hat{x} + b_x \tag{3.9}$$

The following equation is then used to compute the pre-activation for the output softmax layer:

$$q = W_h h + b_h \tag{3.10}$$

where q is the pre-activation for the softmax. Finally, the output probability vector is computed by:

$$\hat{y} = \text{Softmax}(q) \tag{3.11}$$

where $\hat{y}$ is the output of the softmax layer.

As noted earlier, the use of vectors within a continuous space to represent words is a marked departure from the discrete-space approach of the n-gram language model [35]. The latter model essentially treats words as one-hot vectors and computes word-probabilities using a look-up table. The benefit of the continuous space approach is that it can result in better probability estimates for words/n-grams that do not frequently appear in the training corpus [35].

Interestingly, the word-embeddings vectors learned by a FNNLM have been shown to possess remarkable linguistic properties [36, 37]. For example, it has been shown that words with similar meanings will tend to have vector-representations that are close to one another

within the continuous word-embedding space [36]. For a discussion on more sophisticated methods of learning word-embedding vectors, see [38].

3.3 Recurrent Neural Network Language Models

A recurrent neural network language model (RNNLM) further develops the notion of using a continuous history representation by using the *complete* word-history to compute $P(\mathbf{w})$ [39].

$$P(\mathbf{w}) = \prod_{t=1}^{T} P(w_t | w_{t-1}, ..., w_1) \tag{3.12}$$

The hidden-state h_t representing the word-history for all words up to time t is computed by:

$$h_t = \text{RNN}(h_{t-1}, x_t) \tag{3.13}$$

where once again, x_t is the word-embedding vector for the word w_t obtained using an embedding matrix C. Computing the output word-probability is the same as in the case of a FNNLM.

$$q_t = W_h h_t + b_h \tag{3.14}$$

$$\hat{y}_t = \text{Softmax}(q_t) \tag{3.15}$$

where $\hat{y}_t$ is the class-probability vector. Empirically speaking, RNNLMs have been successfully applied in many empirical studies [39, 40, 41, 42, 43].

3.3.1 GRU/LSTM-based RNNLM

Gated RNN model forms an extension of the underlying RNNLM by changing how the hidden-state h_t is computed. For example, in the case of an LSTM:

$$h_t = \text{LSTM}(h_{t-1}, x_t) \tag{3.16}$$

Thus, the RNNLM framework allows for the use of a large number of RNN variants in computing h_t, so long as they share the same "interface" exemplified by the above equation. The particular benefit of using a gated unit like a GRU or an LSTM is the potential to encode longer word histories into h_t, as discussed in Section 2.3. In particular, most state-of-the-art language models are based on some variant of a large LSTM trained with dropout

[27, 44, 28]. Performing model combination with multiple large, regularized LSTMs leads to further performance gains [27, 29].

3.4 Memory Network Language Models

Although the MemN2N model [1] was tailored for the task of Q&A, it is straightforward to modify it for language modeling. Similar to the case of FNNLMs, a memory network language model (MEMLM) also uses a truncated word-history to approximate $P(\mathbf{w})$:

$$P(\mathbf{w}) = \prod_{t=1}^{T} P(w_t|w_{t-1},...,w_1) \tag{3.17}$$

$$\approx \prod_{t=1}^{T} P(w_t|w_{t-1},...,w_{t-n}) \tag{3.18}$$

However, the approach used for the internal (continuous) history representation is different. Given that the input sequence is now a sequence of words $\{w_1,...,w_T\}$ instead of a sequence of sentences – each of the memory cells $\{m_i\}$ will now hold the word-embedded inputs $\{x_i\}$. And since there are no longer any questions, the query vector u is simply fixed to some constant vector such as 0.1. But note that the model will still compute a non-uniform attention distribution over the output vectors. All other components of the network are the same as before.

The softmax output layer is used to predict the next word in the sequence using the response vector o and constant-vector u.

$$\hat{y} = \text{Softmax}(W(o+u)) \tag{3.19}$$

where then gives an approximation for $P(\mathbf{w})$. Language modeling experiments showed improvements over an LSTM of a comparable complexity, although the gains were small.

3.5 Optimization Issues for Neural Language Models

As mentioned earlier, n-gram models tend to suffer from problems of data-sparsity and short word-histories (due to the Markov assumption). These issues also impact the training/testing of neural language models, although they manifest in a slightly different way. Common sources of training problems include:

- Large vocabulary sizes. A typical text corpus will contain hundreds of thousands of unique words – and perhaps more, depending on the size of the corpus. It is computationally infeasible to train a neural network with such a large output layer due to the expensive computation required for the normalization constant in the softmax function [45].

- Long sequences. Some text corpora, such as those drawn from news articles or academic documents, will consist of sentences containing many words broken up by punctuation (i.e. this sentence). Training an RNNLM to capture these long, complex inter-word dependencies successfully is difficult due to the vanishing/exploding gradient problem.

A simple but effective approach for addressing large vocabulary sizes is to only use the top K most frequently-occurring words in the training data as the vocabulary and map the rest of the words to an out-of-shortlist node [46]. This node is typically referred to as OOS or <unk>. When restricting the size of the vocabulary is not a feasible option, alternative approaches such as noise contrastive estimation [47, 45, 3]; hierarchical softmax [48, 49]; and importance sampling [50] are commonly used.

3.6 Incorporating Topic Understanding with LDA

A topic-based language model [51] can be succinctly described by:

$$\sum_{z} P(w_t|z)P(z|h_t) \tag{3.20}$$

where z is the latent topic, w_t is the word, and h_t is the word-history. This equation provides a simple method for incorporating a topic model that computes $P(z|h_t)$ into an existing LM, such as an n-gram model. It is common to interpolate this model with another n-gram to obtain the benefits of modeling a global context – while still modeling the short-term word-history that is useful for immediate next word prediction [52].

In the case of a neural network, it is more common to incorporate topic understanding by appending a topic feature z to the word-input vector w_t at time t [2, 3].

$$\hat{w}_t = (w_t, z) \tag{3.21}$$

This topic feature is typically extracted from the corpus using Latent Dirichlet Allocation [53], which takes a sentence using a bag-of-words representation and maps it to a low-dimensional

topic-vector. For an (R)NNLM, the input to LDA is obtained by using the words in the input context window for the model [2]. The topic-vector returned by LDA can then be used by the neural language model to learn a topic-conditioned context vector $\hat{h}_t$ in the hidden layer. This technique is usually effective towards obtaining gains in LM and ASR performance with relatively little effort [2, 3].

More recently, [54] proposed a topic model based on the variational auto-encoder framework [55], which showed LM improvements over [2]. Other related works in this area include [56, 57, 58].

3.7 Language Model Interpolation

It is common to combine multiple language models to obtain more robust probability estimates for a word/sentence. Two commonly used combination techniques for this purpose are linear interpolation and log-linear interpolation [59, 60].

Let $h_t = \{w_{t-1}, ..., w_1\}$ denote the word-history up to time t. The equation for linear interpolation is given by:

$$P(w_t|h_t) = \sum_{k=1}^{K} \lambda_k P_k(w_t|h_t) \tag{3.22}$$

where k refers to the index of the k^{th} language model, and λ_k sums to 1. The equation for log-linear interpolation is similarly given by:

$$P(w_t|h_t) = \frac{1}{Z(h_t)} \prod_{k=1}^{K} P_k(w_t|h_t)^{\lambda_k} \tag{3.23}$$

where $Z(h_t)$ is a normalization term that can be computed by summing $P(w_t|h_t)$ over the vocabulary. A further simplified equation is obtained by taking the log:

$$\log P(w_t|h_t) = -\log Z(h_t) + \sum_{k=1}^{K} \lambda_k \log P_k(w_t|h_t) \tag{3.24}$$

Although the log-linear interpolation above is performed at a word-level, it can also be re-expressed as sentence-level log-linear interpolation.

$$\log P(\mathbf{w}) = \sum_{t=1}^{T} \log P(w_t|h_t) \tag{3.25}$$

$$= -\sum_{t=1}^{T} \log Z(h_t) + \sum_{t=1}^{T} \sum_{k=1}^{K} \lambda_k \log P_k(w_t|h_t) \tag{3.26}$$

$$= C + \sum_{k=1}^{K} \lambda_k \sum_{t=1}^{T} \log P_k(w_t|h_t) \tag{3.27}$$

$$= C + \sum_{k=1}^{K} \lambda_k \log P_k(\mathbf{w}) \tag{3.28}$$

where C is simply a constant.

Chapter 4

Active Memory Networks

4.1 Motivation

As noted in the introduction, the correct interpretation of the word-history is crucial for the accurate prediction of the next word in a sequence. However, given the highly-variable nature of human language, it is difficult to compute a precise interpretation that accurately models the "true" word-history.

In this chapter, the Active Memory Network (AMN) architecture is introduced to provide a potential solution to this problem. The aim of the AMN model is to generate multiple context representations of the word-history through the use of attention and dynamic memory. These context representations allow the model to combine them (using attention) to form a more accurate internal representation of the word-history.

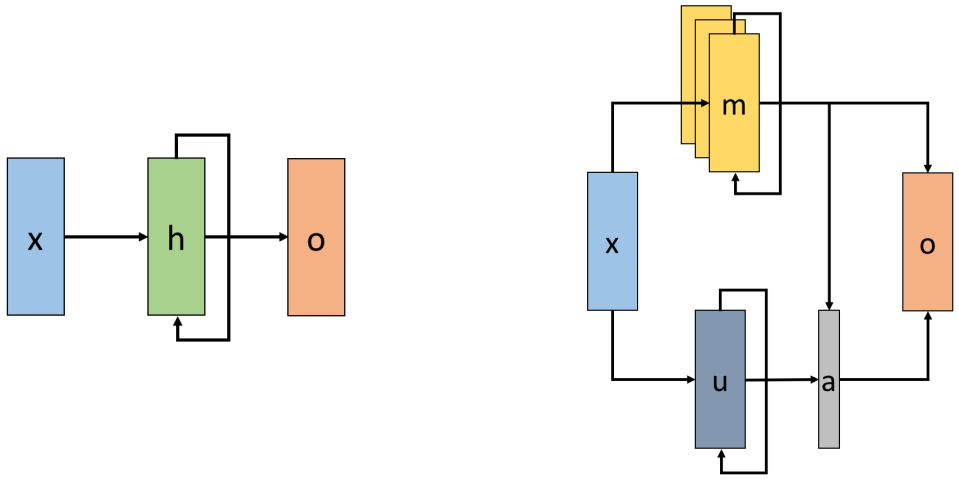
4.2 Model Architecture

At a high-level, an AMN uses a recurrent attention mechanism to attend over K parallel, dynamic memory cells. Each of these K memory cells (abbrv. memcells) operate independent of one another and holds a context vector $c_t^{(i)}$ for the i^{th} memcell. The context vector $c_t^{(i)}$ is meant to contain the information for all the observed inputs up to time-step t. Given that GRUs have been shown to work well for remembering information for long time-durations, it can be used to approximate $c_t^{(i)}$.

$$h_t^{(i)} = \text{GRU}(h_{t-1}^{(i)}, x_t) \tag{4.1}$$

$$c_t^{(i)} = q(x_t, ..., x_1) \tag{4.2}$$

$$= h_t^{(i)} \tag{4.3}$$



(a) RNN (b) AMN

Fig. 4.1 High-level overview of the AMN architecture.

Defining the i^{th} memory cell is then straightforward, since it is essentially the hidden-state of a GRU at time-step t:

$$m_t^{(i)} = c_t^{(i)} \tag{4.4}$$

$$= h_t^{(i)} \tag{4.5}$$

Memory cells are selected by computing a soft-attention vector using a recurrent attention mechanism. This is implemented in the form of a controller that maintains an internal state u_t across time. Once again, this can be implemented with a GRU:

$$u_t = \text{GRU}(u_{t-1}, x_t) \tag{4.6}$$

To generate the attention vector, the inner-product of $m_t^{(i)}$ and u_t is computed and passed through a softmax to obtain the attention weight $\alpha_t^{(i)}$ for the i^{th} memory cell.

$$\beta_t^{(i)} = u_t \cdot m_t^{(i)} \tag{4.7}$$

$$\alpha_t^{(i)} = \frac{\exp(\beta_t^{(i)})}{\sum_j \exp(\beta_t^{(j)})} \tag{4.8}$$

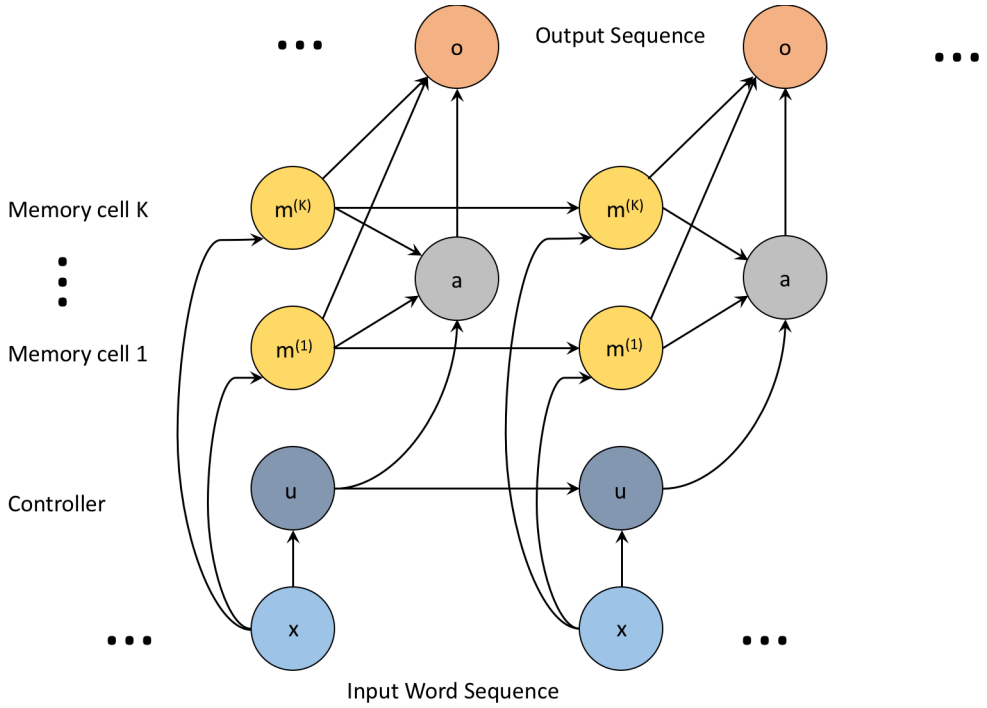


Fig. 4.2 Diagram of the model architecture when "un-rolled" across time. x is the word-embedded input.

Finally, a summation of the memory cell vectors weighted by the attention weights returns the output o_t.

$$o_t = \sum_{i=1}^{K} \alpha_t^{(i)} m_t^{(i)} \tag{4.9}$$

The predicted output at time t can then be computed by:

$$\hat{y}_t = q(Wo_t + b) \tag{4.10}$$

where q is potentially a feed-forward neural network. In the simplest case, q is the softmax function and $\hat{y}_t$ is the prediction for the target y_t.

$$\hat{y}_t = \text{Softmax}(Wo_t + b) \tag{4.11}$$

Once each of these components have been specified, the network can be trained in an end-to-end fashion by minimizing a cross-entropy loss function using backpropagation-through-time.

4.2.1 Adapting an AMN for Language Modelling

For language modeling, the goal is to predict the probability $P(\mathbf{w})$ given the history $h_t = \{w_1, ..., w_{t-1}\}$ of all the words seen up to time t. In the case of an AMN language model, the computation for $P(\mathbf{w})$ is similar to the RNNLM model:

$$P(\mathbf{w}) = \prod_{t=1}^{T} P(w_t | w_{t-1}, ..., w_1) \tag{4.12}$$

The inputs to the model can computed using an embedding matrix C to create word-embedding vectors:

$$x_i = Cw_i \tag{4.13}$$

and $P(w_t|h_t)$ can be computed by:

$$P(w_t|h_t) = \hat{y}_t \tag{4.14}$$

where $\hat{y}_t$ was defined as before.

4.2.2 Specialization in Memcells for Implicit Topics

One of the most interesting aspects of the AMN architecture is the use of dynamic memcells for storing multiple context representations of the input-history.

1. In one scenario, the model learns to adapt the memcells into a single, large ensemble of memcells – which in some ways – can be interpreted as a *fail-safe* mechanism for robust memory preservation.

2. In the other scenario, the model learns to adapt the memcells into specialized memory registers, such that each memcell is responsible for holding highly-contextualized memories.

The two scenarios are not mutually exclusive and it is possible for the model to interpolate between the two. But the second scenario is particularly interesting for language modeling since it opens up the possibility of adapting the memcells for modeling implicit topics. This is closely related to the notion of defining a topic t as a particular *interpretation* of the word-history.

$$t = g(w_t, ..., w_1) \tag{4.15}$$

In the equation above, g is a function that maps word histories to topics. This formulation is very similar to how the context vector contained in the i^{th} memory cell was defined:

$$c^{(i)} = q(w_t, ..., w_1) \tag{4.16}$$

So in theory, an AMN is capable of learning context representations that are topic-dependent, such that:

$$c^{(i)} \approx t \tag{4.17}$$

where $c^{(i)}$ is stored within the memory register $m^{(i)}$ that has been adapted for holding topic-dependent context vectors. Generally speaking, $c^{(i)}$ will be a very rough approximation for t since it also needs to track the syntactical/grammatical context information.

4.3 Training the Attention Mechanism

Given the relatively complex architecture of the AMN model, a simple training procedure based on vanilla BPTT is unlikely to yield the expected model behavior (or performance). In the preliminary experiments, the model often converged towards a set of weights where only a small subset of the memcells was used. This left a lot of unused model capacity in the form of dead memcells.

4.3.1 AMN Gradients for Weight-Updates

To understand why this problem of *over-concentration* occurred, it is useful to examine the derivative of the output o_t with respect to the memcell $m_t^{(i)}$. The complete derivation is provided in Appendix A.1, with the final result given by the equation below. Note that the sub-index t is omitted to improve clarity.

$$\frac{\partial o}{\partial w^{(k)}} = \alpha^{(k)} \frac{\partial m^{(k)}}{\partial w^{(k)}} \Big(1 + \beta^{(k)} - \sum_{i=1}^{K} \alpha^{(i)} \beta^{(i)} \Big) \tag{4.18}$$

$\partial m^{(k)} / \partial w^{(k)}$ is the derivative of the k^{th} memcell vector with respect to the weights used to compute it, and is left un-derived since it is dependent on the RNN variant used to compute $m^{(k)}$.

Equation 4.18 implies that the weight-update for the k^{th} memory cell is directly proportional to the attention weight $\alpha^{(k)}$ assigned to it. In particular, when $\alpha_k = 0$, the k^{th} memcell never gets updated because the error gradient goes to 0. On the other hand, when $\alpha_k = 1$,

only the k^{th} memcell gets trained since it implies that $\alpha_{i \neq k}$ will be 0 (because the attention weights sum to one).

Given that neural nets are usually trained from random weight initialization, it is quite often the case that some memcells will initially get assigned an attention weight near 0. Unfortunately, this has the undesirable effect where only the memcells with non-zero attention weights are tuned during learning. This problem is further compounded by the recurrent nature of how the memcell vectors are computed. If the hidden-to-hidden weight matrix for $m^{(i)}$ is ill-formed, the poor context representation in $m^{(i)}$ at time t will only become worse with each incremental step in time. Consequently, the AMN model is given further encouragement towards activating the few memcells that modeled something sensible.

The remainder of this section provides a discussion of the training techniques developed to address this problem.

4.3.2 Attention-weight Annealing

One simple remedy to address the greedy training behavior of AMN is to force the model to train with all memcells during the first few initial epochs. This can be enforced implicitly by ensuring that the attention weights are evenly distributed (e.g. $\alpha^{(i)} = 1/K$), which can be done via an annealing schedule.

$$\alpha_t^{(i)} = \frac{\exp(\beta_t^{(i)}/T_t)}{\sum_j \exp(\beta_t^{(j)}/T_t)} \tag{4.19}$$

$$T_{t+1} = \gamma \cdot T_t \tag{4.20}$$

where T_t is the temperature at time-step t and γ is the temperature decay value. Note that:

- As T approaches infinity, $\alpha^{(i)}$ approaches $1/K$, which implies that the attention is evenly distributed across all memcells.
- As T approaches 0, one of the weights $\alpha^{(i)}$ approaches 1 with the rest approaching 0, which implies that the model is focused only on a single memory cell.

T is initially set to a high value to encourage weight-tuning in all memcells. The value of T is then slowly lowered with each epoch by multiplying T with the decay-value γ. Note T should always be greater than or equal to 1, since $T = 1$ returns the original softmax activation.

Both T and γ are tunable hyper-parameters for the model. In the preliminary experiments, the model performance was relatively insensitive to the precise setting of T and γ, so long as the attention-weights were uniformly distributed for the first couple epochs. [1]

4.3.3 Dropout

Given the complex architecture of the AMN model, it is fairly easy for the model to overfit the dataset. To address this problem, the use of dropout for model regularization was investigated. A natural question that arises is whether dropout should be applied to the controller hidden-state and/or the memcell hidden-states. An implementation for both is provided below, with a discussion on the implications of each approach.

Firstly, the implementation of dropout in AMN is based on the approach described in [27] (see 2.6.1). Namely, only a random subset of the hidden units in the forward connections are dropped, and the recurrent connections are left untouched. For the controller, this is implemented using:

$$\hat{u}_t = W \begin{pmatrix} x_t \odot z_{u_t} \\ u_t \end{pmatrix} \tag{4.21}$$

whereas for the memcells, this is implemented as:

$$\hat{m}_t^{(i)} = W \begin{pmatrix} x_t \odot z_{m_t}^{(i)} \\ m_t^{(i)} \end{pmatrix} \tag{4.22}$$

Note that a new dropout mask is sampled at each time-step t for both u_t and $m_t^{(i)}$ respectively.

Applying dropout to either the controller and/or the memcells will inject noise into the computation for the attention weights, which indirectly affect the computation of the combined output o_t. However, it is important to observe that the masked-controller $\hat{u}_t$ is only utilized in the computation of the attention weights, and not in the computation of the memcell vectors.

$$\hat{\beta}_t^{(i)} = \hat{u}_t \cdot m_t^{(i)} \tag{4.23}$$

$$\hat{\alpha}_t^{(i)} = \text{Softmax}(\hat{\beta}_t^{(i)}) \tag{4.24}$$

$$\hat{o}_t = \sum_i \hat{\alpha}_t^{(i)} m_t^{(i)} \tag{4.25}$$

[1] A reasonable setting would be $T = 250$ and $\gamma = 0.15$, which was used in all the experiments.

So it will not address the problem of overfitting in memcells, nor will it have an effect on the final output o_t besides changing which memcell the model focuses on.

On the other hand, when dropout is applied to the memcells, $m_t^{(i)}$ is directly modified.

$$\hat{\beta}_t^{(i)} = u_t \cdot \hat{m}_t^{(i)} \tag{4.26}$$

$$\hat{\alpha}_t^{(i)} = \text{Softmax}(\hat{\beta}_t^{(i)}) \tag{4.27}$$

$$\hat{o}_t = \sum_i \hat{\alpha}_t^{(i)} \hat{m}_t^{(i)} \tag{4.28}$$

So unlike the case of dropout applied to the controller, the final output $\hat{o}_t$ has noise injected into both the attention weights and the memcell vectors. In this case, overfitting in the memcells will likely be reduced due to direct regularization from dropout.

In summary, when dropout is applied to the memcell, both the attention mechanism and the memcells are regularized. Dropping the controller, on the other hand, is less important since it will only affect how memcells are interpolated to produce o_t.

4.4 Implicit-Target Loss Regularization

4.4.1 Motivation

In some ways, the AMN model can be viewed as a pseudo mixture-of-experts (MoE) [61, 62], with the controller acting as the gating function and the memcells acting as the experts. However, an AMN differs from a typical MoE model in that the experts do not directly output a vector of class probabilities. Instead, each memcell learns a context representation which it contributes towards an interpolated representation for prediction.

In the current formulation of AMN, the error function is indirectly computed based on a cross-entropy loss between the target class-labels and an interpolated memcell output o_t. This error measure compares the target vector against a "smeared" output consisting of multiple memcell vectors. Consequently, to minimize the error, each memcell must contain a vector that is not only useful for predicting the class-labels but will also correct for the residual errors in the other memcell vectors.

Ideally, the interaction between memcells can be de-coupled so that a weight-update in one memcell does not significantly impact subsequent weight-updates in the other memcells. In [61], this was done by using a loss function which directly minimized the difference between the target output y and the expert output m. A weight-update in one expert will still affect the weight-updates for the other experts, but only due to changes in the mixing

proportions from having an improved expert. Unfortunately, there is no clear way to do this with an AMN, given that the experts do not directly output a vector of class probabilities.

4.4.2 Defining the Implicit-Target Loss (ITL)

To provide an improved training objective for AMN, a regularization term based on minimizing the loss between the i^{th} memcell output $m^{(i)}$ and an *implicit* target output I_t (different from the *explicit* training-target y_t) is introduced:

$$\tilde{L}(\theta) = L(\theta) + R(\theta) \tag{4.29}$$

$$= L(\theta) + \lambda \sum_{i=1}^{K} \alpha_t^{(i)} \| I_t - m_t^{(i)} \|^2 \tag{4.30}$$

$\tilde{L}$ and E are the augmented and original error function respectively; λ is a tunable weight on the regularization term; and $\alpha_t^{(i)}$ is the attention assigned to the i^{th} memcell at time t. This regularization term is referred to as the implicit-target loss (abbrv. ITL).

Intuitively, I_t exists as some desired output useful for predicting the class-labels which the memcells should – ideally – be trained to directly model. Moreover, I_t is a dynamic regularization that varies across time. In some sense, this can be visually interpreted as the model trying to hit a moving dartboard. Notice that in the regularization term, the quantity of the error contributed by each memcell is proportional to the attention value $\alpha^{(i)}$ assigned to that memcell.

An important question remains, and that is how should I_t be chosen? One possibility is to use a teacher network [63] to provide the additional training signal to the memcells. However, for most practical situations, such a network does not exist. Instead, notice that an estimate for I_t already exists within the network, and it is given by the interpolated memcell output o_t. Thus, to avoid additional training supervision, I_t can be set to the interpolated memcell output:

$$o_t = \sum_{j=1}^{K} \alpha_t^{(j)} m_t^{(j)} \tag{4.31}$$

$$\tilde{L}(\theta) = L(\theta) + \lambda \sum_{i=1}^{K} \alpha_t^{(i)} \| o_t - m_t^{(i)} \|^2 \tag{4.32}$$

Under this formulation, $R(\theta) = 0$ when either (i) a single memcell is activated or (ii) the memcells are identical and are all activated equally.

(ii) is interesting since, at first glance, it suggests that this regularization term may encourage the model to train the memcells to be identical. However, it is important to observe that if noise was injected into the computation of the memcells, it is highly unlikely that any of the memcells will ever be identical. In particular, the dropout technique for AMN discussed earlier will achieve precisely this effect. Consequently, $R(\theta)$ will typically be non-zero for most training cases when used in conjunction with memcell-dropout. This will be further examined in the experiments.

4.4.3 Interpreting ITL as L2 Regularization

Alternatively, one can also interpret ITL regularization as a form of L2 regularization on the memcell vectors. This can be shown by setting $I_t = \vec{0}$ and assuming that $\alpha_t^{(i)} = 1/K$.

$$R(\theta) = \lambda \sum_{i=1}^{K} \alpha_t^{(i)} \|I_t - m_t^{(i)}\|^2 \tag{4.33}$$

$$= \lambda \sum_{i=1}^{K} \frac{1}{K} \|\vec{0} - m_t^{(i)}\|^2 \tag{4.34}$$

$$= \tilde{\lambda} \sum_{i=1}^{K} \|m_t^{(i)}\|^2 \tag{4.35}$$

where $\tilde{\lambda}$ is a scaled regularization penalty. Practically speaking, this implies that when the attention weights are evenly distributed and $I_t = o_t$ is close to $\vec{0}$, the regularization effects of ITL will essentially be equivalent to L2. This has implications for setting the weight-initialization scale, since smaller weights will likely push o_t towards $\vec{0}$, which in turn pushes ITL towards L2 regularization.

4.4.4 ITL Gradients for Weight-Updates

The derivative of the ITL regularization term with respect to weights of a memcell is provided below. The complete derivation for the results in this section is in Appendix A.2 for the interested reader.

$$\frac{\partial R(\theta)}{\partial w^{(k)}} = \lambda \sum_{i=1}^{K} \alpha^{(i)} \frac{\partial m^{(k)}}{\partial w^{(k)}} \Big[(\delta_{i,k} - \alpha^{(k)}) u \|o - m^{(i)}\|^2 \tag{4.36}$$

$$+ 2\Big(\alpha^{(k)} + \sum_{z=1}^{K} \alpha^{(z)} (\delta_{z,k} - \alpha^{(k)}) \beta^{(z)}\Big)\Big] - \lambda 2 \alpha^{(k)} \frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{4.37}$$

When $\alpha^{(k)} = 0$ or $\alpha^{(k)} = 1$, $\partial R(\theta)/\partial w^{(k)} = 0$. So, similar to the original training objective, a memory cell will only receive a weight-update if it has a non-zero activation.

Now consider the case where the attention weights are evenly distributed, which reduces Equation 4.37 to:

$$\frac{\partial R(\theta)}{\partial w^{(k)}} = \frac{\lambda}{K} u \frac{\partial m^{(k)}}{\partial w^{(k)}} \sum_{i=1}^{K} \left[(\delta_{i,k} - \frac{1}{K}) \| o - m^{(i)} \|^2 \right] \tag{4.38}$$

This leads to the following two cases:

- Case 1: $m^{(i)} = m^{(j)}$. When the attention is evenly distributed because the memcells are identical, $o = m^{(i)}$. Consequently, $\partial R(\theta)/\partial w^{(k)} = 0$, as expected.
- Case 2: $m^{(i)} \neq m^{(j)}$. This case is more interesting since it shows that the weight-update for a memcell will be weighted by both its distance to the mean memcell *and* the distance of the other memcells to the mean memcell. If the other memcells are far from the mean memcell, the weight-update will tend to be smaller. On the other hand, if the other memcells are already close to the mean memcell, the weight-update will tend to be larger.

The implications of Case 2 are interesting, particularly in the context of memcell-dropout. When a subset of the memcells dominate the interpolated output o_t, the other memcells are encouraged to take bigger weight-update steps. So if ITL was applied in conjunction with dropout, this implies that the other memcells will be making larger weight-update steps using noisy gradients, which essentially leads to exploratory behavior in weight-space. This in turn is more likely to result in specialized memcells.

4.5 Relationship to Other Approaches

4.5.1 Vanilla RNN

The architecture of an AMN can be interpreted as being similar to a standard RNN when the context vectors in the memcells are approximated using an RNN, but with the exception that an AMN contains multiple hidden-states. Each of these hidden-states captures a particular embedding of the input sequence containing the context information from the word-history. In the case where an AMN contains only a single memcell, its behavior will be almost identical to an RNN with a similar hidden layer size.

However, in the case where multiple memcells are used, the fundamental difference between the two models lies in how the final hidden-state is generated for the next layer in

the network. In an RNN, the internal state is directly passed to the next layer.

$$o_t = h_t \tag{4.39}$$

Whereas in an AMN, a recurrent attention mechanism is used to generate an attention vector for selecting which hidden-state to pass to the next layer:

$$o_t = \sum_i \alpha_t^{(i)} h_t^{(i)} \tag{4.40}$$

Consequently, an AMN can potentially keep a hidden-state inactive in memory for many time-steps, and only activate it again when the recurrent controller decides to access that memory cell.

4.5.2 Memory Networks

The AMN model shares many similarities to a MemN2N network with a single computation hop [1]. Both models use a shared embedding matrix to project a word w_t to a word-embedding x_t, and both models attend over a set of memcells to compute an interpolated vector o_t for the final softmax layer. However, the key difference lies in what is stored within the memcells.

1. In MemN2N, the memcells consists of a set of static word-embeddings from the input sequence.

$$m^{(t)} = x_t \tag{4.41}$$

   This implies that MemN2N will be able to access a longer time-span of history information in memory, but at the expense of a degradation in the quality of those representations.

2. In AMN, the memcells consists of dynamic context representations of the word-histories, which are computed using the hidden-state of a RNN/GRU/LSTM.

$$h_t = \text{RNN}(h_{t-1}, x_t) \tag{4.42}$$
$$m^{(i)} = h_t \tag{4.43}$$

   Thus, the memory in an AMN will likely be more unstable over time due to the use of dynamic hidden-states, but it will also contain much richer context representations for the word-history.

In some sense, the MemN2N model can be viewed as storing a *static, long-term* history representation in memory that can be accessed at any point in time, whereas an AMN utilizes a *dynamic, working* memory that maximizes the utility of the information stored in memory – at the expense of limiting its lifespan.

4.5.3 Other Works

[64] described an RNN-variant where the K previous hidden-states were used for computing the current hidden-state h_t.

$$h_t = f(W_x x_t + \sum_{k=1}^{K} W_{h_k} h_{t-k}) \tag{4.44}$$

where f is an activation function, W_x is an input-to-hidden weight matrix, and W_{h_k} is a hidden-to-hidden matrix for the k^{th} hidden-state. The motivation is that the introduction of these *shortcut* paths from the hidden-states $\{h_{t-k}, ..., h_{t-1}\}$ to h_t will reduce the number of error-backpropagation steps required for important inputs seen early in the sequence. This idea is similar to the attention-based RNN model described in [65], except [64] uses learnable weight matrices instead of attention weights for computing h_t. This model has some relation to AMN given the use of multiple hidden-states. But crucially, all context information needs to be encoded within a truncated sequence of highly-correlated hidden-states – instead of being distributed independently across multiple hidden-state trajectories, as is the case in AMN.

Chapter 5

Implementation Details

This chapter presents the implementation details for the experiments, with a focus on the implementation for training RNNLMs and applying them for speech recognition.

5.1 Corpus-Level Training with RNNLM

RNNLMs were trained at a corpus-level for all experiments, which was performed by treating the corpus as a single, giant sentence. So the hidden-state of the RNNLM was never reset – instead, it was used to initialize the hidden-state for the next batch. The reasons for this choice is as follows:

1. *More context:* The sentences in the corpora used in the experiments were in sorted order. So in most cases, the current sentence directly follows from the previous sentence. Training the RNNLMs at a corpus level allowed the models to learn these sentence-to-sentence dependencies, which was particularly relevant for analyzing the learning behaviour of AMN.

2. *Faster processing:* Maintaining the same hidden-state across batches removes the need for padding variable-length word-sequences. Therefore, each mini-batch can fit more words during training.

Tensorflow [66] was used for all code implementation, with models trained using multiple GPUs with synchronous gradient weight-updates [66]. The implementation for performing synchronous gradient weight-updates was based on partitioning the training corpus into K streams, and training a model replica on each stream of data.

5.2 Implementation for Speech Recognition

5.2.1 N-best Rescoring

```
0 here yeah <eos>
0 here we go <eos>
0 here again <eos>
...
1 welcome everybody <eos>
1 well come everybody <eos>
1 well i can everybody <eos>
...
2 um i'm abigail class fun <eos>
2 um i'm abigail clapham <eos>
2 um i'm abigail caplan <eos>
...
```

Fig. 5.1 An example of a N-bestlist.

In the N-best rescoring task [67], the goal is re-rank a set of sentence hypotheses generated by a speech recognition system such that the correct transcription is ranked first. In most standard applications, the N-best sentences are generated by performing Viterbi decoding with an acoustic model and an n-gram language model [68, 4]. These N-best sentences can then be rescored using some other language model, such as an RNNLM. The sentence with the highest score is then chosen from each set of N-best sentences; this is referred to as the 1-best sentence.

5.2.2 Log-linear Interpolation

For RNNLM rescoring, a new score is assigned to each sentence in the N-best list by applying log-linear interpolation using the n-gram and RNNLM scores (see Section 3.7).

Let $\mathbf{o}$ be the acoustic observation, $\mathbf{w} = \{w_1, ..., w_T\}$ be the input word sequence, and $s_a(\mathbf{w}, \mathbf{o})$ be the acoustic model score. $s_a(\mathbf{w}, \mathbf{o})$ is the likelihood that the sentence $\mathbf{w}$ corresponds to the acoustic observation $\mathbf{o}$. The new sentence score $\hat{s}(\mathbf{w})$ is given by:

$$\hat{s}(\mathbf{w}) = s_a(\mathbf{w}, \mathbf{o}) + \delta \cdot s_{lm}(\mathbf{w}) \tag{5.1}$$

where $s_{lm}(\mathbf{w})$ is the language model score and δ is the grammar-scale [1]. The interpolated score s_{lm} is obtained by log-linear interpolation:

$$s_{lm}(\mathbf{w}) = \lambda \sum_{t=1}^{T} \log P_{ngram}(w_t|w_{t-1},...,w_{t-n}) \tag{5.2}$$

$$+ (1-\lambda) \sum_{t=1}^{T} \log P_{rnn}(w_t|w_{t-1},...,w_1) \tag{5.3}$$

where $n > 0$ is the n-gram order and λ is the interpolation weight for the n-gram language model and RNNLM. Note that $\mathbf{w}$ is always terminated with an end-of-sentence symbol (e.g. $w_T =$ <eos>), the probability of which is also included in the language model score.

5.3 Corpus-level N-best Rescoring with RNNLM

In the simplest case where rescoring is performed with an RNNLM trained at a sentence-level (e.g. by feeding it batches of zero-padded sentences), obtaining $P_{rnn}(w_t|w_t,...)$ is straightforward. For each N-best sentence, the RNNLM starts at the same initial state and generates a probability for each word. These probabilities can then be summed in the log-domain to obtain a score for log-linear interpolation.

In the case of an RNNLM trained at a corpus-level, N-best rescoring is complicated by the need to condition on a very long word-history. Since the number of possible word-histories grows exponentially with the number of sentences in the reference transcription, selecting the most-likely word-history is computationally infeasible.

5.3.1 1-Best Word-History Conditioning

One simple solution is to condition the RNNLM on the word-history given by the 1-best sentences. Practically speaking, this is implemented by setting h to be the hidden-state after consuming the <eos> token for the current 1-best sentence. However, there are multiple options for deciding how to choose the 1-best sentence for conditioning:

- Option 1: using the acoustic model and n-gram model score. The caveat here is that it may potentially push the RNNLM towards assigning higher scores to sentences favored by the n-gram model. This can cause a loss in model diversity in the LM interpolation step.

[1] also know as the language model scale

- Option 2: using the acoustic model and RNNLM score. Using this option can potentially increase model diversity during LM interpolation. On the other hand, the 1-best sentences selected will likely be worse than the 1-best sentences chosen using Option 1. This may increase the noise in the word-history, and consequently, reduce the robustness of the probability predictions made by the RNNLM at future time-steps.
- Option 3: using the acoustic model, RNNLM score, and n-gram score. The implications of using this approach is essentially a combination of Option 1 and Option 2.

See [69] for a further discussion on choosing word-histories for RNNLM conditioning.

5.3.2 Code Implementation

In the preliminary experiments, the different methods discussed for conditioning the RNNLM made little difference in terms of the final performance. So for simplicity of implementation, the RNNLM was conditioned using the 1-best sentences from the acoustic model + n-gram scores.

Variable-length N-best sentences were zero-padded to obtain a fixed-length vector, where the length was equal to the length of the longest sentence. The N-best sentences were then split into sequential batches of inputs, where each batch contained the N-best sentences for J reference sentences. This improves computational efficiency since the length of each batch will be limited to the longest sentence in that batch, which significantly reduces the number of zero-elements.

Since the RNNLM starts in the same state for each set of N-best sentences, the N-best rescoring implementation was computed on a GPU to obtain significant speed-gains.

Chapter 6

Language Modelling Experiments

This chapter presents the experimental results and analysis from evaluating AMN for language modeling. The AMN model was compared against a set of competitive baseline models across a number of datasets. The results of particular interest were:

1. The performance of the AMN model for language modeling.
2. The attention behaviors learned by AMN after training, particularly concerning the training techniques discussed in Chapter 4.

This chapter is organized as follows. Section 6.1 discusses the datasets used for the experiments; Section 6.2 provides an overview of the metrics used for analysis; the remainder of this chapter is divided according to the results from each dataset.

6.1 Datasets

The following datasets were used for training and analysis:

- The **Penn TreeBank (PTB)** corpus is commonly used for language modeling research and consists mainly of text related to finance, politics and business. Many baseline results from past research studies already exist for this dataset, which makes it an excellent debugging set for troubleshooting potential training issues (or bugs in the code implementation). PTB experiments were conducted using the same pre-processed dataset as [27].
- The **BBC Multi-Genre Broadcast News (MGB)** corpus consists of both manually-transcribed and automatically-generated subtitles drawn from seven weeks of BBC

broadcasts. This corpus contains highly-structured data with many genres/topics dependencies to be learned by a language model, which makes it a very suitable dataset for evaluating the implicit topic learning ability of AMN. Note that most of the sentences use conversational-style English.

Dataset	Num. Words	Num. Sentences	Vocab Size	Avg. Sentence Length
PTB	0.8m	40k	10k	20
MGB	12.4m	1.5m	35k	8

Table 6.1 Training corpus statistics for the LM datasets.

6.2 Metrics for Analysis

6.2.1 Perplexity (PPL)

Language models are commonly evaluated using the perplexity metric. The perplexity of a word sequence $\mathbf{w} = \{w_1, ..., w_{t-1}\}$ is defined as:

$$\text{Perplexity} = \exp\Big(-\frac{1}{T}\sum_{t=1}^{T}\log P(w_t|w_{t-1}, ..., w_1)\Big) \tag{6.1}$$

where T is the length of the word sequence $\mathbf{w}$. The perplexity of a model is typically obtained by computing the above equation over the entire corpus (e.g. by treating the corpus as a single, giant sentence).

Intuitively, the perplexity measures how good a model is at generating a sequence from the text corpus – a low-perplexity language model will be one that assigns high probability values to sentences in the corpus. So lower perplexities are preferable to higher perplexities.

A significant disadvantage of using the perplexity metric is that it is measured based on a perfect knowledge of all previously seen words. This has important practical implications at test time since the history of the words seen by the model is typically noisy in real-world applications. This applies in particular to speech recognition systems, where the words generated at each time-step comes from a probabilistic system. Consequently, a language model may achieve excellent perplexity results and yet still perform poorly for speech recognition.

6.2.2 Measuring the Attention Behaviour using Entropy

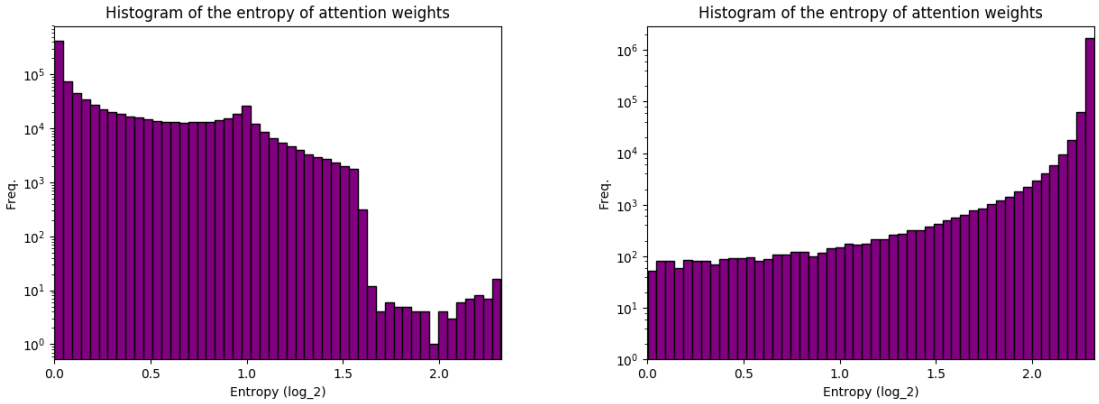
To quantify concepts like "over-concentration" in the AMN model, it is necessary to define a metric that can measure the spread in the attention-weights across inputs. One simple way to do this is to use the entropy measure.

$$E(X) = -\sum_{i=1}^{K} P(x_i)\log_2(P(x_i)) \tag{6.2}$$

where $0 \leq E(X) \leq \log_2(1/K)$ and X is a random variable.

Note that the attention-weights $\alpha^{(i)}$ computed by an AMN for the i^{th} memcell can be interpreted as the probability that a memory cell is activated, since $0 < \alpha^{(i)} < 1$ and $\sum_{i=1}^{K} \alpha^{(i)} = 1$. Thus the entropy metric can be used to measure the spread of the attention weights across all inputs – this will be referred to as the *attention-entropy* of the model:

$$-\sum_{i=1}^{K} \alpha^{(i)}\log_2(\alpha^{(i)}) \tag{6.3}$$



(a) Concentrated attention.

(b) Evenly divided attention.

Fig. 6.1 Examples of attention-entropy plots for an AMN with 5 memcells.

A model which focuses its attention completely on a single memcell will have a low attention-entropy, whereas a model which spreads its attention evenly across all memcells will have a high attention-entropy. An example is shown in Figure 6.1. This metric will be relevant when comparing the different training techniques in terms of their impact on the attention behavior.

6.2.3 Measuring Memory Specialization with Cosine-Similarity

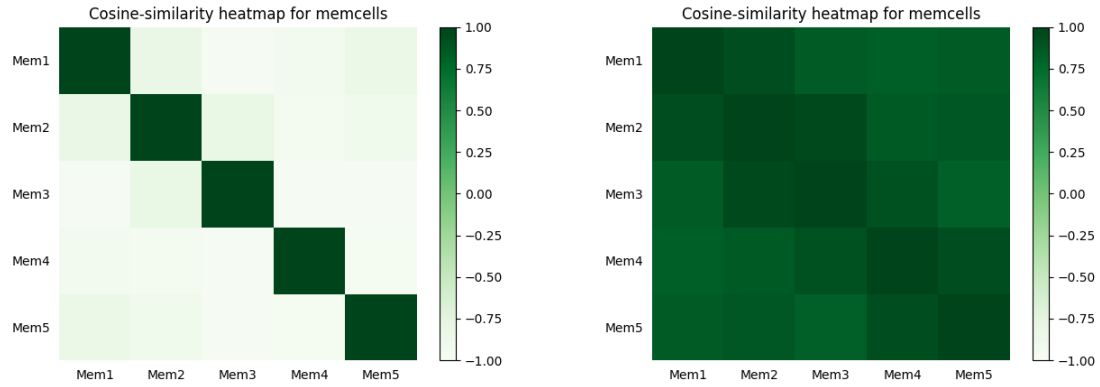
The pairwise cosine similarity between each memory cell can be computed to measure how the model learned to adapt its memory:

$$s_{ij} = \frac{m^{(i)} \cdot m^{(j)}}{\|m^{(i)}\|_2 \|m^{(j)}\|_2} \tag{6.4}$$

where $-1 \leq s_{ij} \leq 1$. This can then be used to compute a K by K matrix containing the pairwise cosine similarity for each pair of memcells.

$$S = \begin{bmatrix} 1 & s_{12} & \dots & s_{1K} \\ s_{21} & 1 & \dots & s_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ s_{K1} & s_{K2} & \dots & 1 \end{bmatrix} \tag{6.5}$$

Note that the diagonal elements are equal to 1 since the cosine similarity of a vector with respect to itself is always equal to 1. Moreover, S is by definition a symmetric matrix.



(a) High memcell specialization.

(b) Low memcell specialization.

Fig. 6.2 Examples of cosine-similarity heat-maps for an AMN with 5 memcells.

The mean cosine similarity matrix across all inputs can be used to produce a heat-map that visualizes the co-similarity between each pair of memcells. This can indicate whether certain memcells have learned to specialize for particular context representations. An example is shown in Figure 6.2.

6.2.4 Perplexity of Individual Memory Cells

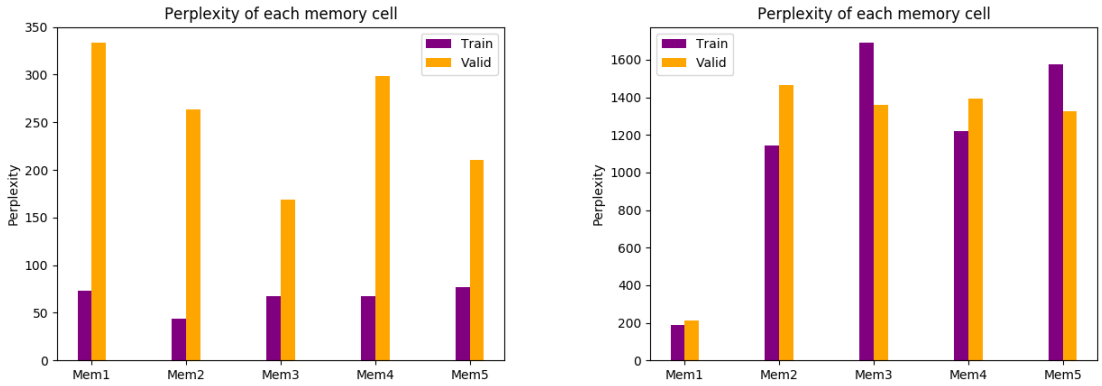
To test the robustness of the context representation learned in the k^{th} memcell, the perplexity can be re-computed with $\alpha^{(k)} = 1$. This forces the model to only use the context vector in the k^{th} memcell for all predictions and isolates any potential co-dependencies with other memcells.

Evaluating the memcells in this way is technically incorrect, since the model could be using the attention mechanism in complex ways to access and combine its memory when predicting the next word. However, this method does give a rough indication of:

1. The presence of dead memcells, which can be inferred by observations of exceptionally high training/test perplexity and zero activation.

2. Whether the memcells have potentially underfitted/overfitted the training set.

An example where the plots indicate the presence of overfitted memcells and dead memcells is shown in Figure 6.3.



(a) Overfitted memcells.

(b) Dead memcells.

Fig. 6.3 Examples of perplexity from individual memcells for an AMN with 5 memory cells.

6.3 Penn TreeBank Corpus (PTB)

PTB was primarily used as a "debugging" corpus for experimenting with different ways of training an AMN. In particular, the effectiveness of training using attention-weight annealing, dropout, and implicit-target loss regularization were examined. A large AMN model was then trained to examine the impact of increasing the number of model parameters.

Most of the analysis for the training techniques discussed in Chapter 4 are performed in this section, but the reader may skip directly to the Appendix B.1 for the complete table of results. This table includes some results from comparable work on contextual RNNLMs.

6.3.1 Experimental Setup

PTB The PTB training set was taken from [27] using the same sentence processing procedure. The top 10,000 most frequent words were used for both the input and output vocabulary. All other words were mapped to the special unknown token <unk>.

Model Parameters The number of model parameters were kept roughly constant across all models to perform a fair comparison.

- The baseline RNN, GRU, and LSTM had a hidden size of 125 recurrent units with roughly 2.6 million parameters. The AMN contained 5 memcells, with all recurrent components implemented using a GRU with a hidden layer of 100 hidden units – this model had roughly 2.3 million parameters.
- For the large models, the AMN model used a recurrent layer of 500 units. The size of the baselines was increased to 750 hidden units. The total number of model parameters was approximately 16–20 million across all the models.

Training Unless stated otherwise, all PTB models were trained using a batch size of 30 [1]. Gradient clipping was performed with a max norm of 10 [20]. The adaptive training optimizer Adam [70] was used to avoid the need for manually tuning the learning rate across training epochs. The BPTT step size was set to 20. All models were trained using early-stopping for regularization. A dropout probability of $\{0.35, 0.5\}$ was used in all experiments when applicable, with results from the best performing model reported.

Experiments A description of the AMN models trained for the PTB experiments is provided here.

- **AMN**: An AMN trained using a vanilla training strategy. This experiment was mainly conducted to illustrate the optimization issues related to AMN and to set up a baseline for comparing different methods of training an AMN.

[1] Although a batch size of 10 yielded better performance across all models.

- **AMN + Anneal**: An AMN trained with attention-weight annealing. The initial temperature was set to 250 and the temperature decay was set to 0.15. These settings were sufficient to force the model to output a set of evenly-distributed attention weights during the first few epochs.
- **AMN + {Drop-Contr, Drop-Mem, Drop-All}**: An AMN with dropout applied to (i) the controller, (ii) the memory cells, or (iii) both.
- **AMN + Implicit**: An AMN trained with implicit-target loss regularization. $\lambda = \{0.5, 2.0\}$ was used for the weight on the ITL regularization term.
- **Large-AMN**: A large AMN model with 500 recurrent units.

All analysis of the large AMN model will be restricted to the Large-AMN + Drop-Mem model since the performance and behavior of this model was sufficient to demonstrate the main results.

6.3.2 Training the Attention Mechanism

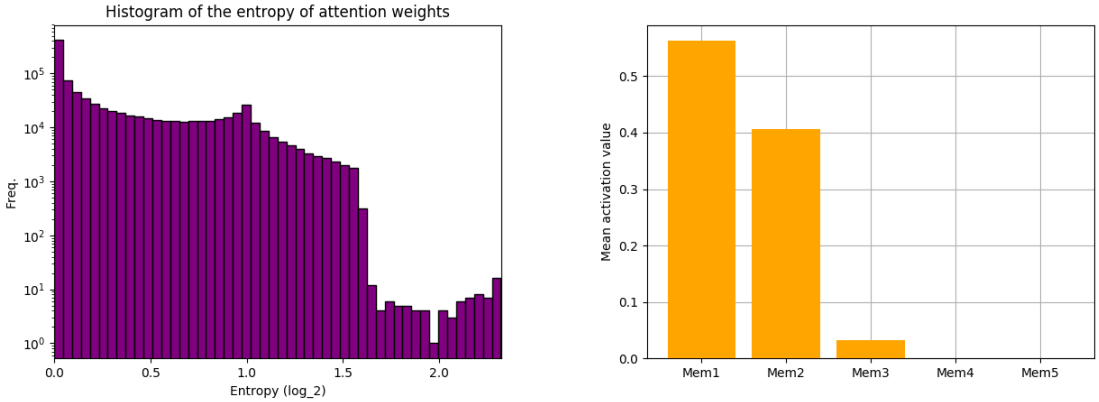
Model	Train	Valid
GRU	76	135
GRU + Dropout	88	116
AMN	82	144
AMN + Anneal	90	148
AMN + Drop-Contr	73	123
AMN + Drop-Mem	73	**108**
AMN + Drop-All	77	**110**

Table 6.2 Perplexity for baselines and AMN on PTB. Lower is better.

The perplexity results from training with each technique are shown in Table 6.2. Training AMN with dropout tend to take longer to reach convergence due to the increased noise in the error gradients, which is consistent with known results [26]. The use of dropout also led to worse performance on the training set but improved performance on validation, which is in line with the results reported in [27]. The best-performing models utilized dropout in the memcells (AMN + Drop-Mem and AMN + Drop-All).

The rest of this section will examine how the learned attention mechanism changes for each of the training techniques examined. Plots of the learning curves are omitted here but can be found in Appendix B.2.

Training with a vanilla configuration



(a) Attention-entropy.

(b) Mean memcell activation.

Fig. 6.4 Attention behaviour for vanilla AMN on the training set at convergence.

When a vanilla configuration was used to train AMN, the model suffered from *over-concentration* on a particular memory cell, as expected. This is shown in Figure 6.4b by the high activation on the first and second memory cell on the training set. As noted in Section 4.3, a memory cell $m^{(i)}$ will never receive a weight-update if the AMN model never activates that memory cell. This is because $\partial o / \partial w^{(k)} = 0$ for $\alpha^{(k)} = 0$.

Instead, the error gradients will be attributed to the memcells that were active at that time-step. Consequently, as those memcells improve in their predictive performance, the controller is also more likely to activate those memcells again at the next time-step. This leads to a positive-feedback loop where only the active memcells gets trained and the rest of the memcells converge to a dead-state.

Figure 6.5 shows that the third, fourth and fifth memcells have essentially learned nothing from the data, which suggests that the learning algorithm never tuned these memcells. Indeed, only the first memory cell was capable of achieving a reasonable perplexity value.

Training AMN with attention-weight annealing

The attention weights were more evenly distributed compared to the vanilla setup, with every memory cell being utilized by the model. This can be seen in Figure 6.6. Furthermore, the attention-entropy plot in Figure 6.7 indicates that the model was altering its level of concentration from input to input, which is shown by the flat distribution across all attention-entropy values. This is generally indicative of a well-behaved attention mechanism. A similar

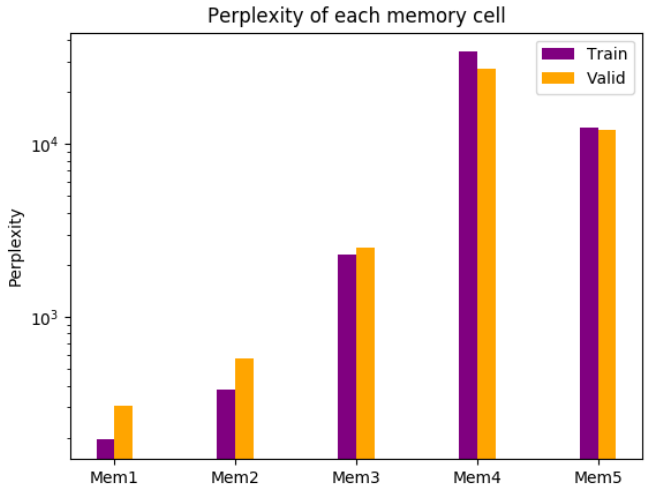


Fig. 6.5 Perplexity of the i^{th} memory cell when $\alpha^{(i)}$ was set to 1 for vanilla AMN model.

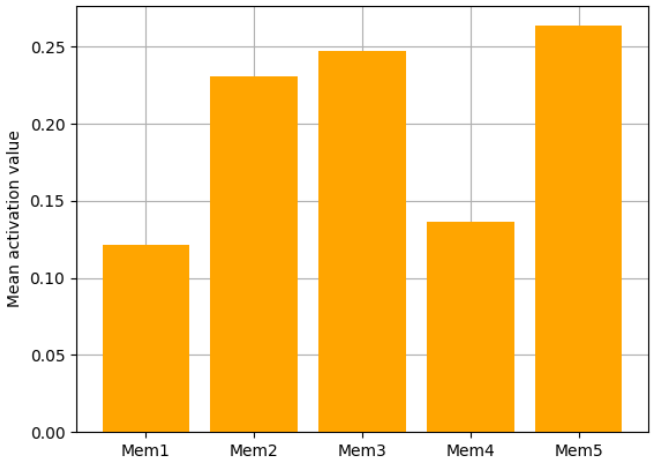
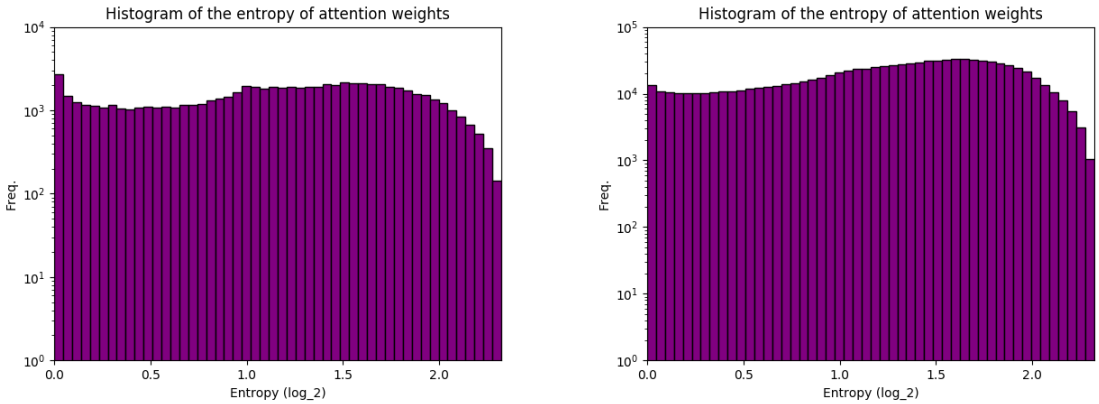


Fig. 6.6 Histogram of attention weights on the training set for AMN + Anneal.

distribution for the attention-entropy can be observed for the validation set, which is shown in the same figure.

Despite the more plausible attention mechanism, the model still performed poorly when evaluated with perplexity. One possible explanation is that the memcells overfitted the dataset. To confirm this hypothesis, the training and validation perplexities for each memcell was computed, which is shown in Figure 6.8. The figure shows that almost all of the memcells have seem to overfit the dataset, with the exception of the third memory cell. This explains



(a) Training set. (b) Validation set.

Fig. 6.7 Attention-entropy on the training and validation set from AMN + Anneal.

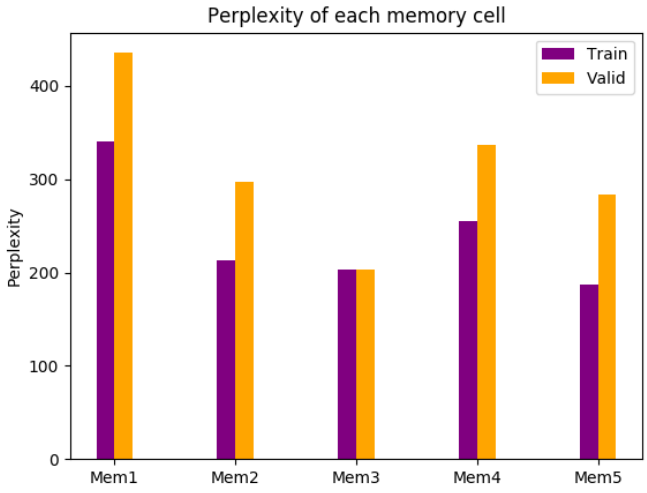


Fig. 6.8 Perplexity of the i^{th} memory cell when $\alpha^{(i)}$ was set to 1 for AMN + Anneal.

why worse perplexity was observed from this model, despite the more plausible attention behaviour.

Indeed, these results demonstrate an important principle. Namely, a seemingly well-behaved attention mechanism does not necessarily imply good performance. If the memcells are not capable of capturing good context representations for the word-history, the performance of the model will still suffer – regardless of the optimality of the attention mechanism.

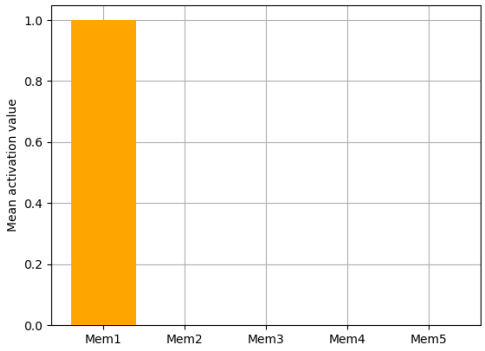


Fig. 6.9 Mean attention weights on the training set for AMN + Drop-Contr.

Training with different methods of dropout

- **AMN + Drop-Contr** learned a trivial attention mechanism where the output layer was completely dominated by the first memory cell (see Figure 6.9). This result was expected since Drop-Contr does not address the problem of dying memcells, nor does it address the problem of overfitting memcells (see 4.3.3). Instead, the principal effect achieved from dropping the controller was variance-reduction in the attention weights. This is shown implicitly in Figure 6.11 by the higher average attention-entropy in AMN + Drop-All.

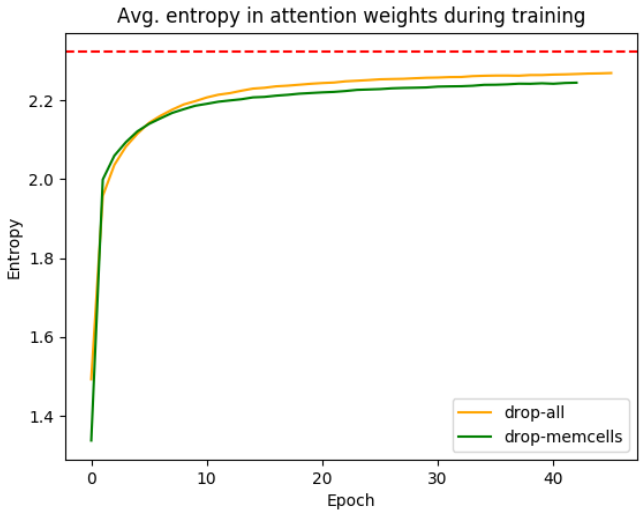
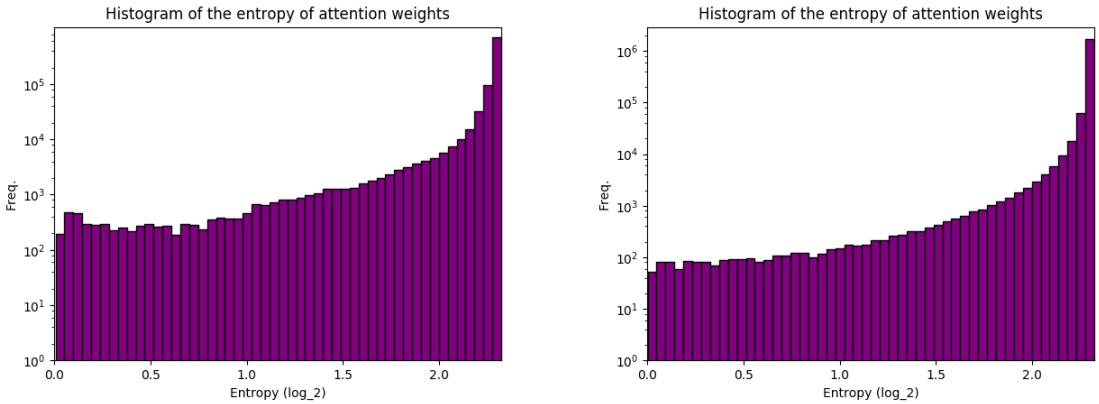
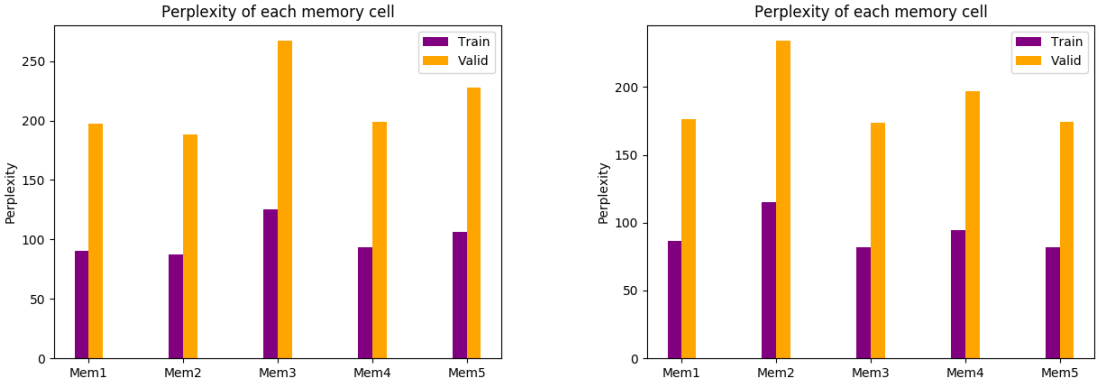


Fig. 6.10 Mean attention-entropy (over the entire corpus) in AMN + Drop-Mem and AMN + Drop-All for each training epoch.



(a) AMN + Drop-Mem (b) AMN + Drop-All

Fig. 6.11 Attention-entropy on the training set at convergence for (AMN + Drop-Mem) and (AMN + Drop-All).



(a) AMN + Drop-Mem (b) AMN + Drop-All

Fig. 6.12 Perplexity of the i^{th} memory cell when $\alpha^{(i)}$ was set to 1 for AMN trained with dropout.

- Both **AMN + Drop-Mem** and **AMN + Drop-All** learned a more plausible attention mechanism than AMN + Drop-Contr. During training, the averaged attention-entropy for both models were consistently increasing, which is shown in Figure 6.10. At the end of training, the attention weights of both AMN + Drop-Mem and AMN + Drop-All were relatively evenly distributed across all memcells, for almost all inputs. This is shown by the high attention-entropy for both models in Figure 6.11.

Model	Train	Valid
AMN	82	144
AMN + Implicit	77	141
AMN + Drop-Mem + Implicit	67	104
AMN + Anneal + Drop-Mem + Implicit	70	**103**

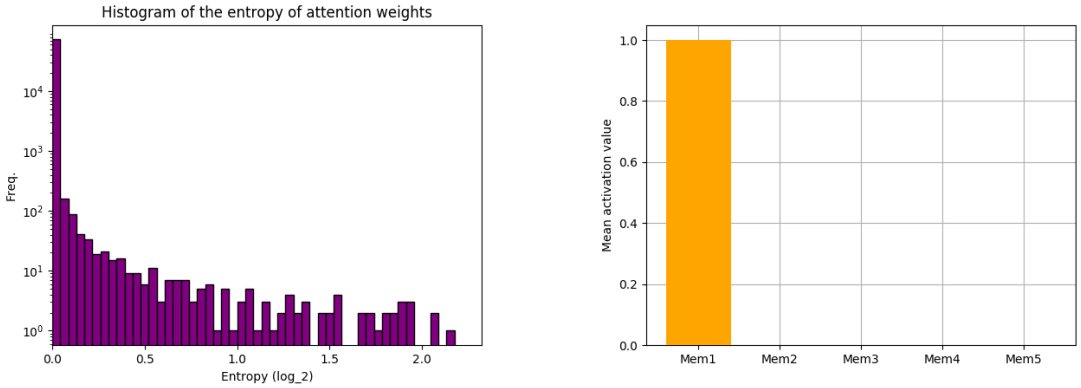
Table 6.3 Perplexity for AMN trained with ITL. Lower is better.

6.3.3 Regularization with Implicit-Target Loss

The results from training with ITL are shown in Table 6.3. The AMN model trained only with ITL performed slightly better than the vanilla AMN model, but the difference is minimal. However, when an AMN was trained with both ITL and dropout on the memcells, the model achieved a significant reduction in perplexity.

Training with ITL

The attention-entropy plot in Figure 6.13a showed that AMN + ITL was consistently very selective regarding which memcells it activated.



(a) Attention-entropy.

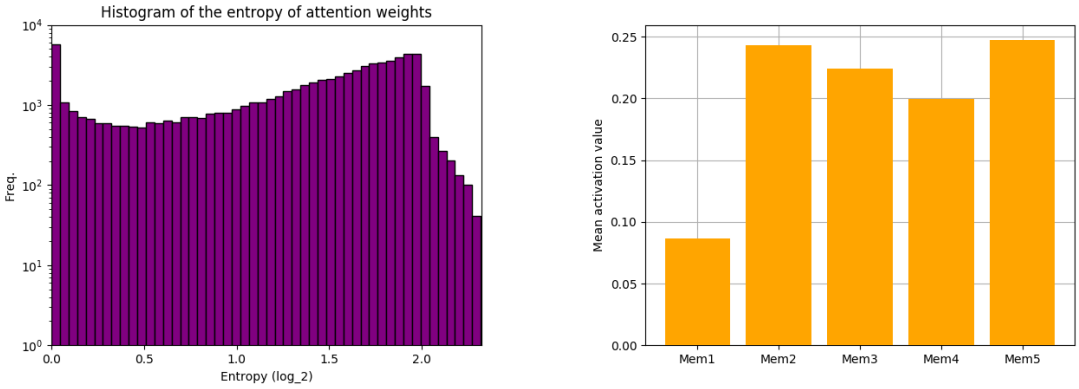
(b) Mean memcell activation.

Fig. 6.13 Attention-behaviour on the validation set from AMN + Implicit.

The plot in Figure 6.13b showed that the model only activated the first memcell and ignored the rest. Thus, training with ITL on its own still resulted in the same problem of dead memcells, as was the case with the vanilla AMN.

Combining ITL with dropout in memcells

However, combining ITL with dropout in the memcells led to a number of more interesting model behaviors. Firstly, the attention-entropy plot in Figure 6.14a showed that the model



(a) Attention-entropy.

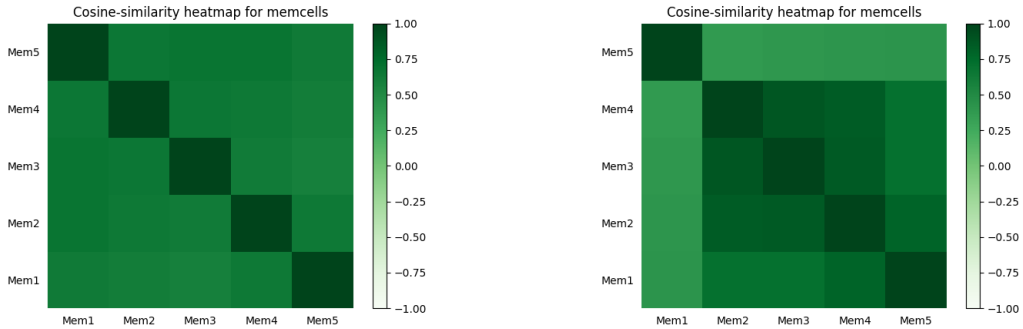
(b) Mean memcell activation.

Fig. 6.14 Attention-behaviour on the validation set from AMN + Drop-Mem + Implicit.

tend to behave in a much more bimodal manner in terms of how it focused its attention. For most of the inputs, the attention-entropy was either very high or very low. This marks a sharp contrast with the attention behaviour of AMN + Drop-Mem, where the attention-entropy tend to be consistently high across all inputs.

Secondly, the mean memcell activation plot in Figure 6.14b showed that all memcells were frequently activated by the model. In particular, this plot showed that the model was not relying on a single memcell to form the interpolated output. Instead, the model was performing a "switching" behaviour where the attention shifted from memcell to memcell depending on what the input was.

Impact on memcell specialization from ITL



(a) AMN + Drop-Mem.

(b) AMN + Drop-Mem + Implicit.

Fig. 6.15 Impact on pairwise cosine-similarity of memcells from training with ITL. Darker colors indicates high similarity, lighter colors indicates lower similarity.

A comparison of the cosine-similarity heat-maps for (AMN + Drop-Mem) and (AMN + Drop-Mem + Implicit) is shown in Figure 6.15. AMN + Drop-Mem showed a low-degree of specialization within each memcells, which suggests that the its strong performance was mainly due to memory cell ensembling. AMN + Drop-Mem + Implicit showed similar co-similarity patterns for its memcells, but with some noticeable differences. In particular:

1. The first memcell differentiated from the rest of the memcells, which suggests it was used to model uncommon word histories.

2. The second, third, and fourth memcells exhibit a very high degree of co-similarity, which suggests they are being ensembled for modeling the frequent word-histories.

These results are consistent with the hypothesis that ITL combined with memcell-dropout encourages specialization.

6.3.4 Training a Larger AMN

Model	**Num. Param**	**Train**	**Valid**	**Eval**
Large-RNN + Dropout	16m	81	146	139
Large-GRU + Dropout	18m	67	115	114
Large-LSTM + Dropout	20m	65	127	117
Large-AMN + Anneal + Drop-Mem	19m	64	102	96
Large-AMN + Anneal + Drop-Mem + ITL	19m	55	98	**91**

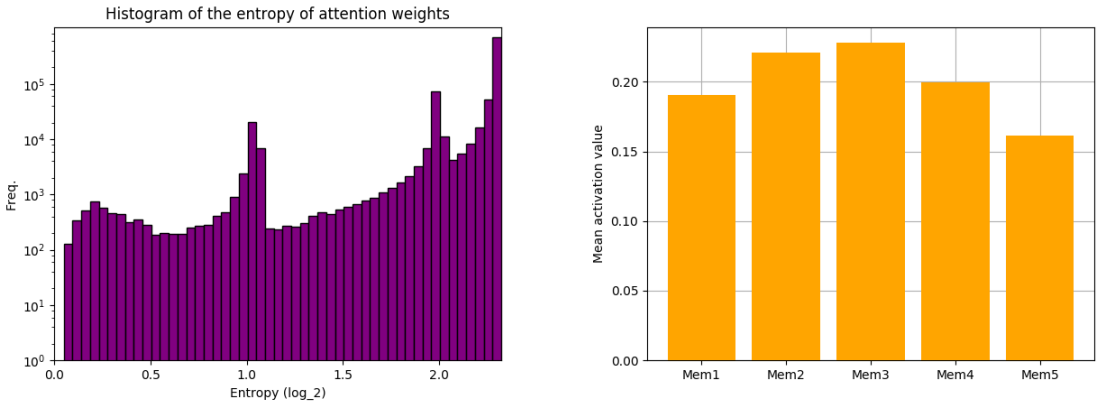
Table 6.4 Perplexity results for large models.

The results in Table 6.4 show consistent perplexity deductions from the large AMN model, with a relative test perplexity decrease of roughly 25% against the best performing GRU baseline. In particular, AMN was able to achieve greater perplexity deductions than the LSTM model, despite having fewer model parameters. Training AMN with ITL resulted in the best performing model, which was consistent with the results from training the smaller models.

Analysis of attention and memcells

Interestingly, the attention-entropy plot in Figure 6.16 reveal a distinctly different shape compared with the attention-entropy plots for the smaller models. Three distinct peaks for the attention-entropy values emerged at 1.0, 2.0, and 2.32.

An equally interesting result emerged for the memcell perplexities in Figure 6.20. The first, second and fourth memcells have exceptionally high perplexity on the training set, but



(a) Attention-entropy (b) Mean memcell attention weights

Fig. 6.16 Attention-behaviour on the training set from Large-AMN + Anneal + Drop-Mem

each of these memcells were consistently assigned non-negligible attention weights on the training set. Moreover, the first memory cell performed well during validation despite the exceptionally poor performance on the training set.

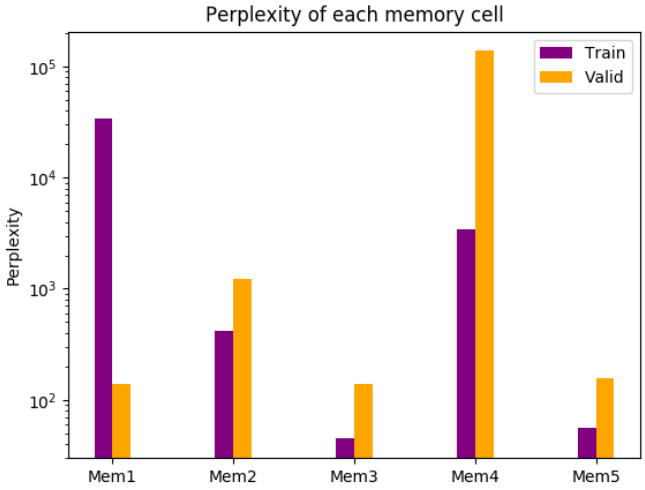


Fig. 6.17 Perplexity of the i^{th} memory cell when $\alpha^{(i)}$ was set to 1 for Large-AMN + Anneal + Drop-Mem.

These results suggests that the model might be using the third and fifth memcells for modeling generic context representations applicable for all sentences in the corpus. The first, second, and fourth memcells were possibly used for modeling topic-specific contexts.

6.3.5 Comparison with Other Neural Topic-Models

Model	**Hidd.**	**Valid**	**Eval**
KN-5 (Mikolov et. al 2012)	NA	148	141
RNN + LDA (Mikolov et. al 2012)	100	132	126
TopicRNN (Dieng et. al 2017)	100	129	122
TopicGRU (Dieng et. al 2017)	100	118	112
TopicLSTM (Dieng et. al 2017)	100	126	118
AMN + Drop-Mem	100	108	103
AMN + Drop-Mem + Implicit	100	104	97
AMN + Anneal + Drop-Mem + Implicit	100	103	**95**
TopicRNN (Dieng et. al 2017)	300	118	112
TopicGRU (Dieng et. al 2017)	300	100	97
TopicLSTM (Dieng et. al 2017)	300	104	100

Table 6.5 Comparison with explicit neural topic-models on PTB.

A comparison of AMN against works that perform *explicit* topic-modeling is shown in Table 6.5. Results are only shown for models that used the same PTB dataset and has a comparable hidden layer size.

If models are evaluated strictly according to perplexity performance, these results suggest that the *implicit* topic modeling approach espoused by AMN is preferable to the explicit topic modeling approach adopted by the other models. It is worth noting that the RNN + LDA model in [2] was not regularized with dropout. Moreover, results were not available for GRU/LSTMs trained with LDA features. This partially complicates the comparison, since AMN models were trained with GRUs for modeling the underlying context vectors in the memcells.

6.3.6 Conclusions

In this section, each of the training techniques for learning the attention mechanism were evaluated and analyzed. To briefly reiterate the main results:

1. A properly trained AMN obtains roughly 11-15% relative perplexity gain against the best RNN/GRU/LSTM baselines.

2. AMN fails to generalize due to the problem of dead memcells. This is caused by biased attention from training with random initialization.

3. Attention-weight annealing helps AMN to learn a smoother attention distribution, but memcells overfit as a consequence.

4. Dropout applied to the memcells leads to a plausible attention mechanism, with less overfitting in the memcells.

5. However, it is unclear whether dropout should be applied to the controller.

6. Augmenting the loss function with the implicit-target loss term leads to the best results when applied in conjunction with memcell dropout.

7. Increasing the model capacity of the AMN model leads to even better performance.

8. A rough comparison with existing work on neural topic-models indicate that AMN performs better in terms of perplexity.

Analysis of the attention mechanism and the memcells showed that the performance of the AMN model is highly correlated with the robustness of the attention mechanism. The results suggests that an improved attention mechanism leads to a higher-degree of specialization in the memcell context representations.

AMN still occasionally learned trivial attention mechanisms when using dropout in the memcells. This can be attributed to the same problem that the vanilla AMN model suffered from. Namely, the initial bias in the attention weights induced by random initialization sometimes caused the model to only tune a single memcell. This suggests that AMN should be trained using both attention-weight annealing and memcell dropout to reduce the sensitivity to initialization.

Training with ITL and memcell-dropout leads to considerable improvements in perplexity over training with either technique independently. The analysis provided suggests that this is due to the ability of ITL to negate the overly-strong smoothing effects of dropout on the context representations. In particular, AMN + Drop-Mem + Implicit learned a highly non-trivial attention mechanism and appeared to have performed some form of specialization in the memcells.

Training with the larger AMN model led to the best results, with significant perplexity gains over comparable baselines that were competitively trained with dropout.

6.4 BBC Multi-Genre Broadcast News (MGB)

In this section, AMN was evaluated for language modeling on the high-structured MGB dataset. Perplexity results are first reported, followed by an analysis of the attention and memory cells. In particular, the analysis will be focused on:

1. Examining the behavior of the learned attention mechanism.

2. Examining the implicit topic learning ability of AMN.

6.4.1 Experimental Setup

The same experimental setup from PTB was adopted for the MGB experiments, with a few minor changes (see 6.3.1). All analysis will be restricted to the AMN + Drop-Mem model since the performance and behavior of this model was sufficient to demonstrate the main goals of these experiments.

MGB The training corpus consisted of roughly 12 million words with a validation corpus of roughly 200k words. The vocabulary consisted of the top 35k most frequent words in the training data and was used for both the input and output layer. Out-of-vocabulary words were replaced with the special token <unk>.

Models The AMN model for MGB was trained with the same setup used for the large PTB AMN model with 500 recurrent units. For the baselines, an RNN and GRU with and without dropout were trained, each with 575 hidden units. All models had roughly 43-44 million parameters. The considerable increase in the number of model parameters was mainly due to the increase in the vocabulary size. Otherwise, all models used the same hyper-parameter configuration as in PTB. Attention-weight annealing for AMN was not strictly necessary but was implemented to avoid the need to tune the weight initialization scale.

6.4.2 Results

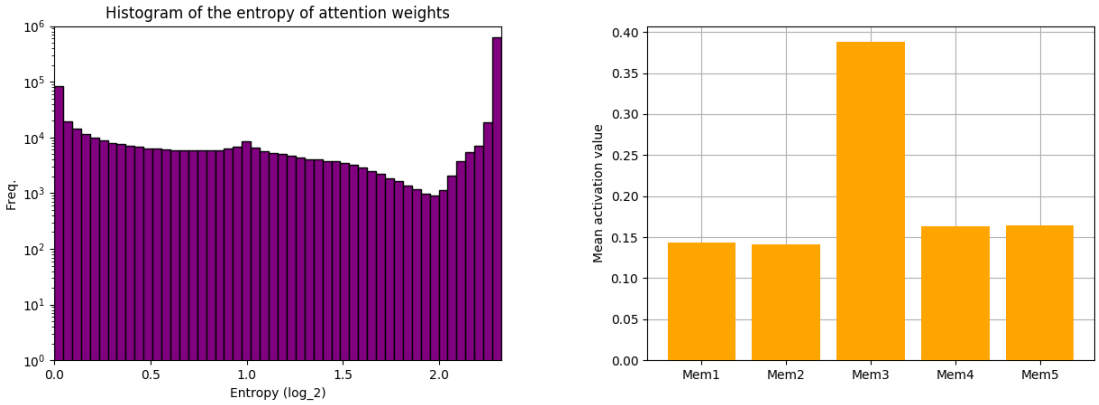
Model	Num. Param	Train	Valid
RNN + Dropout	43m	98	113
GRU	44m	62	83
GRU + Dropout	44m	107	89
AMN	44m	120	96
AMN + Anneal + Drop-Mem	44m	56	**75**
AMN + Anneal + Drop-Mem + ITL	44m	70	**73**

Table 6.6 Perplexity for 12M MGB.

The perplexity results for the MGB LM experiments are reported in Table 6.6. Some of the models had reached convergence before training performance degraded below validation performance. Thus the problem of under-fitting is of particular concern for this dataset. Indeed, in the case of the GRU model, training with dropout had in fact led to worse performance.

The best performing model came from the AMN model trained with Drop-Mem and ITL, with a relative gain of 12% compared to the best baseline.

6.4.3 Analysis of Attention-Mechanism and Memcells



(a) Attention-entropy

(b) Mean memcell attention weights

Fig. 6.18 Attention-behaviour from AMN + Anneal + Drop-Mem on 1 million words randomly sampled from the 12M MGB training set.

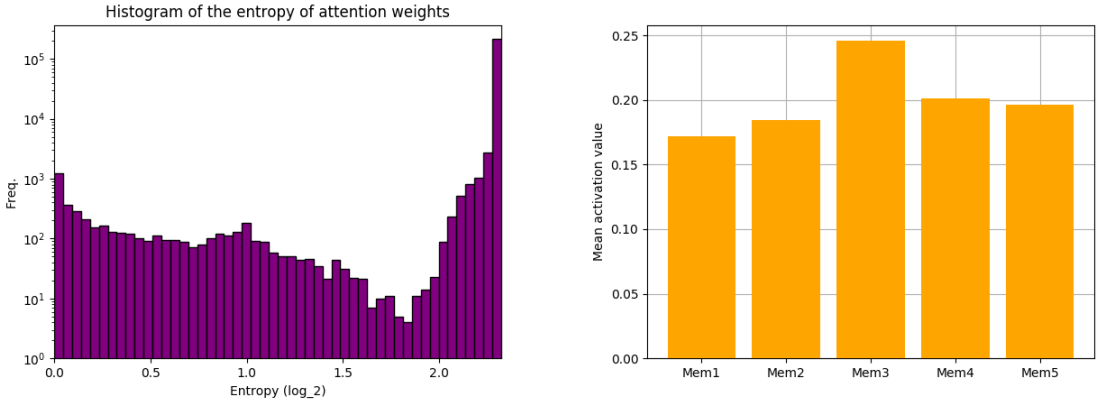
The attention-entropy for AMN during training was relatively well-distributed; this can observed in Figure 6.18a. The plot showed two peaks at attention-entropy values 0 and 2.32, which corresponds to a sharp attention and an evenly distributed attention respectively. Figure 6.18b showed that the third memory cell had the highest mean activation compared with the other memory cells. This was due to the model frequently assigning an attention weight of 1.0 to the third memcell on the training set, which was also the main cause for the cases where the model had a low attention-entropy.

The attention-behaviour of AMN at test-time was similar to the behavior observed during training, which is shown in Figure 6.19. However, the following key differences were observed:

1. The mean activation on the third memcell was considerably lower during test time.

2. The frequency of high attention-entropy values increased by a significant margin.

Interestingly, the memcell perplexity plot in Figure 6.20 showed that the third memcell had a very high perplexity when activated on its own. The same can be observed for the fourth and fifth memcells. It is clear from the attention plots that these memcells did not converge to a dead state since they were frequently assigned a high attention during training.



(a) Attention-entropy (b) Mean memcell attention weights

Fig. 6.19 Attention-behaviour from AMN + Anneal + Drop-Mem on MGB validation set.

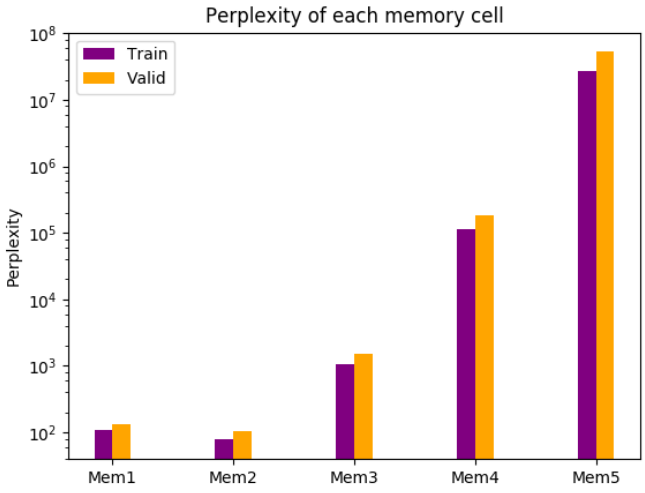


Fig. 6.20 Perplexity of the i^{th} memory cell when $\alpha^{(i)}$ was set to 1 for AMN + Anneal + Drop-Mem on MGB.

One possibility is that the model was using these memcells for tracking particular implicit topics. This will be further discussed in the following sections.

6.4.4 Attention Behaviour on Less-Frequent Words

One interesting observation made during the training of the AMN + Anneal + Drop-Mem model was that it will sometimes suddenly *focus* its attention on a particular memory cell when it observed an uncommon word. Some examples of these words were “transcended”, “dreamcoat” and “sirloin”.

To quantify this at a corpus-level, an average attention-entropy value $A(w_k)$ was computed for each word in the training corpus. The vocabulary words were then ranked in decreasing order, such that words with high attention-entropy were ranked near the top and words with low attention-entropy were ranked near the bottom. Intuitively, this word ranking expressed which words caused the model to divide/focus its attention. Some examples drawn from this word-ranking are shown in Figure 6.21; these words caused the model to have a sharply focused attention. To have a baseline of comparison, the corpus word-ranking was also computed – this ranking ranks words according to their frequency of occurrence in the corpus.

...
34976 **dreamcoat**
34977 walkover
34978 haha
34979 grumped
34980 contradicts
34981 funk
34982 poos
34983 highlander
34984 corrode
...
34994 **sirloin**
34995 formalities
34996 tolgo
34997 hmph
34998 choosy
34999 passover

Fig. 6.21 Examples of low-rank words in the attention-entropy based word ranking.

The word ranking based on the attention-entropy was then plotted against the corpus word-ranking, which produced the scatter-plot shown in Figure 6.22. The resulting plot roughly followed a "U" shape, with three distinct clusters of words emerging from the graph. To visualize this more clearly, Figure 6.23 shows a heat-map indicating the regions of high word-density in Figure 6.22. These clusters can be observed in the:

1. Top-left region, which corresponds to low-frequency words with high attention-entropy. These are words where the AMN model was using all of the memory cells for prediction.

2. Bottom-centre region, which corresponds to high-frequency words with medium attention-entropy.

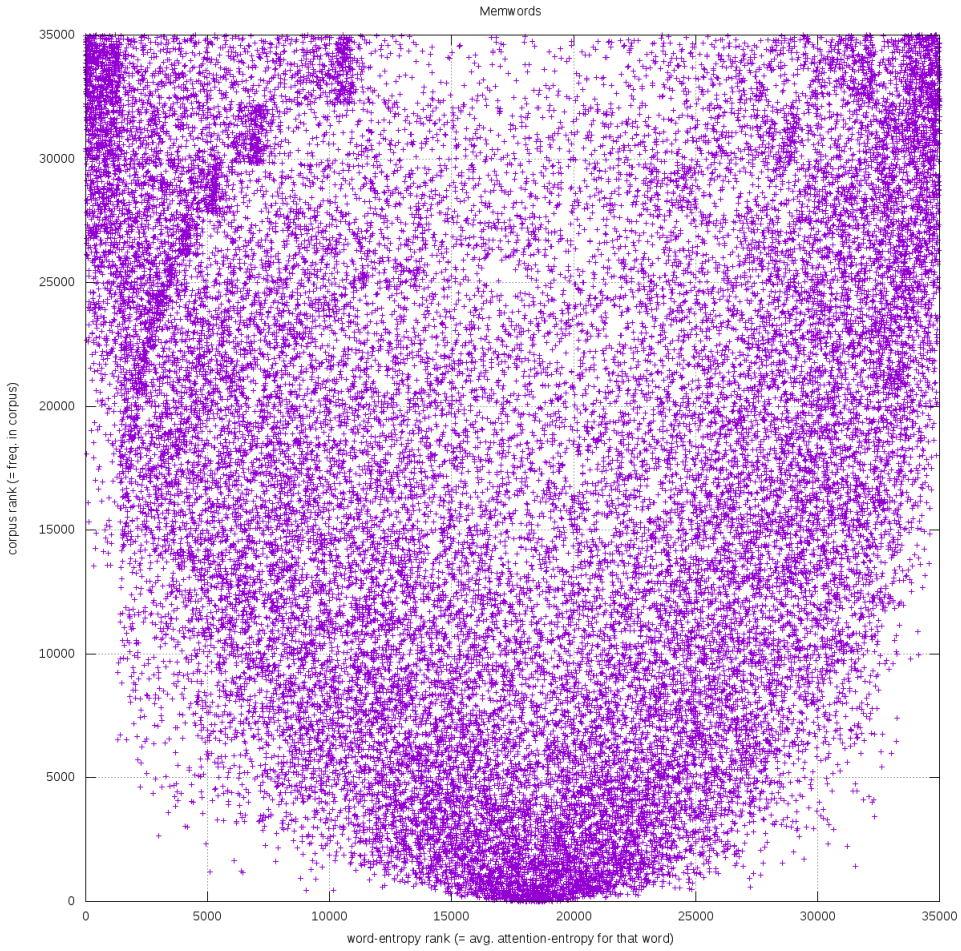


Fig. 6.22 Word-ranking based on attention-entropy versus word-ranking based on corpus-rank.

3. Top-right region, which corresponds to low-frequency words with low attention-entropy. In this case, the AMN model was primarily focusing on one or two memory cells for prediction.

Figures 6.22 and 6.23 indicate the word regions where AMN had a high/low attention focus, but they do not show which specific memcells were being used by the model. To address this problem, an attention heat-map was computed for each memcell over the word-regions in Figure 6.24. This is shown in Figure 6.24. These attention heat-maps were computed by taking the average attention value assigned to the k^{th} memcell when the word w_j was observed. So the heat-maps essentially indicate which memcells were responsible for the word regions with low/high attention-entropy.

Figure 6.24 shows that the third memory cell was primarily responsible for attending to the words in the top-right region, while all the other memory cells were responsible for

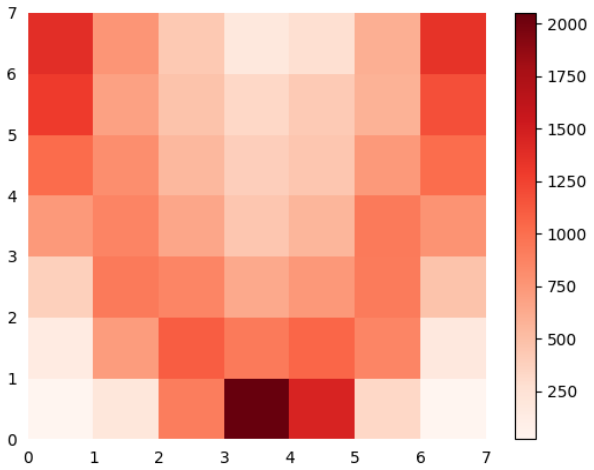
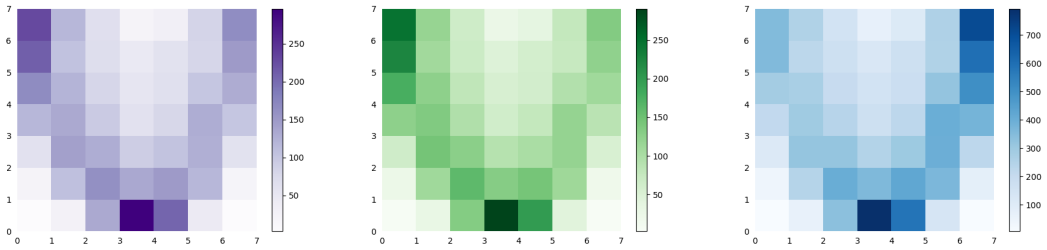
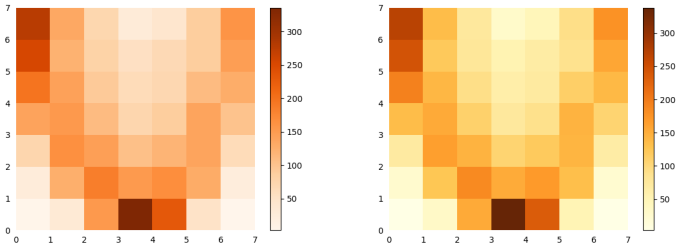


Fig. 6.23 Heat-map indicating the regions of high word-density for Figure 6.22.



(a) Memcell1 (b) Memcell2 (c) Memcell3



(d) Memcell4 (e) Memcell5

Fig. 6.24 Attention heat-maps from each memory cell for Figure 6.22.

attending to the words in the top-left region. This explains why these two word-regions had low and high attention-entropy respectively. This also explains why the third memory cell had a significantly higher mean activation value on the training set when compared with the activations for the other memcells – it was being used to model a specific subset of the vocabulary words.

6.4.5 Implicit Topics Modeling in Memcells

In the previous section, the analysis of the attention mechanism showed that AMN was capable of learning an attention mechanism that partitioned the vocabulary across the memcells. In this section, an analysis of the memcells is provided to observe whether memcells were specializing in topics in the training corpus.

Word-rankings were computed by examining the attention for each memory cell across all words in the vocabulary. Specifically, the word-ranking induced by the memcell $m^{(k)}$ was produced by:

1. Computing the mean activation $\alpha^{(k)}$ when the word w_j was observed.
2. Then assigning a rank to each word according to its mean activation for $m^{(k)}$. Words that caused high activation in $m^{(k)}$ were ranked near the top, while words that caused low activation were ranked near the bottom.

To examine whether these word-rankings were correlated with the genres in the training corpus, the intersection of the top-100 words in each ranking and each genre vocabulary was computed. The results of this are shown in Table 6.7. This table suggests that the memcells were indeed performing a weak-form of topic adaptation in the memcells.

Mem1	**Mem2**	**Mem3**	**Mem4**	**Mem5**
(15) announcer	(1) resurrect	(1) choosy	(4) annihilation	(1) orchestrate
(16) inventiveness	(2) residue	(2) formalities	(5) slashing	(6) socialising
(24) storytellers	(3) marauder	(8) dreamcoat	(17) grimness	(28) romanticised
(27) wowed	(5) naturalistic	(15) dumbstruck	(20) feuds	(29) candlelit
(29) bamboozled	(13) deathbed	(21) emmy	(28) panicky	(37) uninterrupted
(42) phrasing	(21) undertakers	(41) resplendent	(43) legionnaires	(39) amour

Fig. 6.25 Examples of high-rank words in each memcell word ranking (with rank number).

Genre	**Vocab Size**	**Mem1**	**Mem2**	**Mem3**	**Mem4**	**Mem5**
news	45247	0.47	0.48	0.53	0.48	0.51
events	24923	0.25	0.31	0.34	0.30	0.29
competition	32677	0.42	0.28	0.45	0.41	0.34
childrens	26703	0.38	0.33	0.24	0.22	0.22
advice	29959	0.40	0.33	0.31	0.35	0.34
documentary	36262	0.48	0.49	0.42	0.51	0.45
comedy	19961	0.18	0.16	0.20	0.17	0.20

Table 6.7 Percentage of top-100 words in memcell word ranking that appears in the vocabulary of each genre-specific corpus. Values indicating high memcell topic-specialization are highlighted.

6.4.6 Conclusions

To re-iterate the main results of the MGB language modeling experiments:

1. An AMN obtains a 12% relative gain over comparable baselines at test time when evaluated with perplexity.

2. AMN was capable of displaying interesting attention behaviors when dealing with uncommon words. In particular, AMN will tend to either ensemble the context representations or use only a single representation stored in one of the memcells.

3. Analysis of the memcell-induced word-rankings showed that AMN was capable of performing a weak form of implicit-topic adaptation in the memcells.

The results from analyzing the attention mechanism were encouraging since they suggest that the performance gains from the AMN model during training and validation were due to the use of multi-context representations of the word-history.

One potential criticism is that they were derived from the training set, which may exaggerate their applicability at test time due to potential overfitting. However, it is worth noting that the attention-entropy distribution of the AMN model at test-time was consistent with its attention-entropy distribution at the end of training. Moreover, the AMN model was able to obtain strong generalization performance on the validation set. These two factors present strong evidence for why the implicit topic modeling behavior of AMN will likely generalize at test time.

Chapter 7

Speech Recognition

In this chapter, the AMN model was evaluated for ASR performance. The main motivation for these experiments was to examine whether the improved language modeling performance observed in Chapter 6 translates to ASR improvements. This was performed by rescoring a set of N-best lists generated by an ASR system, using the implementation described in Section 5.2.1. The MGB3 corpus was used as the primary dataset, with 100-best lists used in N-best rescoring.

7.1 Word Error Rate (WER)

The word error rate is commonly defined as:

$$\text{WER} = \frac{\text{Sub + Del + Ins}}{\text{Number of Words}} \tag{7.1}$$

where Sub is the substitution error, Del is the deletion error, and Ins is the insertion error computed using the Levenshtein distance (also known as the edit distance). It is commonly used for evaluating speech recognition systems since it computes an absolute measure for the number of errors made by a system.

The WER serves as a complementary metric to the perplexity measure for evaluating LM performance, and the two tend to be positively correlated [71]. A detailed discussion on the relationship between WER and perplexity is provided in [71].

7.2 MGB3

7.2.1 Dataset

The MGB3 corpus consists of a set of manually-transcribed subtitles with roughly 4M words (MGB3-Man) and a set of automatically-transcribed subtitles with roughly 800M words (MGB3-Sub). This corpus is similar to the one used in the language modeling experiments (see 6.1). RNNLM training was conducted with only MGB3-Man for faster training times. A development-set (dev-17a) containing 65k words and 7k sentences was used for testing.

Dataset	Num. Words	Num. Sentences	Avg. Sentence Length	Vocab Size
MGB3-Man	4.3m	500k	8 words	40k

Table 7.1 Corpus statistics for the Speech Recognition datasets.

7.2.2 Experimental Setup

Unless otherwise stated, all RNNLM models used the same hyper parameter configuration described in the PTB language modeling experiments (see 6.3.1).

Vocabulary The input and output layers used the same vocabulary, which contained approximately 40k words. All out-of-vocabulary words were mapped to <unk>.

Models An interpolated Kneser-Ney n-gram model [33] was trained on both MGB3-Man and MGB3-Sub, where the n-gram order was 3. Baseline RNNLMs were trained using an RNN and a GRU with 512 recurrent units. The AMN model was trained with attention-weight annealing, memory-cell dropout, and ITL. The dropout probability was set to 0.35. The batch size was set to 10.

Interpolation Scales When computing the WER, the grammar-scale was not tuned and was set to a default value of 12.0. For the value of the interpolation-weight, $\lambda = 0.7$ was chosen since this value appeared to work well in the preliminary experiments. It is likely that further reductions of 0.1-0.2 WER in each model can be obtained by tuning λ, although this was not investigated.

Model	Datasets	Perplexity
3-gram	Man+Sub	127
RNN	Man	198
GRU	Man	164
AMN	Man	142

Table 7.2 Perplexity on dev17a. RNNLMs were not interpolated with the 3-gram model when computing the perplexity.

Model	WER
3-gram	28.5
RNN + 3-gram	27.8
GRU + 3-gram	27.2
AMN + 3-gram	27.0

Table 7.3 WER from 100-best rescoring on dev17a.

7.2.3 Results

Perplexity results are shown in Table 7.2, with WER results reported in Table 7.3. A smooth decrease in the WER can be observed as the perplexity decreased. Significant WER improvements were observed after interpolating with the RNNLMs, which is consistent with known results [39, 69]. The best results were given by the AMN model, which obtained a 0.2 WER absolute improvement over the GRU model.

For reference, a comparable 512-hidden unit RNN and GRU model trained using the CUED-RNNLM toolkit (with the same word lists) will achieve 27.5 and 27.2 WER respectively when using a grammar-scale of 12.0 and equal interpolation weights. Tuning the interpolation weight results in 27.3 WER and 27.1 WER in the CUED-RNN and CUED-GRU models respectively.

7.3 Conclusions

The main result established in this chapter was that the perplexity gains from AMN do indeed translate to improved ASR performance. In these experiments, the AMN model obtained a 0.2 WER absolute improvement over the GRU baseline.

If time was permitting, it would have been useful to compare the WER performance of AMN with a RNNLM trained with LDA topic features. This would have allowed for a direct evaluation of the implicit topic modeling ability of AMN. It also would have been useful to observe how the performance of AMN scales as the size of the training set is increased. The

training set used to tune AMN was relatively small by neural-network standards, so it was quite possible that the full modeling potential of AMN was not realized in these experiments.

Chapter 8

Conclusion and Future Work

8.1 Conclusions

Language models are a core component of standard ASR pipelines, which motivates their research. RNNs have obtained state-of-the-art results for many language modeling problems due to their ability to encode a long word-history into a compact context vector. Standard methods for contextualizing RNNs with topics are typically based on using the features extracted from a topic model as additional inputs.

The AMN model presented in this dissertation offers an alternative approach based on memory and attention for implicit topic modeling. The language modeling results indicated that this was a promising approach, with significant gains in perplexity observed on both PTB and MGB over baseline GRU and LSTM models. Perplexity gains were also observed over explicit neural topic-models presented in the research literature for PTB. Furthermore, AMN was also able to obtain WER improvements over the baselines during the speech recognition experiments.

Analysis of the attention mechanism showed that AMN was capable of learning plausible patterns of attention for improved generalization performance. In the case of MGB, the results indicated that the AMN model was able to learn an attention on the training set which assigned words to memory cells according to the implicit topic in the stored representation. These qualitative results are encouraging, since they provide justification for the strong generalization performance of AMN.

8.2 Summary of Dissertation

In Chapter 4, a neural network model that used a recurrent attention mechanism to attend to multiple dynamic memory cells was introduced. This model was referred to as an Active Memory Network (AMN). Each of the memory cells store a rich, context representation of the word-history, which could be used for implicit topic modeling. Analysis of the weight-gradients demonstrated potential optimization problems during training. This led to the presentation of some training techniques as possible solutions. A novel regularization term was also introduced to improve the training objective for AMN. This regularization term – which was referred to as the implicit target loss (ITL) – provides an implicit, time-varying training signal to the memory cells for improving the quality of the learned context representations.

In Chapter 6, language modeling experiments were conducted on the Penn Tree Bank and BBC Multi-Genre Broadcast News corpora. The results showed that AMN significantly outperformed comparable GRU and LSTM models when trained using the correct techniques. The performance of AMN was further improved when scaled to a larger model configuration. Similar perplexity gains were observed for the MGB corpus. An in-depth analysis of the learned attention mechanism was provided for the MGB experiments, which showed that AMN was capable of learning a plausible attention mechanism for implicit-topics modeling.

In Chapter 7, the AMN model was evaluated for speech recognition using N-best hypotheses rescoring. AMN obtained perplexity gains over the RNN and GRU baselines, which resulted in corresponding WER improvements.

8.3 Future Work

8.3.1 Applying ITL Towards RNN Regularization

Although ITL was developed within the context of the AMN framework, it is simple to generalize it across all RNN-based models. For example, ITL can be used to enforce smooth-trajectory constraints on the change in the hidden-state of a RNN.

$$\alpha_j = \frac{1}{K} \tag{8.1}$$

$$I_j = \sum_{z=1}^{K} \alpha_z h_{t-z+1} \tag{8.2}$$

$$R(\theta) = \lambda \sum_{j=1}^{K} \alpha_j \| I_j - h_{t-j+1} \|^2 \tag{8.3}$$

where K is the number of time-steps that the hidden-state should remain stable for. For RNNs where an attention mechanism is used [65], α_j can also be computed dynamically using the attention weights.

Intuitively, this can be interpreted as penalizing sudden changes in the hidden-state, which can potentially help an RNN to learn context representation with longer time spans into its hidden-state. This formulation is also related to the notion of using shortcut paths in residual/highway networks for efficient error-backpropagation across time [72, 73]. This is one of many different ways in which ITL can be applied towards improving the RNN hidden representation. Further research is required to establish the applicability of this technique.

8.3.2 Augmenting AMN with Global Memory

In theory, the MemN2N model in [1] could have been augmented to learn the global context of the corpora by adding an additional memory module containing the sentence embeddings in the corpus [1]. Under this formulation, the input sequence is used to generate a query vector, which could then be used to attend over the sentence embeddings in global memory. The resulting interpolated sentence embedding could then be interpreted as a global context vector g for computing the class-prediction $\hat{y}$ along with the response vector o and current sentence q. An AMN can in principle be similarly extended with such a global memory module. In an AMN augmented with global memory, the memcells could be used as a working memory cache for storing rich embeddings of the word-history, whereas the global memory module is used to access long-term context information observed in the inputs. This can potentially produce a highly versatile memory network that is applicable across a wide range of tasks.

[1] Unfortunately, the paper did not comment on this aspect of their work.

Bibliography

[1] S. Sukhbaatar, J. Weston, R. Fergus, *et al.*, "End-to-end memory networks," in *Advances in neural information processing systems*, pp. 2440–2448, 2015.

[2] T. Mikolov and G. Zweig, "Context dependent recurrent neural network language model.," *SLT*, vol. 12, pp. 234–239, 2012.

[3] X. Chen, T. Tan, X. Liu, P. Lanchantin, M. Wan, M. J. Gales, and P. C. Woodland, "Recurrent neural network language model adaptation for multi-genre broadcast speech recognition," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[4] S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, X. Liu, G. Moore, J. Odell, D. Ollason, D. Povey, *et al.*, "The htk book," *Cambridge university engineering department*, vol. 3, p. 175, 2002.

[5] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

[6] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 315–323, 2011.

[7] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[8] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[9] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[10] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pp. 6645–6649, IEEE, 2013.

[11] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[12] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, 2013.

[13] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[14] K. Xu, J. Ba, R. Kiros, K. Cho, A. C. Courville, R. Salakhutdinov, R. S. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention.," in *ICML*, vol. 14, pp. 77–81, 2015.

[15] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[16] R. Jozefowicz, W. Zaremba, and I. Sutskever, "An empirical exploration of recurrent network architectures," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 2342–2350, 2015.

[17] J. Weston, S. Chopra, and A. Bordes, "Memory networks," *arXiv preprint arXiv:1410.3916*, 2014.

[18] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[19] D. E. Rumelhart, G. E. Hinton, R. J. Williams, *et al.*, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[20] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks.," *ICML (3)*, vol. 28, pp. 1310–1318, 2013.

[21] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[22] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Transactions on audio, speech, and language processing*, vol. 20, no. 1, pp. 30–42, 2012.

[23] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?," *Journal of Machine Learning Research*, vol. 11, no. Feb, pp. 625–660, 2010.

[24] J. Martens, "Deep learning via hessian-free optimization," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 735–742, 2010.

[25] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[26] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[27] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *arXiv preprint arXiv:1409.2329*, 2014.

[28] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," in *Advances in Neural Information Processing Systems*, pp. 1019–1027, 2016.

[29] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, "Exploring the limits of language modeling," *arXiv preprint arXiv:1602.02410*, 2016.

[30] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *Computational linguistics*, vol. 18, no. 4, pp. 467–479, 1992.

[31] J. T. Goodman, "A bit of progress in language modeling," *Computer Speech & Language*, vol. 15, no. 4, pp. 403–434, 2001.

[32] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," in *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pp. 310–318, Association for Computational Linguistics, 1996.

[33] R. Kneser and H. Ney, "Improved backing-off for m-gram language modeling," in *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, vol. 1, pp. 181–184, IEEE, 1995.

[34] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[35] H. Schwenk, "Continuous space language models," *Computer Speech & Language*, vol. 21, no. 3, pp. 492–518, 2007.

[36] T. Mikolov, W.-t. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations.," in *hlt-Naacl*, vol. 13, pp. 746–751, 2013.

[37] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, pp. 3111–3119, 2013.

[38] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[39] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model.," in *Interspeech*, vol. 2, p. 3, 2010.

[40] T. Mikolov, S. Kombrink, L. Burget, J. Černockỳ, and S. Khudanpur, "Extensions of recurrent neural network language model," in *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pp. 5528–5531, IEEE, 2011.

[41] T. Mikolov, "Statistical language models based on neural networks," *Presentation at Google, Mountain View, 2nd April*, 2012.

[42] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.

[43] M. Sundermeyer, I. Oparin, J.-L. Gauvain, B. Freiberg, R. Schlüter, and H. Ney, "Comparison of feedforward and recurrent neural network language models," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 8430–8434, IEEE, 2013.

[44] J. G. Zilly, R. K. Srivastava, J. Koutník, and J. Schmidhuber, "Recurrent highway networks," *arXiv preprint arXiv:1607.03474*, 2016.

[45] X. Chen, X. Liu, Y. Wang, M. J. Gales, and P. C. Woodland, "Efficient training and evaluation of recurrent neural network language models for automatic speech recognition," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 24, no. 11, pp. 2146–2157, 2016.

[46] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Černockỳ, "Strategies for training large scale neural network language models," in *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*, pp. 196–201, IEEE, 2011.

[47] M. U. Gutmann and A. Hyvärinen, "Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 307–361, 2012.

[48] F. Morin and Y. Bengio, "Hierarchical probabilistic neural network language model.," in *Aistats*, vol. 5, pp. 246–252, 2005.

[49] A. Mnih and G. E. Hinton, "A scalable hierarchical distributed language model," in *Advances in neural information processing systems*, pp. 1081–1088, 2009.

[50] Y. Bengio, J.-S. Senécal, *et al.*, "Quick training of probabilistic neural nets by importance sampling.," in *AISTATS*, 2003.

[51] D. Gildea and T. Hofmann, "Topic-based language models using em," in *Sixth European Conference on Speech Communication and Technology*, 1999.

[52] X. Chen, "Scalable recurrent neural network language models for speech recognition,"

[53] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[54] A. B. Dieng, C. Wang, J. Gao, and J. Paisley, "Topicrnn: A recurrent neural network with long-range semantic dependency," *arXiv preprint arXiv:1611.01702*, 2016.

[55] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[56] S. Ghosh, O. Vinyals, B. Strope, S. Roy, T. Dean, and L. Heck, "Contextual lstm (clstm) models for large scale nlp tasks," *arXiv preprint arXiv:1602.06291*, 2016.

[57] Y. Miao, E. Grefenstette, and P. Blunsom, "Discovering discrete latent topics with neural variational inference," *arXiv preprint arXiv:1706.00359*, 2017.

[58] R. Lin, S. Liu, M. Yang, M. Li, M. Zhou, and S. Li, "Hierarchical recurrent neural network for document modeling.," in *EMNLP*, pp. 899–907, 2015.

[59] A. Gutkin, "Log-linear interpolation of language models," *Mémoire de DEA, University of Cambridge*, 2000.

[60] D. Klakow *et al.*, "Log-linear interpolation of language models.," in *ICSLP*, 1998.

[61] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *Neural computation*, vol. 3, no. 1, pp. 79–87, 1991.

[62] M. I. Jordan and R. A. Jacobs, "Hierarchical mixtures of experts and the em algorithm," *Neural computation*, vol. 6, no. 2, pp. 181–214, 1994.

[63] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[64] R. Soltani and H. Jiang, "Higher order recurrent neural networks," *arXiv preprint arXiv:1605.00064*, 2016.

[65] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[66] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[67] M. Ostendorf, A. Kannan, S. Austin, O. Kimball, R. M. Schwartz, and J. R. Rohlicek, "Integration of diverse recognition methodologies through reevaluation of n-best sentence hypotheses.," in *HLT*, vol. 91, pp. 83–87, 1991.

[68] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[69] S. Kombrink, T. Mikolov, M. Karafiát, and L. Burget, "Recurrent neural network based language modeling in meeting recognition," in *Twelfth Annual Conference of the International Speech Communication Association*, 2011.

[70] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[71] D. Klakow and J. Peters, "Testing the correlation of word error rate and perplexity," *Speech Communication*, vol. 38, no. 1, pp. 19–28, 2002.

[72] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[73] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," *arXiv preprint arXiv:1505.00387*, 2015.

Appendix A

Active Memory Networks

A.1 Derivation for Memcell Weight Updates

We first consider the equations for computing the attention weights and the final output o for a particular time step (the t subscript is omitted for clarity).

$$\beta^{(i)} = u \cdot m^{(i)} \tag{A.1}$$

$$\alpha^{(i)} = \frac{\exp(\beta^{(i)})}{\sum_j \exp(\beta^{(j)})} \tag{A.2}$$

$$o = \sum_{i=1}^{K} \alpha^{(i)} m^{(i)} \tag{A.3}$$

where K is the number of memory cells, u is the controller hidden state, $m^{(i)}$ is the i^{th} memory cell vector, $\alpha^{(i)}$ is the attention weight, and o is the interpolated output.

Note that the gradient of interest here is the derivative of the interpolated output o with respect to the weights $w^{(k)}$ used to compute $m^{(k)}$. Thus, the product rule can be applied to yield:

$$\frac{\partial o}{\partial w^{(k)}} = \sum_{i=1}^{K} m^{(i)} \frac{\partial \alpha^{(i)}}{\partial \beta^{(k)}} \frac{\partial \beta^{(k)}}{\partial m^{(k)}} \frac{\partial m^{(k)}}{\partial w^{(k)}} + \delta_{i,k} \alpha^{(i)} \frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.4}$$

where $\delta_{i,j}$ is the Kronecker delta function.

$$\delta_{i,j} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{else} \end{cases} \tag{A.5}$$

The partial derivative for the softmax with respect to the k^{th} logit is given by:

$$\frac{\partial \alpha^{(i)}}{\partial \beta^{(k)}} = \alpha^{(i)}(\delta_{i,k} - \alpha^{(k)}) \tag{A.6}$$

The partial derivative for the k^{th} logit is given by u.

$$\frac{\partial \beta^{(k)}}{\partial m^{(k)}} = u \tag{A.7}$$

After performing the appropriate substitutions, the following equation is obtained:

$$\frac{\partial o}{\partial w^{(k)}} = \sum_{i=1}^{K} m^{(i)}(\alpha^{(i)}(\delta_{i,k} - \alpha^{(k)}))(u)\frac{\partial m^{(k)}}{\partial w^{(k)}} + \delta_{i,k}\alpha^{(i)}\frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.8}$$

$$= \sum_{i=1}^{K} \alpha^{(i)}(\delta_{i,k} - \alpha^{(k)})\beta^{(i)}\frac{\partial m^{(k)}}{\partial w^{(k)}} + \delta_{i,k}\alpha^{(i)}\frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.9}$$

$$= \alpha^{(k)}\frac{\partial m^{(k)}}{\partial w^{(k)}} + \sum_{i=1}^{K} \alpha^{(i)}(\delta_{i,k} - \alpha^{(k)})\beta^{(i)}\frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.10}$$

For an expression without the Kronecker delta function, the sum in the second term can be re-distributed and simplified:

$$\frac{\partial o}{\partial w^{(k)}} = \alpha^{(k)}\frac{\partial m^{(k)}}{\partial w^{(k)}} + \sum_{i=1}^{K}\left[\alpha^{(i)}(\delta_{i,k})\beta^{(i)}\frac{\partial m^{(k)}}{\partial w^{(k)}} - \alpha^{(i)}\alpha^{(k)}\beta^{(i)}\frac{\partial m^{(k)}}{\partial w^{(k)}}\right] \tag{A.11}$$

$$= \alpha^{(k)}\frac{\partial m^{(k)}}{\partial w^{(k)}} + \sum_{i=1}^{K}\left[\alpha^{(i)}(\delta_{i,k})\beta^{(i)}\frac{\partial m^{(k)}}{\partial w^{(k)}}\right] - \sum_{i=1}^{K}\left[\alpha^{(i)}\alpha^{(k)}\beta^{(i)}\frac{\partial m^{(k)}}{\partial w^{(k)}}\right] \tag{A.12}$$

$$= \alpha^{(k)}\frac{\partial m^{(k)}}{\partial w^{(k)}} + \alpha^{(k)}\beta^{(k)}\frac{\partial m^{(k)}}{\partial w^{(k)}} - \sum_{i=1}^{K}\alpha^{(i)}\alpha^{(k)}\beta^{(i)}\frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.13}$$

$$= \alpha^{(k)}\frac{\partial m^{(k)}}{\partial w^{(k)}}\left(1 + \beta^{(k)} - \sum_{i=1}^{K}\alpha^{(i)}\beta^{(i)}\right) \tag{A.14}$$

which returns an expression that is proportional to $\alpha^{(k)}$ and the derivative of the memcell vector with respect to the weights used to compute it.

A.2 Derivation for Memcell Weight Updates from ITL

We will now derive the weight-updates for the k^{th} memory cell from using the implicit-target loss regularization term. Firstly:

$$\frac{\partial R(\theta)}{\partial w^{(k)}} = \lambda \sum_{i=1}^{K} \Big[\frac{\partial \alpha^{(i)}}{\partial w^{(k)}} \|o - m^{(i)}\|^2 + 2\alpha^{(i)} \Big(\frac{\partial o}{\partial w^{(k)}} - \delta_{i,k} \frac{\partial m^{(i)}}{\partial w^{(k)}} \Big) \Big] \tag{A.15}$$

$$= \lambda \sum_{i=1}^{K} \Big[\frac{\partial \alpha^{(i)}}{\partial w^{(k)}} \|o - m^{(i)}\|^2 + 2\alpha^{(i)} \frac{\partial o}{\partial w^{(k)}} \Big] - \lambda 2 \alpha^{(k)} \frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.16}$$

The derivations from Section A.1 can be re-used to obtain:

$$\frac{\partial o}{\partial w^{(k)}} = \alpha^{(k)} \frac{\partial m^{(k)}}{\partial w^{(k)}} + \sum_{z=1}^{K} \alpha^{(z)} (\delta_{z,k} - \alpha^{(k)}) \beta^{(z)} \frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.17}$$

$$\frac{\partial \alpha^{(i)}}{\partial w^{(k)}} = \frac{\partial \alpha^{(i)}}{\partial \beta^{(k)}} \frac{\partial \beta^{(k)}}{\partial m^{(k)}} \frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.18}$$

$$= \alpha^{(i)} (\delta_{i,k} - \alpha^{(k)}) u \frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.19}$$

By substituting these terms back into Equation A.16, we obtain:

$$\frac{\partial R(\theta)}{\partial w^{(k)}} = \lambda \sum_{i=1}^{K} \Big[\alpha^{(i)} (\delta_{i,k} - \alpha^{(k)}) u \frac{\partial m^{(k)}}{\partial w^{(k)}} \|o - m^{(i)}\|^2 \tag{A.20}$$

$$+ 2\alpha^{(i)} \Big(\alpha^{(k)} \frac{\partial m^{(k)}}{\partial w^{(k)}} + \sum_{z=1}^{K} \alpha^{(z)} (\delta_{z,k} - \alpha^{(k)}) \beta^{(z)} \frac{\partial m^{(k)}}{\partial w^{(k)}} \Big) \Big] - \lambda 2 \alpha^{(k)} \frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.21}$$

Which simplifies to:

$$\frac{\partial R(\theta)}{\partial w^{(k)}} = \lambda \sum_{i=1}^{K} \alpha^{(i)} \frac{\partial m^{(k)}}{\partial w^{(k)}} \Big[(\delta_{i,k} - \alpha^{(k)}) u \|o - m^{(i)}\|^2 \tag{A.22}$$

$$+ 2 \Big(\alpha^{(k)} + \sum_{z=1}^{K} \alpha^{(z)} (\delta_{z,k} - \alpha^{(k)}) \beta^{(z)} \Big) \Big] - \lambda 2 \alpha^{(k)} \frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.23}$$

Consider the case where the attention is evenly distributed across all memcells:

$$\alpha^{(k)} = \alpha^{(i)} = \frac{1}{K} \tag{A.24}$$

Note that this also implies that:

$$\beta^{(k)} = \beta^{(i)} \tag{A.25}$$

Substituting this back into Equation A.23, we get:

$$\frac{\partial R(\theta)}{\partial w^{(k)}} = \lambda \sum_{i=1}^{K} \frac{1}{K} \frac{\partial m^{(k)}}{\partial w^{(k)}} \Big[(\delta_{i,k} - \frac{1}{K}) u \| o - m^{(i)} \|^2 \tag{A.26}$$

$$+ 2\Big(\frac{1}{K} + \sum_{z=1}^{K} \frac{1}{K} (\delta_{z,k} - \frac{1}{K}) \beta^{(z)} \Big) \Big] - \lambda 2 \frac{1}{K} \frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.27}$$

which simplifies to:

$$\frac{\partial R(\theta)}{\partial w^{(k)}} = \frac{\lambda}{K} \frac{\partial m^{(k)}}{\partial w^{(k)}} \sum_{i=1}^{K} \Big[(\delta_{i,k} - \frac{1}{K}) u \| o - m^{(i)} \|^2 \tag{A.28}$$

$$+ 2\Big(\frac{1}{K} + \frac{1}{K} \sum_{z=1}^{K} (\delta_{z,k} - \frac{1}{K}) \beta^{(z)} \Big) \Big] - \frac{2\lambda}{K} \frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.29}$$

Since $\beta^{(k)} = \beta^{(z)}$, we can further simplify the second term in the inner sum to get:

$$\frac{\partial R(\theta)}{\partial w^{(k)}} = \frac{\lambda}{K} \frac{\partial m^{(k)}}{\partial w^{(k)}} \sum_{i=1}^{K} \Big[(\delta_{i,k} - \frac{1}{K}) u \| o - m^{(i)} \|^2 \tag{A.30}$$

$$+ 2\Big(\frac{1}{K} + \frac{1}{K} \beta^{(k)} \sum_{z=1}^{K} (\delta_{z,k} - \frac{1}{K}) \Big) \Big] - \frac{2\lambda}{K} \frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.31}$$

To simplify this term further, we make the observation that:

$$\sum_{z=1}^{K} (\delta_{z,k} - \frac{1}{K}) = 0 \tag{A.32}$$

This then reduces Equation A.31 to:

$$\frac{\partial R(\theta)}{\partial w^{(k)}} = \frac{\lambda}{K} \frac{\partial m^{(k)}}{\partial w^{(k)}} \sum_{i=1}^{K} \Big[(\delta_{i,k} - \frac{1}{K}) u \| o - m^{(i)} \|^2 + \frac{2}{K} \Big] - \frac{2\lambda}{K} \frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.33}$$

$$= \frac{\lambda}{K} \frac{\partial m^{(k)}}{\partial w^{(k)}} \sum_{i=1}^{K} \Big[(\delta_{i,k} - \frac{1}{K}) u \| o - m^{(i)} \|^2 \Big] + \frac{2\lambda}{K} \frac{\partial m^{(k)}}{\partial w^{(k)}} - \frac{2\lambda}{K} \frac{\partial m^{(k)}}{\partial w^{(k)}} \tag{A.34}$$

$$= \frac{\lambda}{K} u \frac{\partial m^{(k)}}{\partial w^{(k)}} \sum_{i=1}^{K} \Big[(\delta_{i,k} - \frac{1}{K}) \| o - m^{(i)} \|^2 \Big] \tag{A.35}$$

which gives the final simplified form.

Appendix B

Language Modeling Experiments

B.1 Complete Table of Results for PTB

Model	Hidd.	Param.	Anneal	Dropout	ITL	Train	Valid	Eval
KN-5 (Mikolov et. al 2012)	NA	NA	NA	NA	NA	NA	148	141
RNN (Mikolov et. al 2012)	100	NA	NA	NA	NA	NA	150	142
RNN + LDA (Mikolov et. al 2012)	100	NA	NA	NA	NA	NA	132	126
TopicRNN (Dieng et. al 2017)	100	NA	NA	NA	NA	NA	129	122
TopicGRU (Dieng et. al 2017)	100	NA	NA	NA	NA	NA	118	112
TopicLSTM (Dieng et. al 2017)	100	NA	NA	NA	NA	NA	126	118
TopicRNN (Dieng et. al 2017)	300	NA	NA	NA	NA	NA	118	112
TopicGRU (Dieng et. al 2017)	300	NA	NA	NA	NA	NA	100	97
TopicLSTM (Dieng et. al 2017)	300	NA	NA	NA	NA	NA	104	100
RNN	125	2.5m	NA		NA	85	154	143
RNN + Dropout	125	-	NA	✓	NA	102	138	128
GRU	125	2.6m	NA		NA	76	135	127
GRU + Dropout	125	-	NA	✓	NA	88	116	107
LSTM	125	2.6m	NA		NA	97	165	146
LSTM + Dropout	125	-	NA	✓	NA	98	152	134
Large-RNN + Dropout	575	16m	NA	✓	NA	81	146	139
Large-GRU + Dropout	575	18m	NA	✓	NA	67	115	114
Large-LSTM + Dropout	575	20m	NA	✓	NA	65	127	117
AMN	100	2.3m				82	144	135
AMN + Anneal	100	-	✓			90	148	139
AMN + Drop-Contr	100	-		✓		73	123	113
AMN + Drop-Mem	100	-		✓		73	108	103
AMN + Drop-All	100	-		✓		77	110	101
AMN + Implicit	100	-			✓	77	141	135
AMN + Drop-Mem + Implicit	100	-		✓	✓	67	104	97
AMN + Anneal + Drop-Mem + Implicit	100	-	✓	✓	✓	70	103	**95**
Large-AMN + Anneal + Drop-Mem	500	19m	✓	✓		64	102	**96**
Large-AMN + Anneal + Drop-Mem + Implicit	500	-	✓	✓	✓	55	98	**91**

Table B.1 Complete table of results for Penn Tree Bank models. The first block contain results from other works on neural topic-models. The second block contain results from baseline models. The third block contain results from AMN models.

B.2 PTB Learning Curves

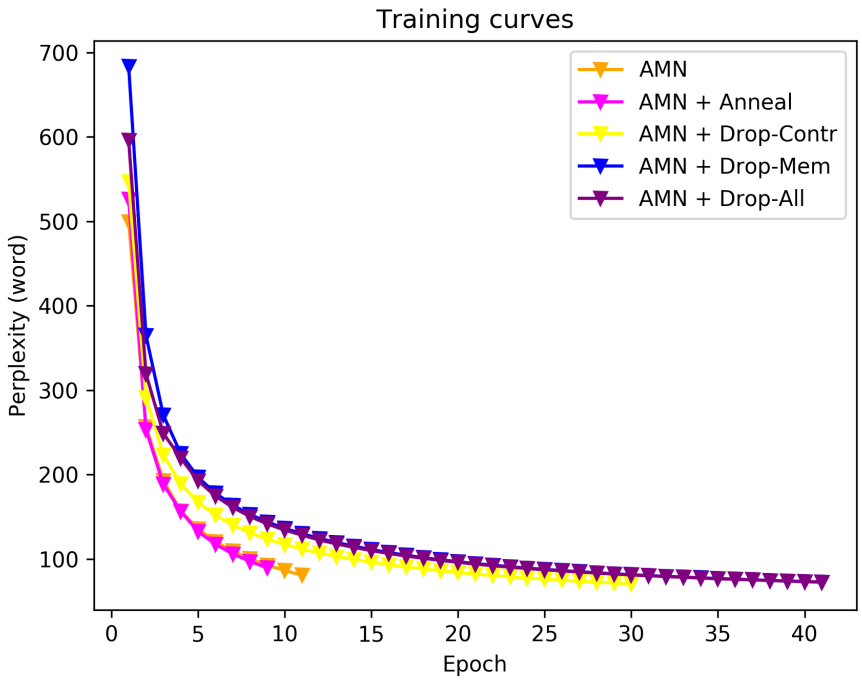


Fig. B.1 Training curves for small AMN models.

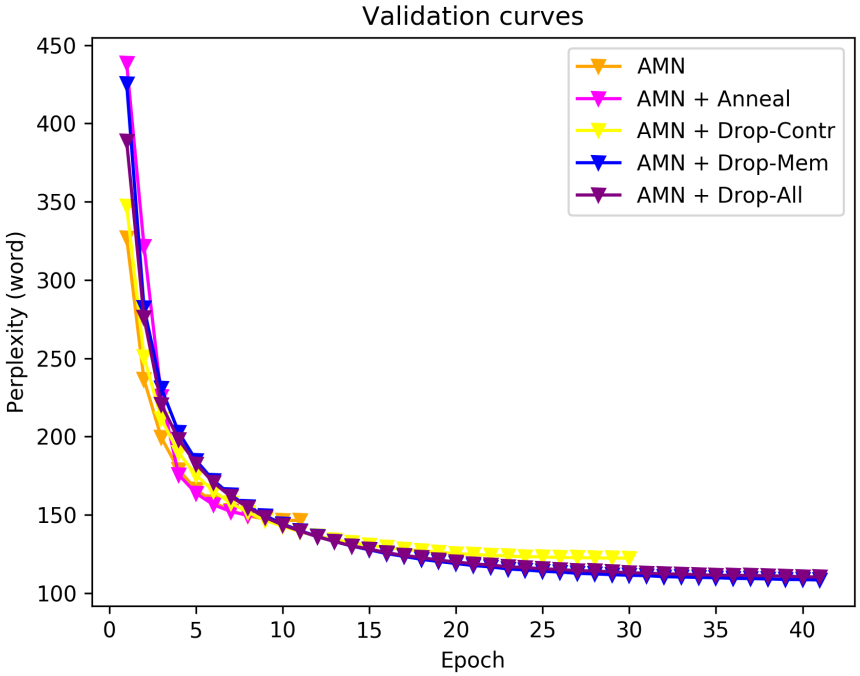


Fig. B.2 Validation curves for small AMN models.