Machine Learning Software Frameworks

Abhijit Singh

s1788323

Master of Science

Artificial Intelligence

School of Informatics

University of Edinburgh

2018

Abstract

Deep learning is being used to make breakthroughs in many areas of research. There are many new deep learning frameworks available, but we do not yet know their comparative strengths and weaknesses, especially when using complicated models. This project aims to fill this gap in the literature. We state the reasons why these frameworks are important, and give a brief review of some of the popular frameworks. We implement a recent state-of-the-art neural language model in Keras, with a Tensorflow back-end. We also outline why this model is ideal for our project, and test its efficacy. We describe some of the concepts that are used, and subjectively document the strengths and limitations of Keras. A critical analysis of this project and possible extensions of this study conclude this thesis.

Acknowledgements

I sincerely thank my supervisors, Ben and Tania, for their insightful guidance during the course of this project. I would also like to thank my friends working on the same topic, Cong, Zihan and Xin, for the interesting discussions that we had while working together.

Last but not least, I wish to thank my family and friends, who kept motivating me throughout this period and ensured that I maintained a healthy work-life balance.

Declaration

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

(*Abhijit Singh*
s1788323)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last two decades there has been an exponential increase in the amount of raw information available. Until a little while back, we did not know how to analyze and draw conclusions from this vast amount of data in a reasonable amount of time. For example, in the field of space exploration, there are many observatories that collect data from space in the form of raw images or measurements made by sophisticated sensors. Then scientists spend an enormous amount of time to manually analyze this data and make meaningful conclusions. Similarly in medical diagnosis, doctors or lab technicians spend a lot of time going over the scans of individual patients. This extra time and effort that is spent on analysis can be saved if the process is automated. This is where machine learning becomes extremely useful.

In situations where we have a lot of data to process, neural networks and specifically deep learning models are exceedingly useful tools. The main advantage that deep learning models have over other models is that they can create more representations of the input data over their successive layers, which help them do a better job at tasks like classification, amongst others. We have seen machine learning technology in various fields now, such as face recognition softwares in cameras, web searching algorithms, automated chat-bots, recommendation systems on e-commerce and video streaming sites, and even medical diagnosis (LeCun et al., 2015). The applications of machine learning, and particularly deep learning, are only going to increase in the future.

Back-propagation (Rumelhart et al., 1986) is a key component of a deep learning model. It revolutionized the field of machine learning, and it is this step that makes neural networks so powerful. However, computing the gradients that need to be back-propagated through a network is quite tricky, especially for complicated deep learning models, and immensely time-consuming to do analytically. Hence it was necessary to have tools which would make these tasks easier, and this is why machine learning software frameworks were developed. Each framework has its own method of automatically computing any gradients that might be required. Nowadays a researcher does not need to spend any time on checking if the intricate details of their model are faultless, they can use one of the numerous frameworks available and quickly build many different models and test them.

This naturally makes us consider which framework should be used, out of the myriad options available today. At the moment, there is no definitive answer to this question. Since this is a new field with a lot of active development ongoing, there have been many significant changes in the last few years. This project compares some of the popular and most widely used frameworks on a neural language modeling task, and attempts to gain a better understanding of the pros and cons of these frameworks. This work should be beneficial to all researchers who want to know which framework is ideal for their specific task, and what hurdles they might face during the developmental phase. (Singh, 2018)

The main language modeling architecture that we implement in this project was introduced recently by Yang et al. (2017). This was chosen specifically because it is an interesting solution to a crucial bottleneck in the language modeling domain. We wanted to test if this solution provided a significant boost to the performance, and what trade-offs it had in terms of time required to train the model. Since it is a novel concept, there were no existing functions which could be used off the shelf to implement it, which meant that we had to write the code ourself. This served the dual purpose of testing the efficacy of the proposed architecture while getting a better feel and understanding of the advantages and shortcomings of our framework. These have been subjectively documented, and will help researchers gain an insight into the kind of roadblocks they may face while using the framework.

The next chapter gives a brief overview of some of the popular deep learning frameworks and the current literature comparing them. It also gives a little background about natural language modeling. Chapter 3 gives a brief description of the concepts being used in the language models. Chapter 4 is the main experiments section. It contains details about each of the architectures that were tested, followed by their experimental results and analysis. Chapter 5 contains the subjective documentation of the ease of using Keras (the framework we use), and its limitations. Chapter 6 concludes this thesis, and suggests extensions of this work.

Chapter 2

Background and Literature Review

2.1 Brief review of Deep Learning Frameworks

One of the earliest open source and comprehensive machine learning frameworks was Torch (Collobert et al., 2002). It was first released in 2002, and it provides data structures for many useful components such as multi-dimensional tensors. It is a script language based on the Lua programming language, with an underlying implementation in C. When it was initially released it was used by a significant number of researchers, but many have now switched to the newer frameworks.

Theano (Bergstra et al., 2010) was the first significant competitor of Torch. It was developed at the University of Montreal around 2008. The coding is done in Python, and it combines the convenience of using the NumPy syntax with the speed of optimized native machine language. It provided massive speed ups on processing times for large neural networks, compared to the other frameworks available at its time of release. Theano computes symbolic differentiation of complex expressions automatically and minimizes the memory usage during processing, amongst a host of other features. This is done by storing the mathematical expressions defined by the user as a static graph of variables and operations, that is pruned and optimized at compilation time. Active development of Theano was stopped in September 2017, due to competing frameworks launched by many big technology companies like Google, Facebook and Microsoft.

Convolutional Architecture for Fast Feature Embedding (CAFFE) (Jia et al., 2014) was developed at UC Berkeley in 2014. As the name suggests, it primarily focuses on Convolutional Neural Networks (CNN or conv-net). In recent releases the developers added some support for Recurrent Neural Networks (RNN) as well, but most researchers still avoid using CAFFE when dealing with text or time series data. It is written in C++ with a Python interface.

Like Theano, Tensorflow (Abadi et al., 2016) also uses a static computation graph. Its flexible architecture allows for easy deployment of computation across various platforms like desktops, GPUs and mobile devices. It was developed by the Google Brain team for internal use, but was finally released as an open source library in November 2015. It is built keeping extensibility in mind, and Abadi et al. (2016) claim that it scales well to multiple GPUs.

MXNet (Chen et al., 2015) is another deep learning framework that is backed by a big commercial company, in this case, Amazon. Its features are quite similar to Tensorflow, but the low level implementation is obviously different. It handles the embeddings of both symbolic expression and tensor operation in a unified fashion, and supports many different programming languages for its front-end, such as Python, R, Scala and Perl.

PyTorch was released in October 2016, by Facebook's Artificial Intelligence (AI) research group. It can be viewed as a Python front-end for the Torch engine. Like the other frameworks, it allows users to define mathematical functions and computes gradients automatically. However, PyTorch uses a dynamic computation graph, which makes it an easy to use and efficient framework for handling text or time series data. Due to this, many researchers have started using PyTorch even though it is a new entrant to the scene.

Keras (Chollet et al., 2015) is an open source library written in Python. It was designed by an engineer at Google to enable fast experimentation with deep neural networks, and it focuses on being user-friendly, modular and extensible. It is basically a high level interface which abstracts the back-end, making it quite easy for beginners. It supports various back-ends like Tensorflow, Theano, MXNet and Microsoft Cognitive Toolkit (CNTK) (Seide & Agarwal, 2016). Its ease of use and support for many popular back-ends have resulted in Keras having the fastest growing user base, and being second only to Tensorflow in overall popularity, according to many surveys and reports. Figure 2.1 (taken from (Chollet et al., 2015)) compares the number of mentions that each framework received in the papers submitted to arXiv in October 2017, which validates this point.

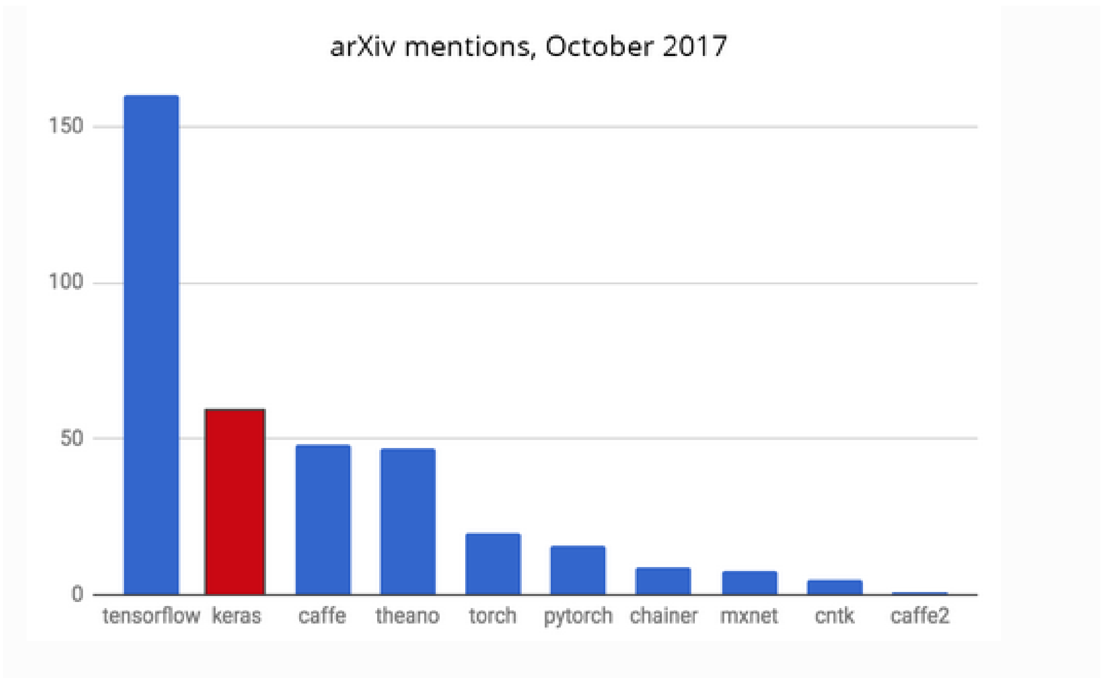


Figure 2.1: Comparison of mentions that each framework received, in papers submitted to arXiv in October 2017. Figure from updated version of Chollet et al. (2015).

With so many competitive frameworks being available, one would expect that many people have carried out comparative studies. However, this is not the case. There is a dearth of such studies in the current literature, and this is one of the key reasons for undertaking this project. Of the few studies that have been done, Chintala (2015) benchmarked some of the public open-source implementations of CNNs. This study was limited because it only compared processing speeds and did not talk about performance of the models. Moreover, it did not use any dropout or softmax layers in any of its implementations, and these are quite often the main source of longer processing times. Bahrampour et al. (2015) compared CAFFE, Neon, Tensorflow, Theano, and Torch on metrics such as extensibility, hardware utilization and speed. They used stacked auto-encoders and CNNs on the MNIST dataset (LeCun, 1998) and the ImageNet dataset (Deng et al., 2009), and a Long Short-Term Memory (LSTM) network on the IMDB review dataset (Maas et al., 2011). However, they also did not compare performance of the different models and neither did they have a documentation of building non-standard deep learning models. Similarly, Shi et al. (2016) compared CAFFE, CNTK, MXNet, Tensorflow and Torch in terms of time taken for processing with different mini batch sizes, on the same datasets as used by Bahrampour et al. (2015). While Bahrampour et al. (2015) only tested on one type of GPU, Shi et al. (2016) tested their architectures on three different architectures of GPUs.

There are a few key points to note after going through this brief review. Firstly, there is a vacuum of studies which compare the latest frameworks with each other. Secondly, the existing studies implement only a few standard architectures on standard datasets, for each framework. There has been no attempt to compare complicated architectures on a different dataset. Lastly, there has not been any mention of challenges faced during model building in each of the frameworks. This is a significant concern for most researchers, as a lot of time that is spent on coming up with hacks can be saved, if building models is an easy and smooth process (Singh, 2018).

2.2 Background of Natural Language Modeling

The models we implement in this project are neural language models. A language model can be seen as a probability distribution over sequences of words. They can be broadly divided into two categories: count-based language models and continuous-space language models.

2.2.1 Count-based language models

The simplest count-based language model is the unigram model. In this model, it is assumed that the probability of occurrence of each word is independent of all other words. Mathematically speaking,

$$P(t_1 t_2 t_3) = P(t_1)P(t_2)P(t_3),$$

where t_i denotes a word in the corpus or document. This means that what is essentially done is that the frequency of each word t_i in the document is noted, and that is divided by the total number of words in the document to come up with the probability of that word, $P(t_i)$. This also ensures that the probability distribution of all the words sums up to 1.

The assumption that all words are independent of each other is a rather strong one, which leads to a poorer performance compared to other more sophisticated methods. For example, we know that if we see the word "blue" in the middle of a sentence, it is likely to be followed by a word which denotes an object, for example a "blue toy" or a "blue car" or something similar. It is clear that having knowledge of the preceding word affects the probability of seeing a certain word after it. This necessitated the development of models which could model these dependencies.

N-gram language models solved this issue. In these models, we define a term called context length: the total number of words that depend on each other. If the context length is n, then the probability of the n^{th} word depends on the n-1 words preceding it. If the context length is 2, it is called a bigram model. For a bigram model, the probability distribution over m words would look like this:

$$P(t_1, t_2, ...t_m) = \prod_{i=1}^{m} P(t_i|t_{i-1})$$

This same concept is extended for n-grams. Increasing the value of n does provide better modeling, but it also has its drawbacks. The curse of dimensionality is especially prevalent in count-based language modeling. As the size of training corpus increases, so does the number of unique words found in it (also called vocabulary). This causes the number of possible sequences of words to grow exponentially. To get a rough idea, if the vocabulary has 10,000 or 10^4 words, and we have a sequence of m words, then the number of possible sequences becomes 10^{4m}. This scales very poorly (exponentially) with increase in m, causing a data sparsity problem (having very high dimensional matrices, whose majority of elements have the value 0).

N-gram models have other drawbacks as well. They rely on exact patterns (word sequence matching), and thus are not based on knowledge of linguistics. For example, a good language model should be able to recognize that a sequence like "the knife is in the kitchen" is semantically and syntactically similar to the sequence "the football is in my room". Such modeling abilities are absent in n-gram models (Bengio et al., 2003b). Another problem is that dependencies beyond the context length are completely ignored.

This is in conflict with the fact that humans can use longer contexts with great success. These reasons prompted the development of continuous-space language models.

2.2.2 Continuous-space language models

Continuous-space language models are also called neural language models. They make use of neural networks and use continuous representations or embeddings of words to make predictions.

Prior to the development of word embeddings, words being taken as input were converted into one-hot encodings. This meant that one column was allocated for each word in the vocabulary, and the presence of that word was given the value 1 and absence of words was given the value 0. These values had no inherent relationship to one another (Mikolov et al., 2013).

Word embeddings provided a distributed representation of high dimensional, real-valued vectors, where words with similar meanings have similar vectors. They try to capture the similarities and differences between words, and this can be well understood from Figure 2.2

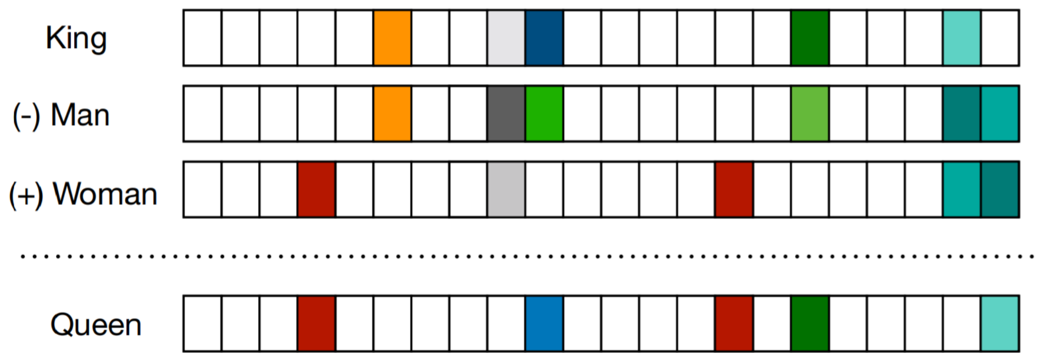


Figure 2.2: Word vectors represented by a D-dimensional vector where D « V, where V is the size of Vocabulary. Figure from (Young et al., 2017).

This figure is a demonstration of how the real-valued vectors in a word embedding matrix are constructed by a neural network. It is clear that certain dimensions capture the gender of a word, while others capture aspects like whether the word refers to a person, object, and so on. As a result of similar words being grouped together, when parameters are adjusted for a particular word or sequence, the improvements also apply to similar words and sequences. This leads to a better generalization performance than other models like the n-gram (Mikolov et al., 2013). Further, word embeddings solve the problems that occur due to high dimensional sparse matrices, as in the case of using one-hot encoding for each word in the vocabulary.

Neural language models can be further classified into two main categories:

- Feed forward neural network based models
- Recurrent neural network based models

Since we use a special type of Recurrent Neural Network (RNN) in this project, the next chapter has one section dedicated to discussing it.

Chapter 3

Main Concepts and Evaluation Criteria

The first major development in the field of neural language models involved feed forward neural networks, proposed by Bengio et al. (2003). However, all of the recent advancements have involved Recurrent Neural Networks (RNN) based models, after the pioneering work of Mikolov et al. (2010). In this project, we shall focus on RNN based models only. The next section discusses why RNN based models work well on language modeling tasks.

3.1 Long-Short Term Memory

All models in this project employ a special type of RNN, a Long-Short Term Memory (LSTM) network. RNNs were developed to address the shortcomings of the feed-forward neural language model approach in Bengio et al. (2003); in particular the fixed and small window size for the input. A conspicuous limitation of vanilla neural networks (and conv-nets as well) is that they are too constrained. They accept a fixed-sized vector as input (perhaps images or videos) and produce a fixed-sized vector as output (usually the probabilities of different classes, when the task is multi-class classification). Moreover, this mapping is performed using a fixed number of computational steps (for example, the number of layers in the model). The main reason why RNNs are effective is that they allow us to operate over sequences of vectors: sequences in the input, the output, or in the most general case, both input and output. Fundamentally, RNNs are neural networks that take both an extrinsic input and its own previous state as a combined input. This way the network is able to consider past states in addition to the current state; it has a memory (see Figure 3.1). As a result, RNNs do not use a limited size context; information can cycle inside these networks for an arbitrarily long time, although long-term dependencies are not learned easily (Mikolov et al., 2010a).

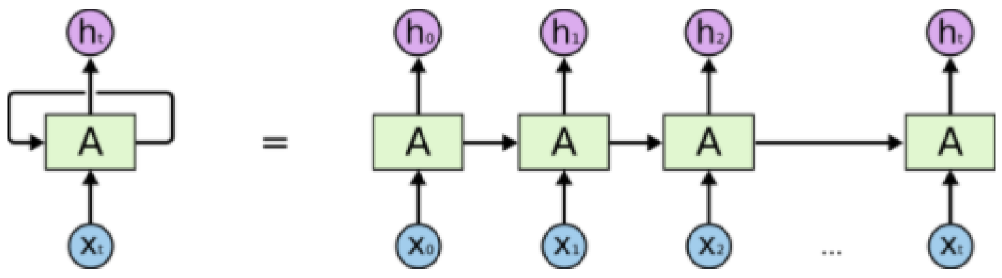


Figure 3.1: Diagram of an RNN. Right hand side is 'unrolled', with sequential timesteps from left to right. x are inputs, A is a hidden layer and h are outputs. Figure from Olah (2015).

For a simple RNN (specifically an Elman network) there is an input layer x, a hidden layer s and an output layer y; this is the same as a typical single layer neural network. Critically, the hidden, or context layer, s is preserved for one timestep, such that at time t, s_{t-1} is available for computation. Input, hidden and output layers are computed by

$$x(t) = w(t) + s(t-1) \tag{3.1}$$

$$s_j(t) = f\left(\sum_i x_i(t) u_{ji}\right) \tag{3.2}$$

$$y_k(t) = g\left(\sum_j s_j(t) v_{kj}\right) \tag{3.3}$$

where $f(z)$ is the sigmoid function and $g(z)$ is the softmax function.

LSTMs are a special kind of RNN, which were designed to solve the long term dependency problem. As they are capable of learning long term dependencies, they work very well on a large variety of problems, and hence are widely used (Olah, 2015). All RNNs are in the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module has a simple structure, such as a single tanh layer. LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four layers, interacting in a special way. RNNs are architecturally forced to consider the input of the previous state. LSTMs on the other hand, make this connection a learned gate function, meaning that the network learns how to weight the previous states given the current context. LSTMs also make the input and output dependent on learned gate functions, allowing the network to optionally and flexibly consider input, output and context (previous state) dependent on the current input (Figure 3.2 shows this architecture). This is computed by

$$f_t = \sigma(W_f[s_{t-1}, x_t] + b_f) \tag{3.4}$$

$$i_t = \sigma(W_i[s_{t-1}, x_t] + b_i) \tag{3.5}$$

$$\widetilde{C}_t = tanh(W_C[s_{t-1}, x_t] + b_C) \tag{3.6}$$

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t \tag{3.7}$$

$$o_t = \sigma(W_o[s_{t-1}, x_t] + b_o) \tag{3.8}$$

$$s_t = o_t * tanh(C_t) \tag{3.9}$$

where f_t is the forget gate, i_t is the input gate, o_t is the output gate, $\widetilde{C}_t$ are candidate values to be propagated dependent on the input gate and C_t are the updated cell values.

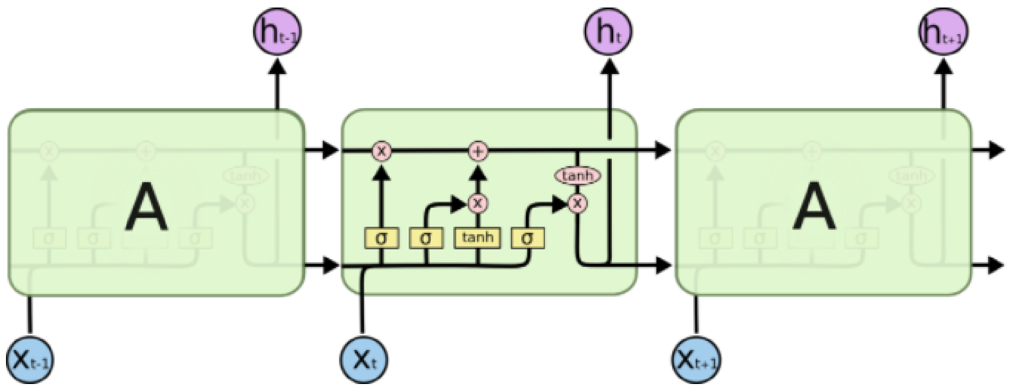


Figure 3.2: LSTM architecture diagram showing the gating functions (the three sigmoid functions σ). Figure from Olah (2015).

3.2 Early Stopping

Early stopping is an important technique that should be used in all machine learning algorithms. The dataset is usually split into three parts: training set, validation set, and test set. As the name suggests, only the training set is used to train the machine learning model. Training here refers to the fact that only this data is fed as input to the model. The model then adjusts and learns its parameters after going through this chunk of data iteratively.

At the same time, we also want to know how well the model is learning. This is done by checking its performance on the unseen validation set. We make decisions about the choice of hyper-parameters of the model based on its performance on the validation set. If a model is built correctly, then there is an optimum point where the model performs at its best. We obviously want to use the parameter settings at that specific point. This is not possible without early stopping.

In early stopping, we save the model parameters after the end of every epoch. These settings are only overwritten if at the end of a subsequent epoch the model has a better performance on the validation set. Just after the optimum point in its training phase, the model stops learning essential features, and starts to over-fit to the training data. We want to save the model settings at the optimum point, and save computation cost by stopping the training soon after the optimum point. This is implemented by setting a 'patience' value. Patience is the number of epochs after the best validation score, when the model stops training. Figure 3.3 explains the concept of early stopping graphically.

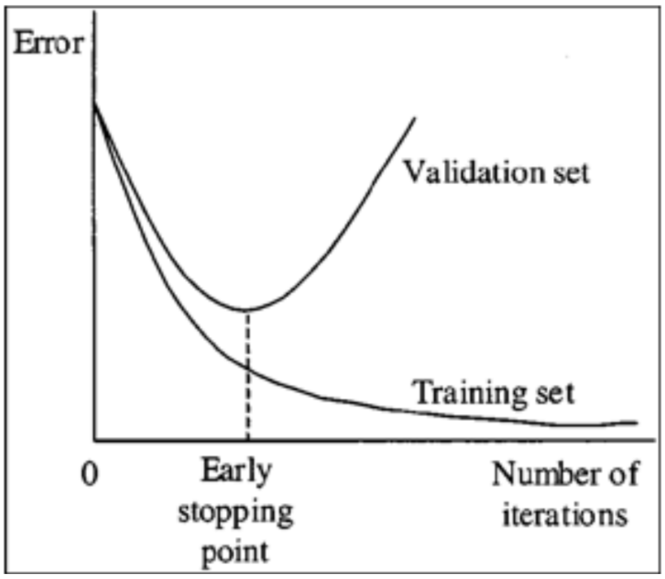


Figure 3.3: When error on the validation set starts to increase after the optimum point, we stop training the model.

3.3 Evaluation criteria

Four students worked on this same topic. Each student picked one framework in which they built baseline models and implemented the "Mixture of Softmaxes" (MoS) model as described in Yang et al. (2017), on the Penn Tree Bank (PTB) dataset (Marcus et al., 1993). The PTB dataset has 929k training words, 73k validation words, and 82k test words. We picked Keras, with a Tensorflow back-end. To enable a comparative analysis between the four frameworks, we agreed to certain evaluation metrics which we would all report in our thesis. We want to compare the deep learning frameworks on the following metrics:

1. Time taken to load the data into memory and completing the pre-processing. Here we will also see whether always streaming the data is better than caching it to the disk.

2. Time taken for forward and backward pass of data. This will include:

    - Total time taken to make a prediction - this is important because for many applications the prediction time needs to be extremely quick (for example, choice of online ad to be displayed).
    - Total training time per epoch - this is important because it is the most computationally expensive part of building a machine learning model. Apart from compute time, we can also calculate what the effective monetary cost is.

3. How well does the framework scale with the use of multiple GPUs. This metric is especially important for training complex models with vast amounts of data.

4. Are there any significant differences in results (perplexity of the neural language model) obtained on the different frameworks.

5. Ease of use. This is subjective, and will include:

    - How many functions required in the models we build are available as built in functions, and how many do we have to code on our own.
    - How much time did we spend coding custom functions, and how helpful were the error messages (and material available online) during debugging.

Chapter 4

Experiments and Results

4.1 First baseline model: Single LSTM layer

To start the first experimental phase of the project, we built a very simple language model. The pre-processed data was used as the input to the Embedding layer. The outputs of the Embedding layer were fed to an LSTM layer, whose outputs were processed by a Softmax layer to get a probability distribution over the vocabulary (our vocabulary size was fixed at 10,000). The final output from the Softmax layer was used to predict the next word in the sequence. Figure 4.1 shows a block diagram representation of this model architecture.

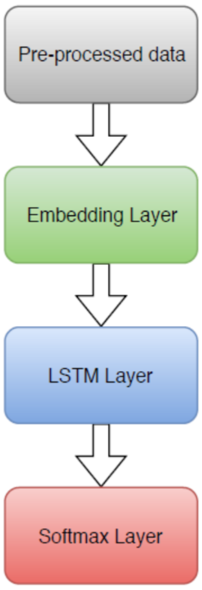


Figure 4.1: Block diagram representation of the single layer LSTM model.

The objectives of building this basic model were threefold:

- Ensure the pre-processing was being done as required.
- Build a well modularized pipeline, which would only need a modification of the model architecture to run various different models.
- Check the limits of a very simple baseline

The criteria used to judge the performance of our language models is called perplexity. It is a measurement of how well a probability distribution predicts a sample. A lower perplexity indicates a better model.

To ensure that random initializations do not influence the performance, every experiment was run 5 times, and the mean and standard deviations are reported in the results. All experiments also used Early Stopping, with a patience of 5 epochs. The following sub-sections summarize the key results from experiments performed on this baseline model.

4.1.1 Learning Rate

One of the most crucial hyper-parameters of a model is its learning rate. If set too low, the model takes much longer to train and converge at an optimum point than necessary. This costs compute time, which is at a premium in most cases. If set too high, the model may miss the optimum point because its updates are too big. Thus the first set of experiments aimed to find a good learning rate for our model. The other model settings were kept fixed, and they were: context length = 35, batch size = 20, number of hidden units in the LSTM layer = 650, optimizer = Adam.

These settings were chosen because they were used in the model made by Zaremba et al. (2014), which is our second baseline model. In these experiments, we increased the learning rate by a factor of 1 on the logarithmic scale. Table 4.1 compares the perplexity scores on the validation set when using different learning rates.

Table 4.1: Perplexity scores of learning rate (LR) experiments

Model	Perplexity	Time per epoch (s)	Epochs	Total time (min)
LR 0.0001	176 ± 3	140	20	47
LR 0.001	143 ± 2	138	10	23
LR 0.01	204 ± 4	138	9	21

Learning rates higher than 0.01 were also tried, but they performed very poorly (perplexity on validation set > 3000). These results show that when using Adam (Kingma & Ba, 2014) as the optimizer, a learning rate of 0.001 works best with this model.

4.1.2 Context Length

After fixing our learning rate at 0.001, we proceeded to test what different context lengths could this model process. At this point, we also use a learning rate schedule. A learning rate schedule is a function which monitors a specific quantity (in this case, the loss on the validation set), and adjusts the learning rate when the monitored quantity stops moving in the direction we want it to (for example: validation loss stops decreasing, or validation accuracy stops increasing. In language models, we are concerned with validation loss). We used a learning rate schedule which decreased the learning rate by a factor of 10, when the validation loss stopped decreasing. We tested context lengths (CL) from 15 to 55, at intervals of 10. The other hyper-parameters were same as in the previous section. Table 4.2 summarizes the results (perplexity on validation set).

Table 4.2: Perplexity scores of context length (CL) experiments

Model	Perplexity	Time per epoch (s)	Epochs	Total time (min)
CL 15	142 ± 2	186	8	25
CL 25	141 ± 1	157	21	56
CL 35	139 ± 1	150	10	25
CL 45	692 ± 126	147	9	22
CL 55	626 ± 60	144	15	36

These results suggest that a single LSTM layer with 650 hidden units does not have the capacity to process long context lengths. With this observation, we find one of the limitations of this very simple baseline model.

We also conducted a series of experiments which validated our hypothesis that using smaller batch sizes would improve the model's performance. This is because the updates to the parameters are made at the end of every batch, and having large batch sizes would mean that the chances of some gradients canceling each other out if they are in opposing directions increases, or the chances of some gradients dominating the others and moving the update away from the optimal region increases.

Using a batch size of 5 instead of 20, the perplexity on the validation set improved to 136 ± 1. This improvement comes at the cost of longer computation time. Using a batch size of 5 increased compute time by 10 minutes (or 40%), which is not too much when looked at in isolation, but is likely to be higher for more complex model architectures. For batch sizes of 50 and above, the perplexity was greater than 500.

Since this single LSTM layer model is a very simple model, we did not do more detailed analysis or experiments on it. The experiments done served the purpose of validating our pre-processing and data generator modules. Now we could be sure that our pipeline functioned well, and without any bugs. They also gave us an idea of what we could expect by changing the hyper-parameters in more complex models.

4.2 Second baseline model: Two LSTM layers with dropout

Our second baseline model follows the work of Zaremba et al. (2014). They presented a scheme where Dropout (Srivastava et al., 2014) could be used with LSTMs. Until then, using dropout naively in LSTMs resulted in very poor performance. Their method achieved a state-of-the-art performance on the Penn Tree Bank (PTB) dataset at that time. It consists of applying dropout on only the non-recurrent connections of the LSTMs, while not interfering with the recurrent connections.

As before, the pre-processed data is the input for the Embedding layer. The output of the Embedding layer is the input for the first LSTM layer, with dropout applied on these connections. The outputs of the first LSTM layer are the inputs for the second

LSTM layer, with dropout again applied on the connections. The outputs of the second LSTM layer are subjected to another round of dropout, and they are fed into the Softmax layer, which computes the probability distribution over the vocabulary. The output from the Softmax layer is then used to predict the next word in the sequence. Figure 4.2 shows a block diagram representation of this model architecture.

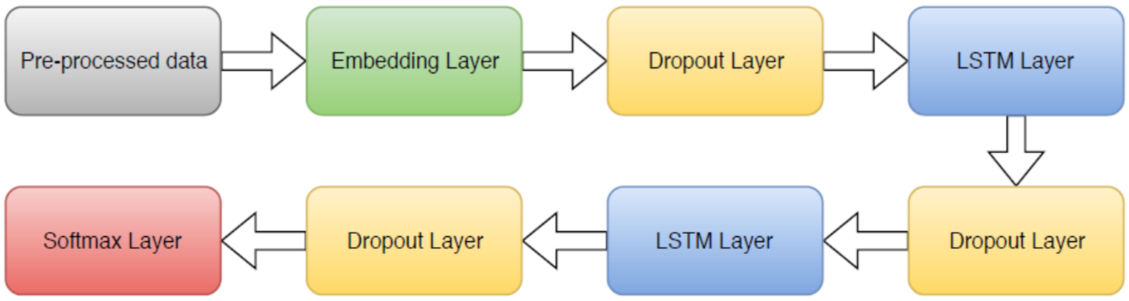


Figure 4.2: Block diagram representation of the second baseline model.

As mentioned in the previous section, to ensure that random initializations did not influence the performance, every experiment was run 5 times, and the mean and standard deviations are reported in the results. All experiments also used Early Stopping, with a patience of 5 epochs. The following sub-sections summarize the key results from experiments performed on the second baseline model.

4.2.1 Dropout

Our first experiments in this phase of the project tested the effects of using dropout as mentioned in Zaremba et al. (2014). This meant that we compared two models: one which had dropout disabled, and another which had a dropout of 0.5 at every stage (as shown in Figure 4.2). All other model settings were kept fixed, and they were: context length = 35, batch size = 20, number of hidden units in each LSTM layer = 650. These settings were chosen to match those of Zaremba et al. (2014). The only difference with their model was the choice of optimizer. They employed an intricate and optimized learning rate schedule, while we chose Adam with a learning rate of 0.001. Table 4.3 shows the difference in their performance on the validation set.

Table 4.3: Perplexity scores of dropout experiments

Model	Perplexity	Time per epoch (s)	Epochs	Total time (min)
w/o dropout	158 ± 2	160	11	29
with dropout	142 ± 1	162	22	59

This result makes it clear that implementing dropout between non-recurrent connections improves the model, as stated in Zaremba et al. (2014). They achieved a perplexity of 115 when they used a small model with some tricks, and it took 1 hour to train on a GPU. Their larger model (the model specifications are mentioned in their paper), which trains for 24 hours on a GPU, achieved a state of the art performance at that time, with a perplexity of 82. We have used their basic model as our second baseline, and try to explore its limits of modeling capacity in the following experiments.

4.2.2 Learning Rate

Once again we performed experiments to choose an optimal learning rate. Noticing that Zaremba et al. (2014) used an intricate and optimized learning rate schedule, we too employed a learning rate schedule of our own, albeit a much simpler one. We used a learning rate schedule which decreased the learning rate by a factor of 10, when the validation loss stopped decreasing.

All model settings for these experiments were the same as in the previous section, with dropout of 0.5 included. The learning rates tested differed by a factor of 1 on the logarithmic scale. Table 4.4 compares the perplexity achieved on the validation set.

Table 4.4: Perplexity scores of learning rate (LR) experiments

Model	Perplexity	Time per epoch (s)	Epochs	Total time (min)
LR 0.0001	183 ± 6	165	20	55
LR 0.001	129 ± 1	163	26	71
LR 0.01	200 ± 8	161	20	54

These experiments confirm that even with this model architecture, when we use Adam as our optimizer the best learning rate is 0.001. In the future experiments, this setting is retained unless specified otherwise.

4.2.3 Context Length

After fixing our learning rate, we checked if this model architecture could process longer context lengths. We tried different context lengths, from 15 to 55, at intervals of 10. Table 4.5 summarizes the performance of the models on the validation set.

Table 4.5: Perplexity scores of context length (CL) experiments

Model	Perplexity	Time per epoch (s)	Epochs	Total time (min)
CL 15	132 ± 3	215	31	111
CL 25	131 ± 2	184	37	113
CL 35	129 ± 1	163	26	71
CL 45	640 ± 89	159	18	48
CL 55	684 ± 65	156	18	47

We observe that in the range 15 to 35 the performances are similar, while longer context lengths are processed as poorly by this model architecture as the first baseline model.

As mentioned in Section 4.1.2, having smaller batch sizes improved the model performance again. As this was a more complex model than the first baseline, we saw a bigger improvement in performance. Using a batch size of 5, the perplexity on the validation set was 115 ± 1 (for a context length of 35). However, the compute time increased by almost 100%, to 300s per epoch. This model took about 3 hours to train on a GPU. Larger batch sizes (50 and above) reduced compute time significantly (lesser than 130s per epoch), but they also performed very poorly on the core prediction task, having perplexities greater than 350 on the validation set.

We also ran some experiments to check if increasing the depth of the network improved model performance. This was accomplished by stacking more LSTM layers and Dropout layers (with dropout of 0.5) in the same manner as in Figure 4.2. We found that having more LSTM layers only led to over-fitting, and that the two layer LSTM model as proposed in Zaremba et al. (2014) was the best option for this language modeling task.

4.2.4 Choice of Optimizer

Our experiments until now were using Adam as the optimizer. However, reading through the literature and looking through the implementations of Yang et al. (2017) and Zaremba et al. (2014), we felt that using Stochastic Gradient Descent (SGD) could provide good results. Thus in this set of experiments, we compared the performances when using Adam, Root Mean Square Propagation (RMSProp) (Tieleman & Hinton, 2012) and SGD as optimizers. For Adam and RMSProp, the default values of learning rates were used (0.001). For SGD, we used a learning rate of 1. The other hyper-parameters were same in every model: context length = 35, batch size = 5, number of hidden units in each LSTM layer = 650, dropout probability = 0.5. Table 4.6 compares the results of each model on the validation set.

Table 4.6: Perplexity scores of different optimizers on the validation set

Model	Perplexity	Time per epoch (s)	Epochs	Total time (min)
Adam	115 ± 1	300	39	195
RMSProp	146 ± 3	290	31	150
SGD	107 ± 2	252	41	172

These results show that using SGD has clear benefits. Not only do the models perform better, but it also takes significantly lesser time per epoch while training. Thus we decided to explore it further. As SGD has different learning characteristics when compared to Adam, we had to tune the learning rate for SGD separately. We found that this model architecture performed best with a learning rate of 5. The training time per epoch remained the same, but the total training time dropped to roughly 140 minutes, while the perplexity on the validation set was 97 ± 1. We also tried to increase the depth of the network again, to test if more layers provided a better performance when SGD was used as the optimizer. However, the results were the same as before. A two LSTM layer model was found to be the most effective architecture.

4.2.5 Hidden Size

In the next set of experiments, we tested if increasing the number of hidden units in each LSTM layer (referred to as hidden size) improved the model's performance on the validation set. All the other hyper-parameters were the same as in the previous section (the optimizer used was SGD with a learning rate of 5). Table 4.7 summarizes the results of these experiments.

Table 4.7: Perplexity scores of hidden size (HS) experiments

Model	Perplexity	Time per epoch (s)	Epochs	Total time (min)
HS 500	99 ± 1	220	50	183
HS 650	97 ± 1	253	34	143
HS 800	95 ± 1	274	40	183
HS 900	98 ± 1	322	39	209
HS 1000	98 ± 1	362	34	205
HS 1200	97 ± 1	470	34	267
HS 1500	96 ± 1	640	39	416

Though the results are quite similar, having a hidden size of 800 is the optimal choice, not only because it has the best performance in terms of perplexity, but also because it does not take too long to train.

After these experiments, we also employed Random Search (Bergstra & Bengio, 2012) to find a combination of hyper-parameters which performed better than our best model. Hundreds of combinations were tested for five epochs each, and the promising combinations were allowed to run until convergence. However, we failed to find a better combination. Thus after optimizing the hyper-parameters of this model architecture, the ceiling of performance in terms of perplexity on the validation set is about 95. This is much higher than the perplexity of 115 reported by Zaremba et al. (2014) for their basic model, and a bit lower than the perplexity of 82 achieved by their large model, which takes 24 hours to train on a GPU. In comparison, our much simpler model takes only 3 hours to train on a GPU.

With this we come to the end of experiments done on the baseline models. We learnt that having smaller batch sizes improves performance, but at the cost of longer training times, and that these baseline models were incapable of handling large context lengths, which would have improved their time cost and possibly their performance on the validation set. We observed that using SGD with a well tuned learning rate was the best optimizer. We also found that having more stacked LSTMs gave us no added advantage, and this held true for large number of hidden units as well.

4.3 Mixture of Softmaxes language model

Recently, Yang et al. (2017) proved that language modeling can be seen as a matrix factorization problem. Further, they showed that the expressiveness of Softmax based models is limited by a *Softmax Bottleneck*. They said that as natural language is highly context dependent, it implied that Softmax with distributed word embeddings did not have the necessary capacity to model natural language. They proceeded to propose an effective method to address this problem, and they termed it *Mixture of Softmaxes* (MoS).

The traditional Softmax distribution operates on a context vector (or hidden state) $\mathbf{h}_c$ and a word embedding $\mathbf{w}_x$ to define the conditional distribution $P_\theta(x|c)$, where x is the next token (word) and c is the context. The model distribution is usually expressed as:

$$P_\theta(x|c) = \frac{\exp \mathbf{h}_c^\top \mathbf{w}_x}{\sum_{x'} \exp \mathbf{h}_c^\top \mathbf{w}_{x'}} \tag{4.1}$$

where $\mathbf{h}_c$ is a function of c, and $\mathbf{w}_x$ is a function of x. Both functions are parametrized by θ . The vectors $\mathbf{h}_c$ and $\mathbf{w}_x$ also have the same dimension d, and their dot product $\mathbf{h}_c^\top \mathbf{w}_x$ is called a *logit*.

In their solution to address the softmax bottleneck, Yang et al. (2017) introduce discrete latent variables into the language model, and then formulate the next-token probability distribution as a Mixture of Softmaxes (MoS). They claim that this model is more expressive, and their state of the art results validate these claims. MoS formulates the conditional distribution as

$$P_\theta(x|c) = \sum_{k=1}^{K} \pi_{c,k} \frac{\exp \mathbf{h}_{c,k}^\top \mathbf{w}_x}{\sum_{x'} \exp \mathbf{h}_{c,k}^\top \mathbf{w}_{x'}}; \quad s.t. \sum_{k=1}^{K} \pi_{c,k} = 1 \tag{4.2}$$

where $\pi_{c,k}$ is the *prior* or *mixture weight* of the k-th component, and $\mathbf{h}_{c,k}$ is the k-th context vector associated with the context c. Basically MoS computes K different softmax distributions, and uses a weighted average of them to get the probability distribution for the next token.

This model architecture is built by stacking the MoS structure on top of the existing language model. Figure 4.3 shows a high-level block diagram for the MoS model used in our work.

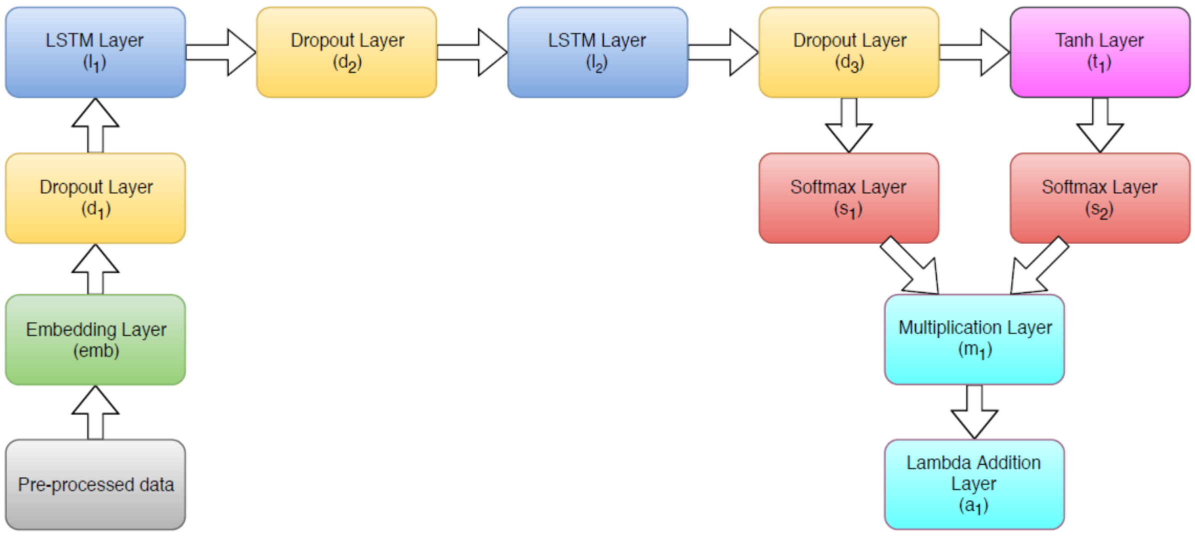


Figure 4.3: High-level block diagram representation of the MoS model.

By referring Figure 4.2, we can see that until block d_3 we have the same architecture as in the second baseline model. Then, instead of having a softmax layer which computes the probability distribution for the next token, we have more blocks which perform the extra processing required to implement the MoS model. The MoS model obtains a sequence of hidden states $(\mathbf{g}_1, \ldots, \mathbf{g}_T)$ from the last LSTM layer (with dropout applied). Then the prior and the context vector for context c_t are parametrized as

$$\pi_{c_t,k} = \frac{\exp \mathbf{w}_{\pi,k}^{\top}\mathbf{g}_t}{\sum_{k'=1}^{K} \exp \mathbf{w}_{\pi,k'}^{\top}\mathbf{g}_t} \quad \text{and} \quad \mathbf{h}_{c_t,k} = \tanh(\mathbf{W}_{h,k}\mathbf{g}_t) \tag{4.3}$$

where $\mathbf{w}_{\pi,k}$ and $\mathbf{W}_{h,k}$ are model parameters (In machine learning convention, small letters as variables indicate vectors and capital letters as variables indicate a matrix).

In Figure 4.3, block s_1 implements the computation of $\pi_{c_t,k}$ from Equation 4.3 (which ensures that the condition $\sum_{k=1}^{K} \pi_{c,k} = 1$ from Equation 4.2 is met), while block t_1 implements the computation of $\mathbf{h}_{c_t,k}$. Block s_2 computes $\frac{\exp \mathbf{h}_{c,k}^{\top}\mathbf{w}_x}{\sum_{x'} \exp \mathbf{h}_{c,k}^{\top}\mathbf{w}_{x'}}$ from Equation 4.2, and block m_1 multiplies the outputs of blocks s_1 and s_2. Finally, block a_1 performs the required summation in Equation 4.2 and gives us the probability distribution $P_\theta(x|c)$, which is used to predict the next word.

To ensure that random initializations did not influence the model performances, every experiment was run 5 times, and the mean and standard deviations are reported in the results. All experiments also used Early Stopping, with a patience of 5 epochs, and the same learning rate scheduler as used before. The following sub-sections summarize the key results from experiments performed on our MoS architecture.

4.3.1 Comparison with baseline models

Before optimizing the performance of the MoS model, we decided to compare it to our two baseline models, to see how much of a boost (in terms of perplexity on the validation set) the MoS architecture provided when the same hyper-parameters were used for all the models.

The hyper-parameters used were: context length = 35, batch size = 5, hidden size (for all LSTMs) = 650. Adam was used as the optimizer, with a learning rate of 0.001. We used Adam instead of SGD because the learning rate of SGD for the MoS model would need to be optimized before any comparisons could be made, as our second baseline model used an optimized learning rate with SGD. With Adam, the previous experiments had made it clear that a learning rate of 0.001 would be best for all models. For the second baseline model and the MoS model, a dropout of 0.5 was used at every stage (as specified in Figures 4.2 and 4.3). The MoS model had an additional hyper-parameter: the "number of experts", which was arbitrarily chosen as 10. This basically corresponds to K in Equation 4.2, the number of softmaxes that are used to produce a weighted average. Table 4.8 summarizes the results.

Table 4.8: Comparison of perplexity scores of the three models

Model	Perplexity	Time per epoch (s)	Epochs	Total time (min)	No. of Param (in millions)
Baseline 1	136 ± 1	263	8	35	16 M
Baseline 2	115 ± 1	300	39	195	20 M
MoS model	111 ± 1	400	37	247	24 M

The main comparison here is between the second baseline model and the MoS model, as the MoS model is basically the MoS structure added to the second baseline model. These results make it quite clear that though the MoS architecture is slower, there is a clear benefit of using it.

After this set of experiments, we used SGD as our optimizer and optimized the learning rate. A learning rate of 3 worked best with the MoS model. This improved our perplexity on the validation set to 104 ± 1. We also tested across various batch sizes just to reconfirm that using smaller batch sizes was the best option. We found this to be true again. Increase in the depth of the network (number of stacked LSTMs) also did not yield a better performance.

4.3.2 Hidden Size

In the next set of experiments, we tried increasing the hidden size for the MoS model. The other hyper-parameters were fixed: context length = 35, batch size = 5, number of experts = 10, dropout of 0.5, and SGD with a learning rate of 3 as the optimizer. Table 4.9 compares the perplexity achieved on the validation set by each model.

Table 4.9: Perplexity scores of hidden size (HS) experiments

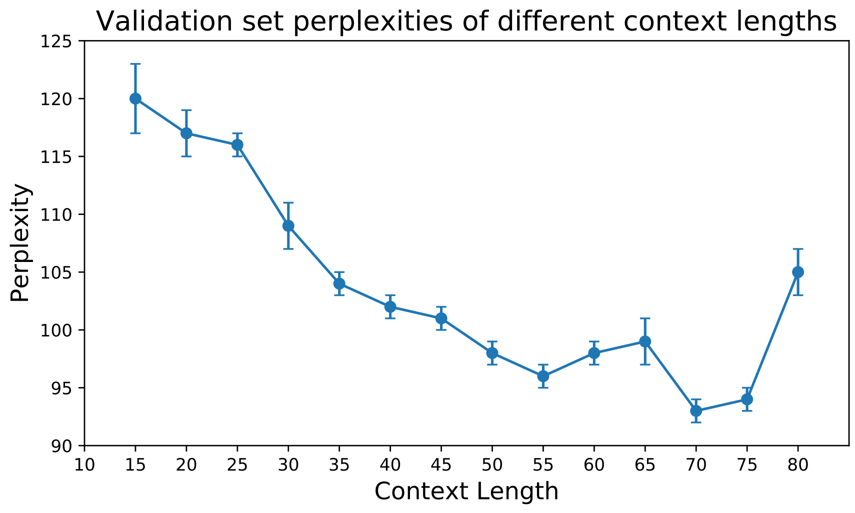
Model	Perplexity	Time per epoch (s)	Epochs	Total time (min)	No. of Param (in millions)
HS 650	104 ± 1	433	41	296	24 M
HS 1000	109 ± 2	530	42	371	46 M
HS 1250	113 ± 1	726	39	472	66 M
HS 1500	113 ± 2	850	29	410	88 M

These results suggest that increasing the hidden size only makes the training time much longer, while the added parameters lead to over-fitting instead of improvement in performance. Thus we fix the hidden size at 650 (we tried lower hidden sizes as well, but they had much lower perplexity scores, and hence are not tabulated).

4.3.3 Context Length

After fixing our hidden size, we checked if the MoS model could process longer context lengths. The hyper-parameters used in these experiments were the same as in the previous section. However, we modified our optimizer a little bit. Taking inspiration from the more sophisticated optimizers of Zaremba et al. (2014) and Yang et al. (2017), we used an aggressive learning rate of 20, and used Keras' in-built gradient clipping option to ensure that the high learning rate did not result in a poor performance. We clipped the norm of the gradients ("clipnorm") to 0.25, while using the same learning rate schedule as before.

We tried many different context lengths, and the most relevant results are presented in Figure 4.4. The MoS architecture allows this model to process much higher context lengths than either of the baseline models, which performed poorly for context lengths higher than 35. In fact, increasing the context length beyond 35 provides a significant boost to the model's performance on the validation set. The best performing model had a context length of 70, with a perplexity of 93 ± 1 on the validation set. Increasing the context length also results in lower training times, with the best model taking 396s per epoch. The high learning rate also meant that the total compute time dropped a bit, as each model took about 35 epochs to finish training (on average), instead of 38-39 earlier.



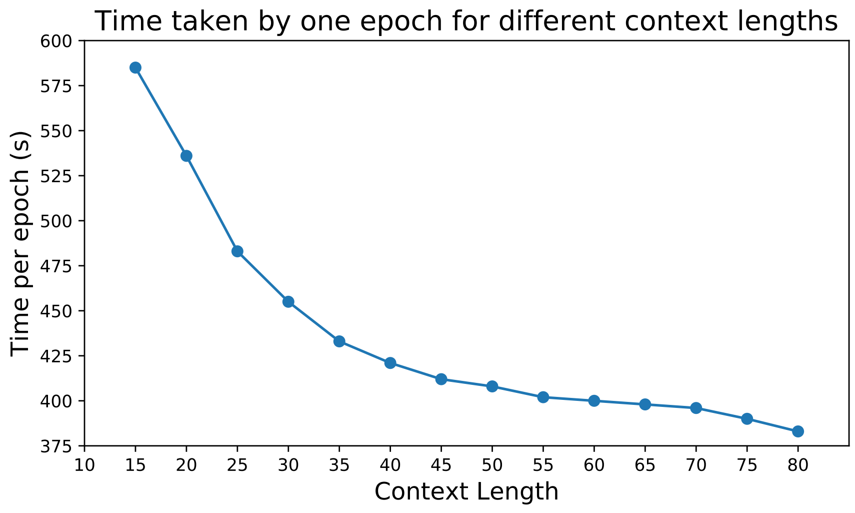


Figure 4.4: Results of context length experiments.

4.3.4 Number of Experts

Until now, we were arbitrarily using 10 experts for each of our models. In this set of experiments, we tested what effect changing the number of experts had on the model performance. The other hyper-parameters were fixed: context length = 70, batch size = 5, dropout of 0.5, and SGD with a learning rate of 20 and "clipnorm" of 0.25 as the optimizer. Figure 4.5 shows the results.

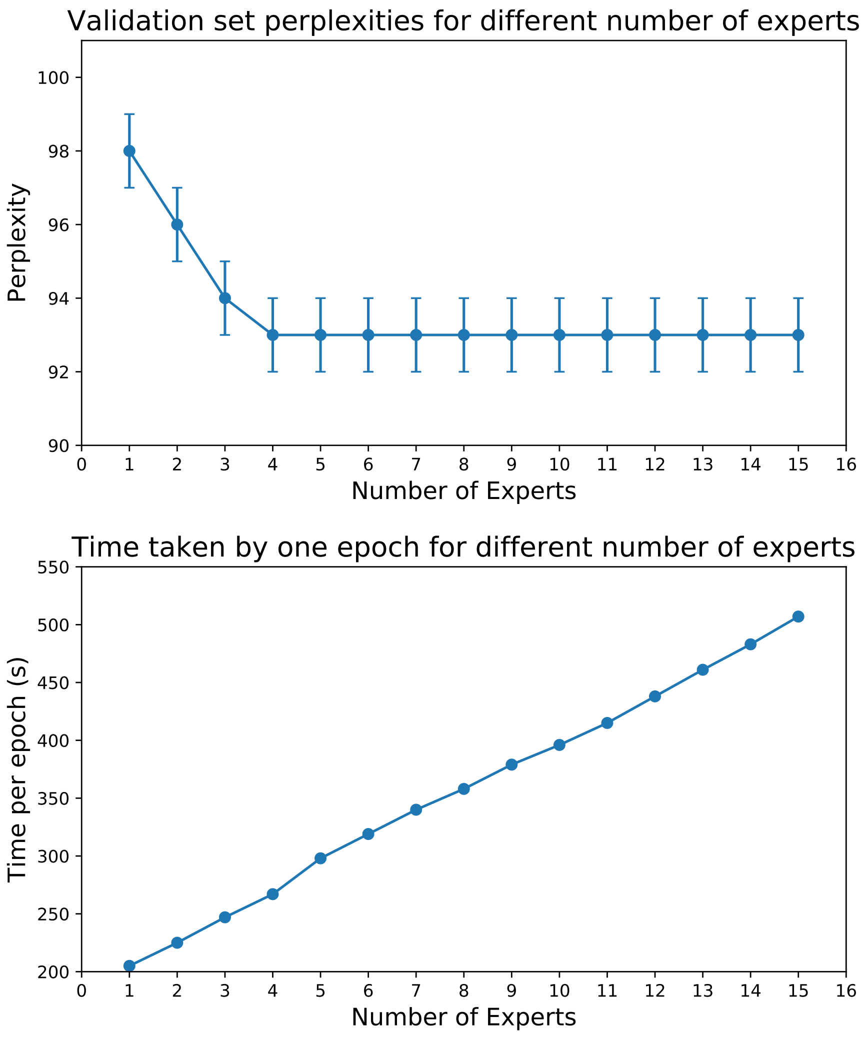


Figure 4.5: Results of number of experts experiments.

These results make it clear that increasing the number of experts beyond 4 does not provide any improvement in model performance. However, the number of parameters increases by about 0.4 million, when the number experts increases by 1. Higher number of experts also take longer training times. Thus it is in our interests to keep the number of experts low.

Following these experiments, we implemented Random Search (Bergstra & Bengio, 2012) again, to search for a hyper-parameter combination which was better than our best model. Hundreds of combinations were tested for five epochs each, and the promising combinations were allowed to run until convergence. We found a configuration which worked better than our best model. Its hyper-parameters were: context length = 70, batch size = 5, hidden size = 700, number of experts = 4, dropout of 0.4, and SGD with a learning rate of 20 and "clipnorm" of 0.25 as the optimizer. This model achieved a **perplexity of 91** on the validation set.

4.3.5 Scaling to multiple GPUs

After winding up with the experiments to optimize the MoS model, we tried scaling our model to run on multiple GPUs, to decrease training time. Our pipeline used batch generators to generate training batches for our model. This was done because many datasets are too large to be fit into memory. In such cases, the dataset is stored on the disk, and the batch generator retrieves the required amount of data and stores it in memory. The Penn Tree Bank (PTB) dataset is not this big, but to have an idea of what would have happened if we used a bigger dataset, we nonetheless used this technique. To our surprise, we found that the training time per epoch remained roughly the same, irrespective of the number of GPUs being used. Figure 4.6 shows the time taken to train each batch and one full epoch.

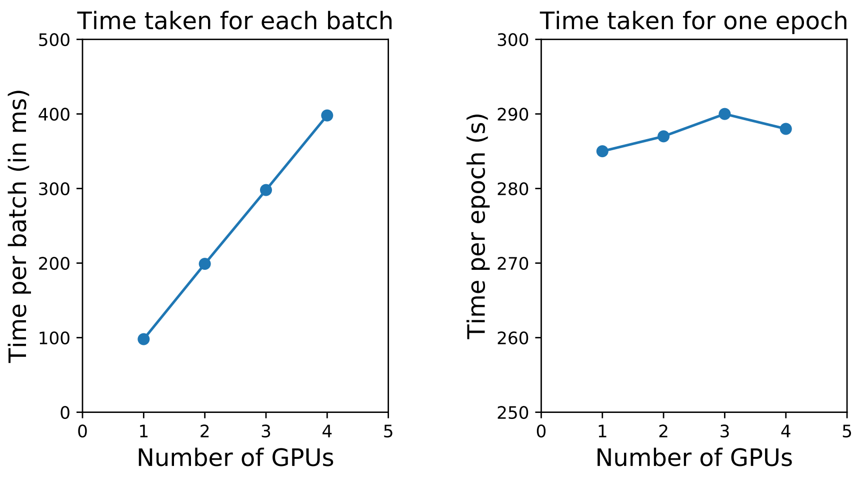


Figure 4.6: Time taken to train each batch (in milliseconds) and one full epoch (in seconds).

After analyzing these results we concluded that this was happening because of our batch generator. The linear increase in time taken to train one batch and the roughly constant time taken to train one epoch suggests that the bottleneck in our pipeline is the batch generator. The GPUs do their computations as expected, but each GPU needs to wait to receive the batch which it will compute. These batches are generated sequentially by the batch generator, and our results suggest that by the time the next GPU receives its batch, the previous GPU has already finished its computation (linear increase in time taken to process one batch). This nullifies the effect of using multiple GPUs (nearly constant time taken to train one epoch). The bottleneck created due to retrieving every batch from the disk makes this system equivalent to using just a single GPU.

Thus instead of spending time to create an efficient batch generator which would not suffer from these issues, we modified our pipeline to store the entire PTB dataset in memory, as our primary aim for this experiment was to check how well multi-GPU scaling works in Keras. Figure 4.7 shows the results. We observe that we achieve minimal speed ups on scaling to multiple GPUs.

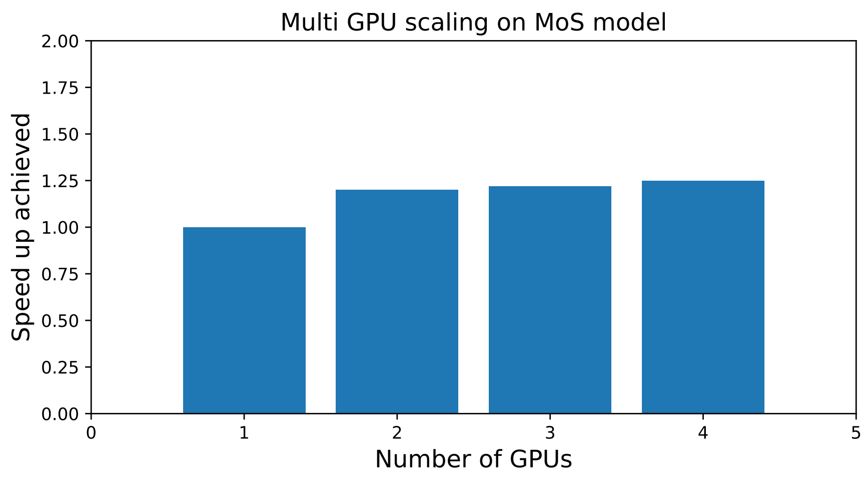


Figure 4.7: Speed ups achieved on Multi GPU scaling in Keras.

4.4 Evaluation metrics

In Section 3.3, we had discussed certain criteria which we would report. This section documents the results of those criteria.

1. **Time taken to load the data into memory and complete the pre-processing**: **0.78 seconds**. This is the average value over several runs. If we want to use multiple GPUs, then caching the data in memory is a better option.

2. **Time taken for forward and backward pass of data**:

   - Time taken to make a prediction: **0.014 seconds** per word. This value corresponds to time taken to make a forward pass of data for our best model. This result shows that even after using the slower MoS architecture, the time taken to make a prediction is still competitive, which means it can be used for applications requiring fast predictions.
   - Time taken to make one backward pass of data: **0.084 seconds**. This value is for our best model (because different models will have different times, depending on their architecture and hyper-parameters).

3. **Scaling to multiple GPUs**: On our best model, we found that scaling to multiple GPUs did not provide much speed up. Keras' updated documentation (Chollet et al., 2015) says that the data-parallel Multi-GPU option provides quasi-linear speed ups on upto 8 GPUs, but these tests have been performed on image datasets like CIFAR-10 (Krizhevsky et al., 2014) and ImageNet (Krizhevsky et al., 2012) with large batch sizes. No benchmarks exist for similar speed ups on text data. Further, several issues on this matter have also been reported (Zamecnik, 2017) (ghostplant, 2018). Thus we believe that a combination of the facts that we are using text datasets and have small batch sizes are to blame for the poor scaling to multiple GPUs.

4. **Significant differences in results**: Yang et al. (2017) reported a validation set perplexity of 56.54 and test set perplexity of 54.44 (without dynamic evaluation). In comparison, our best model had a validation set perplexity of 91.23 and a test set perplexity of 90.91. At a first glance, these results seem very poor in comparison. However, analyzing the details reveals a completely different scenario.

   Yang et al. (2017) stacked their MoS architecture over the previous state of the art model by Merity et al. (2017), which used a sophisticated modification of LSTMs, called AWD-LSTM (Weight-Dropped LSTM which uses Averaged Stochastic Gradient Descent (Polyak & Juditsky, 1992)) along with some fine-tuning techniques. In Keras, it was not possible to implement AWD-LSTMs in the available time, or use the powerful variational dropout (Gal & Ghahramani, 2016) to regularize our LSTMs (For clarity: variational dropout is available as a built-in parameter for native LSTMs, but it is not available yet for CuDNNLSTMs, which are the LSTMs we have used in this project, as they are optimized to run on GPUs).

The improvement in perplexity achieved by Yang et al. (2017) was 3.5 points over the base model of Merity et al. (2017) that they had used. In comparison, our base model was an optimized version of the model proposed by Zaremba et al. (2014), which had a perplexity of 95. With an optimized version of the MoS architecture stacked on top of it, the model reached a perplexity of 91, an improvement of 4 points over the base model. This demonstrates two things: the MoS architecture works well, and the difference between our results and Yang et al.'s (2017) results can be attributed to the effects of using AWD-LSTM with fine-tuning.

Chapter 5

Experience of using Keras

After using Keras, we can confidently say that if a person is not an expert in programming or using deep learning frameworks, then Keras will be an excellent starting point. This is not to say that it will not be a good choice for experienced users. It is a good blend of being simple to use while still being powerful enough to do serious experimentation. The coding required in Keras is quite intuitive, and the documentation in Chollet et al. (2015) is very helpful and clear. It is also updated as soon as new Keras updates are released.

The simple and straightforward coding style that is used in Keras makes building new models quick and easy. While building our baseline models, we spent minimal time in coding the core architecture. Majority of the time was spent in building an appropriate data supplying mechanism (our batch generator). Even this process was aided by the extensive discussions and solutions available on online forums (Keras' github page and Stackoverflow, among others). Community participation is a big advantage in Keras. If you face any issues, it is highly likely that others before you have faced the same or similar issues, and thus some solutions are already available. For cases when an issue is being reported for the first time, Francois Chollet (the creator of Keras) often suggests solutions himself, and fixes any relevant issues in future updates.

Keras also has several useful in-built functions which save a lot of time during model building. A case in point being the learning rate scheduler that we used in our code. Apart from providing the option of making our own learning rate schedule, Keras also has a function called "ReduceLROnPlateau", which has parameters such as "quantity to be monitored", "patience", and "factor" (the factor by which the learning rate will be multiplied, to modify it). For all but the most sophisticated learning rate schedules, this function would meet the needs perfectly. Keras' Model Checkpointing and Early Stopping functions are some of the other examples of useful in-built functions that can be used off-the-shelf. Similarly, the native Keras implementation of LSTMs (optimized for CPUs) has many useful features available as in-built function parameters, such as variational dropout, non-recurrent dropout, and various other regularizing constraints. This holds true for many other functions as well.

Keras is quite flexible with different back-ends; we can choose between Tensorflow, Theano, MXNet and Microsoft Cognitive Toolkit (CNTK). There is also an option in Keras to access certain back-end functionalities and use them in our Keras model.

However, there are some caveats which must be kept in mind while using Keras. In tasks such as language modeling, the best type of input to a model is a full sentence or paragraph (or many sentences and paragraphs, to retain long term contextual information). This implies that every batch will have a different size, as all sentences or paragraphs will not have the same number of words. As Keras uses a static computation graph, we cannot use dynamic batch sizes. This means that we have to resort to using techniques such as padding (which reduces the efficiency and accuracy of our model), or use fixed length inputs. We used the latter in our project.

Though the error messages in Keras are informative, there can be major concerns when the model has no errors (in terms of syntax) but is not performing as expected. This is because we cannot access intermediate tensors of our model, making debugging needlessly difficult. If we know exactly what we want in our models, it is easy to build them and perform extensive experiments, as was the case in our baseline models. However, if we are building a complex model where we are not sure of some of the intermediate operations, we have no way of identifying any mistakes we might be making. This happened with our Mixture of Softmaxes (MoS) model. We have to rely solely on analyzing the dimensions of each layer, and while this can be helpful, it is not guaranteed that we will get an idea of where the mistakes might be happening.

Another drawback in Keras is that the same features are not available in the GPU optimized CuDNNLSTMs, as those available in CPU optimized native implementations of LSTMs. In native LSTMs, we have several useful features (as mentioned earlier), such as variational dropout. However, these options are currently unavailable on CuDNNLSTMs. Francois Chollet has stated that these differences exist because NVIDIA (the developer of CuDNN) does not support these features yet. Hopefully this will be resolved in future updates, but as of now, these differences are present in many functions which have a different GPU optimized version which uses CuDNN.

This is an issue because using CuDNNLSTMs on GPUs is approximately *ten times faster* than using LSTMs on CPUs for the same task. Thus we are forced into making a trade-off between using fancy options on the slower LSTMs, and using basic options on the much faster CuDNNLSTMs. Since we had access to GPUs, we chose to use CuDNNLSTMs to build all our models. Our best model took about 3 hours to run on a single GPU. In comparison, the same model would have taken close to 30 hours, if we had used native LSTMs. Though it seems to have cost us some points on the final perplexity score achieved by our best model (as we could not use variational dropout, amongst other features), it allowed us to run many more experiments in this limited period of time, than would have been possible if we had used native LSTMs.

To conclude this chapter, it would be fair to say that if the objective is to quickly build and test some models, then Keras is an excellent choice. However, if we want more control over the intricate details or internal functioning of a model, then perhaps we should consider other frameworks which are more suited to meet these goals.

Chapter 6

Conclusion

Our experimental results demonstrated that the Mixture of Softmaxes (MoS) model as proposed by Yang et al. (2017) improves the performance of a language model. Our results also showed that after a certain point, increasing the number of experts does not add to the performance, instead it just makes the training time longer. The MoS architecture also allows the use of long context lengths, which improves performance and reduces training time. However, using multiple GPUs to train our model did not provide the expected speed ups. We believe this happened because we used text data with small batch sizes. Further, building this complex MoS model gave us interesting insights into Keras' strengths and limitations, which have been documented in Chapter 5.

This work can be extended in several ways. Future studies could look at implementing complex architectures which involve other types of data, such as image or audio datasets. Some research could also look into analyzing the "hotspots" of the MoS model, the parts of the computation which take the longest time, and try to come up with methods which reduce the training time. Additionally, more frameworks could be tested using the same methodology, and perhaps on various different GPUs, to check if perplexity performance and speed performance of the model suits a particular hardware configuration.

Appendix A

Appendix

Penn Tree Bank dataset: The PTB dataset (Marcus et al., 1993) has been a popular choice for language modeling tasks for a long time. Its main advantage is that it is already well pre-processed. It does not contain any capital letters, numbers, or punctuations. Its vocabulary has been capped to 10,000. This is much smaller when compared to modern datasets, and thus many Out Of Vocabulary (OoV) tokens (represented bu <unk>) are present in the data.

Pre-processing: In our code, there is minimal pre-processing, because of the reasons stated above. The only modification we make is to replace new line symbols with <eos> (end of sentence) markers.

GPUs used: NVIDIA GTX 1060

Keras Version: 2.1.3

Back-end: Tensorflow 1.4.0

The code is available at: https://github.com/singhabhijit3/msc_project_files

Bibliography

Abadi, Martín, Barham, Paul, Chen, Jianmin, Chen, Zhifeng, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Irving, Geoffrey, Isard, Michael, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pp. 265–283, 2016.

Bahrampour, Soheil, Ramakrishnan, Naveen, Schott, Lukas, and Shah, Mohak. Comparative study of deep learning software frameworks. *arXiv preprint arXiv:1511.06435*, 2015.

Bengio, Y, Ducharme, R, Vincent, P, and of machine learning, Jauvin C. A neural probabilistic language model. *Journal of machine learning* . . . , 2003a.

Bengio, Yoshua, Ducharme, Réjean, Vincent, Pascal, and Jauvin, Christian. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003b.

Bergstra, James and Bengio, Yoshua. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

Bergstra, James, Breuleux, Olivier, Bastien, Frédéric, Lamblin, Pascal, Pascanu, Razvan, Desjardins, Guillaume, Turian, Joseph, Warde-Farley, David, and Bengio, Yoshua. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, volume 1, 2010.

Chen, Tianqi, Li, Mu, Li, Yutian, Lin, Min, Wang, Naiyan, Wang, Minjie, Xiao, Tianjun, Xu, Bing, Zhang, Chiyuan, and Zhang, Zheng. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Chintala, Soumith. Convnet benchmarks, 2015.

Chollet, François et al. Keras, 2015.

Collobert, Ronan, Bengio, Samy, and Mariéthoz, Johnny. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

Deng, Jia, Dong, Wei, Socher, Richard, Li, Li-Jia, Li, Kai, and Fei-Fei, Li. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255. IEEE, 2009.

Gal, Yarin and Ghahramani, Zoubin. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in neural information processing systems*, pp. 1019–1027, 2016.

ghostplant. Multi-gpu problems, 2018. URL https://github.com/singhabhijit3/msc_project_files/blob/master/previous_project_reports/IPP-final-draft.pdf.

Jia, Yangqing, Shelhamer, Evan, Donahue, Jeff, Karayev, Sergey, Long, Jonathan, Girshick, Ross, Guadarrama, Sergio, and Darrell, Trevor. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678. ACM, 2014.

Kingma, Diederik P and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Krizhevsky, Alex, Nair, Vinod, and Hinton, Geoffrey. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014.

LeCun, Yann. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

LeCun, Yann, Bengio, Yoshua, and Hinton, Geoffrey. Deep learning. *nature*, 521(7553): 436, 2015.

Maas, Andrew L, Daly, Raymond E, Pham, Peter T, Huang, Dan, Ng, Andrew Y, and Potts, Christopher. Learning word vectors for sentiment analysis. In *Proceedings* *of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1*, pp. 142–150. Association for Computational Linguistics, 2011.

Marcus, Mitchell P, Marcinkiewicz, Mary Ann, and Santorini, Beatrice. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19 (2):313–330, 1993.

Merity, Stephen, Keskar, Nitish Shirish, and Socher, Richard. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.

Mikolov, T, Karafiát, M, Burget, L, and of ..., Černocký J. Recurrent neural network based language model. ... *Annual Conference of* ..., 2010a.

Mikolov, Tomáš, Karafiát, Martin, Burget, Lukáš, Černockỳ, Jan, and Khudanpur, Sanjeev. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010b.

Mikolov, Tomas, Yih, Wen-tau, and Zweig, Geoffrey. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 746–751, 2013.

Olah, C. Understanding lstms, 2015. URL http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

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

Rumelhart, David E, Hinton, Geoffrey E, and Williams, Ronald J. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

Seide, Frank and Agarwal, Amit. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 2135–2135. ACM, 2016.

Shi, Shaohuai, Wang, Qiang, Xu, Pengfei, and Chu, Xiaowen. Benchmarking state-of-the-art deep learning software tools. In *Cloud Computing and Big Data (CCBD), 2016 7th International Conference on*, pp. 99–104. IEEE, 2016.

Singh, A. Informatics project proposal, 2018. URL https://github.com/singhabhijit3/msc_project_files/blob/master/previous_project_reports/IPP-final-draft.pdf.

Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Tieleman, Tijmen and Hinton, Geoffrey. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Yang, Zhilin, Dai, Zihang, Salakhutdinov, Ruslan, and Cohen, William W. Breaking the softmax bottleneck: a high-rank rnn language model. *arXiv preprint arXiv:1711.03953*, 2017.

Young, Tom, Hazarika, Devamanyu, Poria, Soujanya, and Cambria, Erik. Recent trends in deep learning based natural language processing. *arXiv preprint arXiv:1708.02709*, 2017.

Zamecnik, B. Towards efficient multi-gpu training in keras with tensorflow, 2017. URL https://medium.com/rossum/towards-efficient-multi-gpu-training-in-keras-with-tensorflow-8a0091074fb2.

Zaremba, Wojciech, Sutskever, Ilya, and Vinyals, Oriol. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.