Birkbeck, University of London

Master Thesis

Machine Comprehension with Long Short Term Memory

Author:
Carmen Dibut

Supervisor:
Dr. Alessandro Provetti

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Department of Computer Science and Information Systems

September 17, 2018

Declaration of Authorship

I, Carmen DIBUT, declare that this thesis titled, “Machine Comprehension with Long Short Term Memory” and the work presented in it are my own. I confirm that:

- This report is substantially the result of my work, expressed in my own words, except where explicitly indicated in the text.
- I give my permission for it to be submitted to the JISC Plagiarism Detection Services.
- The report may be freely copied and distributed provided the source is explicitly acknowledged.

BIRKBECK, UNIVERSITY OF LONDON

Abstract

School of Business, Economics and Informatics
Department of Computer Science and Information Systems

Master of Science

Machine Comprehension with
Long Short Term Memory

by Carmen DIBUT

The objective of this Msc graduation work is to explore the impact of word representation based on deep neural network models, in the specific field of Machine Reading Comprehension. I review the theory of neural networks, which shows their potential to improve the rule-based linguistic approach and carry out experiments using word representation models such as Word2vec and Glove. I then implement a basic but robust linguistic embedding with part-of-speech POS-tagging that captures semantic and syntactic information with on a Long Short Term Memory model. The validation of my solution consist of adding unsupervised linguistic embedding representation such as part-of-speech results in an improvement applying Long Short Term Memory.

Acknowledgements

I would like to thank my supervisor, Dr. Alessandro Provetti, for his encouragement, patience and guidance during my time as a student. I could not be more grateful that he placed his trust and confidence in my abilities. I also want to thank Dr. Dell Zhang for suggesting the dataset I used to validate my research. I modified an existing LSTM software, and extensively used its documentation. A special thank you to my cousin Isidora, who made this challenge possible. Finally, a warm thanks to my husband, for his love, dedication, patience and unconditional support throughout my studies. Without him all of this would not have been possible.

Contents

List of Figures

List of Tables

List of Abbreviations

MRC	**M**achine **R**eading **C**omprehension
POS	**P**art of **S**peech
RNN	**R**eccurrent **N**eural **N**etworks
CNN	**C**onvolutional **N**eural **N**etwork
LSTM	**L**ong **S**hort **T**erm **M**emory
GRU	**G**ated **R**eccurrent **U**nit
NLP	**N**atural **L**anguage **P**rocessing
NLU	**N**atural **L**anguage **U**nderstanding
Word2vec	**W**ord to **V**ector
Glove	**G**lobal **V**ector
NER	**N**amed **E**ntity **R**ecognition
SGD	**S**tochastic **G**radient **D**escendant
EM	**E**xact **M**atch
DAG	**D**irected **A**cyclic **G**raph
ReLU	**R**ectified **L**inear **U**nit
LReLU	**L**eaky **R**ectified **L**inear **U**nit
ELU	**E**xponential **L**inear **U**nit
NLI	**N**atural **L**anguage **I**nference
SVD	**S**ingular **V**alue **D**ecomposition
CBOW	**C**ontinuos **B**ag of **W**ords

Dedicated to my husband Steven, and my cousin, Isidora

Chapter 1

Introduction

Disruptive technologies have revolutionised the customer services industry. Artificial intelligence is replacing people in tasks such as answering the phone and solving problems. Each of these AI technologies is based on Natural Language Processing. Machine Learning, alongside traditional computational linguistic methods, has transformed this field. Thus academic research is needed to demonstrate good natural language processing (NLP) and natural language understanding (NLU).

Researchers have hypothesised that difficult machine reading comprehension MRC open problems can be resolved with multi-reasoning steps of linguistic analysis and feature representation so called *feature engineering*, including syntactic parsing, part-of-speech (POS) tagging, named entity recognition (NER), question classification, semantic parsing, etc.

Only recently, neural network models have been applied to solve these open problems. A key feature of these models is the word representation, which converts words into meaningful vector representations. One of its main results is that it vastly simplifies the feature engineering process by introducing a small set of parameters, to let the trainable model learn *on its own*. Nonetheless, word representation ignores long-distance similarities between words. Unlike word embedding, part-of-speech (Also known as POS, word-classes, or syntactic categories) is a more difficult task due to the ambiguity of words. Several studies have shown that they are useful because of the large amount of information they give about a word and its neighbours.

In this work, I will first review several neural network models before selecting the optimum option. I then introduce a number of word embedding, and POS encoding methods for neural networks and discuss their advantages and disadvantages. In the next section, I describe the steps I take to implement the optimum neural network model. Finally, in the practical section, I perform a set of experiments on the model and compare the results.

Chapter 2

Background

2.1 State of the Art

There are several significant challenges when trying to resolve the Question - Answering open problem domain on Machine Reading Comprehension (MRC).

Recent research is focused on matching explicit information in the given dataset. With this objective, several high quality datasets have been created: MCTest (Richardson, Burges, and Renshaw, 2013), MovieQA (Tapaswi et al., 2016), MS MARCO dataset (T. Nguyen et al., 2016), SQuAD (Rajpurkar et al., 2016), and TriviaQA (Joshi et al., 2017).

The MCTest is a crowdsourced collection of 600 elementary level children's stories in the English language, with associated questions and multiple answers.

The stories are fictional to ensure the answer must occur on the passage of the text. The content is limited so young children and foreign language students understand. Each question has a set of four candidate answers, from single words to full structured sentences. The questions are designed to require some kind of reasoning steps, and synthesis of information across multiple sentences, making the dataset quite challenging. The dataset is small for the task of neural network. Nevertheless, recent end-to-end neural network models have been implemented successfully (Z. Wang et al., 2016), and (Trischler et al., 2016).

Rajpurkar released Stanford Question - Answering dataset: SQuAD, developed a logistic regression model with NLP pipeline. Their approach includes multiple steps reasoning of linguistic analyses with feature engineering such as POS, NER and constituency parse generated by Stanford CoreNLP. However, these approaches are heavily dependent on feature engeneering which requires deep linguistic knowledge hence they are not exhaustive. Humans outperformed their models considerably which suggests ample opportunities for improvements. The performance of the logistic regression model was F1 = 51%, and EM = 40 % compared with human, of F1=77.3%, and EM= 86.5%.

TriviaQA is a large question - answering dataset, containing 650,000 Question - Answer pairs from Wikipedia articles and web documents. In comparison with SQuAD, this dataset is more complex in term of diversity of content style and also the kind of reasoning needed to answer the questions. Compared with SQuAD, over three times as many questions in TriviaQA require reasoning across multiple sentences, which implied the difficulties over previous reasoning.

Neural network challenges on above datasets are focused on incorporating semantic matching between texts by relying on word representation (also called word embedding). A typical RNN architecture consists of an embedding, encoding, interaction, and answer layer.

Long Short Term Memory (LSTM) is the most common RNN model used for its ability to keep long term memory. LSTM has been used to build several state-of-the-art architectures (S. Wang and Jiang, 2015), (Yu et al., 2017), (Xiong, Zhong, and Socher, 2016), (Seo et al., 2016), (W. Wang et al., 2017). Inspired by Wang and Jiang 2016, several variations have been released. Multi-perspective context matching (Z. Wang et al., 2016), Bi-Directional attention (Seo et al., 2016) or Co-attention (Xiong, Zhong, and Socher, 2016).

Gated Recurrent Unit (GRU) has been used as part of LSTM architectures, mainly with good results on the answer extraction of final answer layer (Ravanelli et al., 2017), (Lei, Zhang, and Artzi, 2017).

Convolutional neural network (CNN) which is widely explored on computer vision problems has been used to address out-of-vocabulary words after word embedding, and to model contextual information with some extended. Results have been reproduced on several works suchas Multi-stages context, (W. Wang et al., 2017).

Chapter 3

Problem specification

In this chapter I will describe the problem, dataset, and the evaluation metrics.

3.1 Statement of the problem

The problem can be formally stated as follows. Given a context paragraph C with a maximum length m_C and a question Q with a maximum length m_Q. I look to identify the boundaries of the answer to that question within the context-paragraph i.e. I will look for two indices, $a_i and a_j$ such that the words from C_{ai} to C_{aj} construct my answer.

POS tagging is a sequence labelling technique for word classification, also known as lexical categories or word classes, which allows us to create **tagsets** for a particular task. Also, named entities recognition (NER), labels sequence of words in a text which are the name of things, such as person, organisations, protein names, etc, generated from POS-tags nouns.

I hypothesised that words can be close to each other if we are able to capture their linguistic relationship. I am looking to incorporate POS by looking at the *distance* between the words and the *identities* of the words that appear between them. Inferred linguistic properties such as POS have been proven to capture the above syntactic dependency. For example, words that are far from each other in a sentence (in terms of the number of words separating them) can be close to each other in the syntactic structure (Figure 3.1 and 3.2).

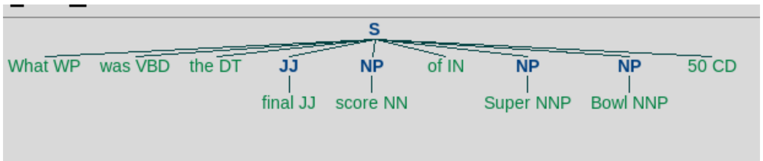

FIGURE 3.1: POS-tag of a sample question from the SQuAD dataset.

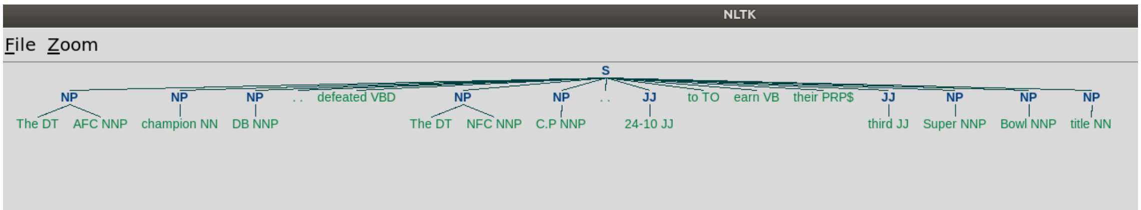

FIGURE 3.2: POS-tag of a sample context-paragraph from the SQuAD dataset.

3.2 data analysis

SQuAD is a relatively recent dataset, hence it has not been explored as much as other datasets such as MCTest or MS MARCO. A sample of an input tuple above is described in Table 3.1.

To have a better understanding of the data I will work with, I conducted several analyses of SQuAD dataset, generating comparison of the Question and Answer lengths vs its frequency as shown in histograms 3.2, and 3.3. One interesting fact to notice is that answer length and the number of answers of that length are inversely correlated.

I hypothesized that since many answers consist of only one or two words, these answers are most likely to be simple facts, thus nouns or numbers which provide my second motivation to do several experiments with word embeddings, integrating with part-of-speech POS tagging.

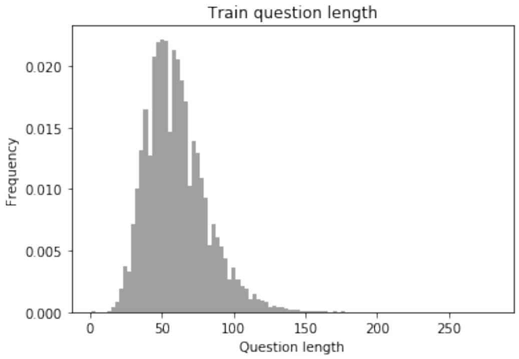

FIGURE 3.3: Histogram to show question length vs frequency for the SQuAD dataset.

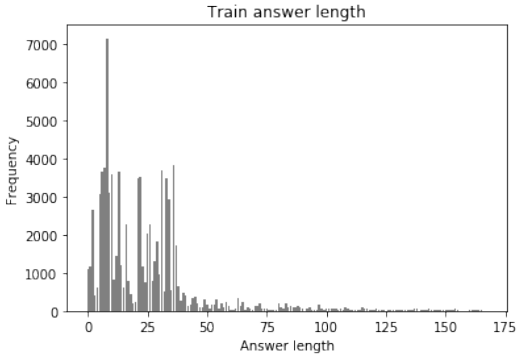

FIGURE 3.4: Histogram to show answer length vs frequency for the SQuAD dataset.

3.3 Evaluation metrics

There are different metrics to evaluate the performance of MRC problems. The most common approaches for Question - Answering tasks are:

Exact Match (EM), which measures the Percentage of span predictions that match any one of the ground-truth answers exactly, and

Description	Value
Context-paragraph	Super Bowl 50 was an American football game to determine the champion of the National Football League (NFL) for the 2015 season. The American Football Conference (AFC) champion Denver Broncos defeated the National Football Conference (NFC) champion Carolina Panthers 24–10 to earn their third Super Bowl title. The game was played on February 7, 2016, at Levi's Stadium in the San Francisco Bay Area at Santa Clara, California. As this was the 50th Super Bowl, the league emphasized the "golden anniversary" with various gold - themed initiatives, as well as temporarily suspending the tradition of naming each Super Bowl game with Roman numerals (under which the game would have been known as "Super Bowl L"), so that the logo could prominently feature the Arabic numerals 50.
Question	What was the final score of Super Bowl 50?
Answer	24-10
Answer boundaries	[40,41]

TABLE 3.1: A sample input of the context-paragrapgh, question, and its answer pointer from the SQuAD dataset.

F1 Macro-average word level score, which measures the average overlap between the prediction and the ground-truth answer. The goal of this metric is to summarize the performance of the classifier with a single number rather than a curve. It converts the standard precision p and recall r scores into a combined, average score, F_1, defined as follows.

$$F_1 = \frac{2}{\frac{1}{p} + \frac{1}{r}} = 2 \cdot \frac{p \cdot r}{p + r}$$

Where F_1 reaches the best value at 1 (perfect precision and recall) and worst at 0.

Chapter 4

Neural networks

Neural networks, and by extension deep neural networks, are a powerful class of machine learning algorithms, which use nonlinearity projection to extend vertical linear layers on top of one another. They are modelled using the biological brain as an example and aim to approximate its functions. In this section, I review several neural network models, outline the challenges of hyperparameters, and critically discuss recurrent neural networks for machine reading comprehension Question - Answering problems.

4.0.1 Definition

The basic computing unit of a neural network is a neuron, which takes several inputs and applies a weighted sum. A single neuron is equivalent to a perceptron.

$$p^{(i)} = \sum_{i=1}^{|x_i|} x_i w_i$$

It applies a nonlinear function σ (I will discuss this function in section 4.4) to the output, which is known as an activation function. Neurons are grouped in layers in a directed acyclic graph (DAG) (Cho et al., 2014). A single layer is expressed by a weight matrix W and the calculation of W^T with the input x computes in parallel. The DAG results in a feed forward network (Nielsen and Jensen, 2009) - in some literature this is called a multilayer perceptron - where the information travels in one direction. There are neural networks where data can move in both directions, forward and back. I will discuss these in section 4.3. A neural network can be represented as a series of nested functions, which will be useful later during training:

$$y^i = \sigma(w_i^T \times \sigma(w_{i-1}^T \times \sigma(w_{i-2}^T \times x))$$

4.0.2 Activation function

The following activation function was used in early perceptron models, but was not continuous and hence was not suitable for gradient-based backpropagation:

$$f(x) = \begin{cases} 1, if x \geq 0 \\ 0, otherwise \end{cases}$$

Additionally, its derivative has the following similar form:

$$f'(x) = f(x)(1 - f(x))$$

The most common activation function was the logistic sigmoid. There is ongoing research on activation functions and how they influence training. The research is

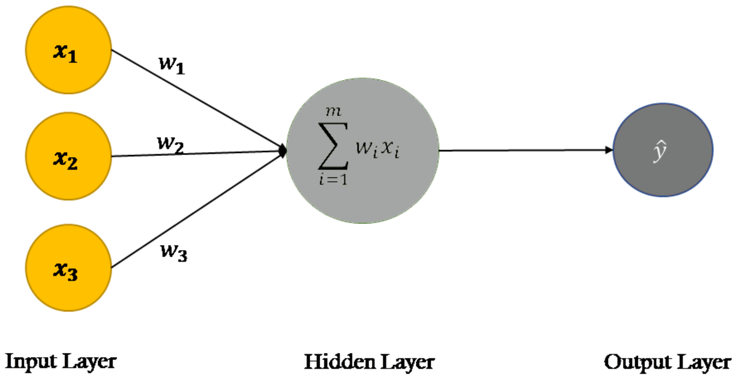

FIGURE 4.1: A single neuron from a neural network with a representation of the hierarchical layers

tackling issues such as the vanishing gradient problem, where the error gradient is too close to zero and does not influence the deeper layers (Hochreiter and Schmidhuber, 1997). This was partly addressed by introducing the Rectifier Linear Unit (ReLU) with the following form, where the derivative for large x values is 1 and the error signal does not vanish:

$$f(x) = \begin{cases} x, if x \geq 0 \\ 0, otherwise \end{cases}$$

The function ReLU also introduces sparsity (Zeiler et al., 2013), and improves generalization. There are different types of ReLU, the most commonly used on natural language processing are Leaky Rectified Linear Unit (LReLU) and Exponential Linear Unit (ELU). Finally, I mention the activation function used at the output or final layer, known as the softmax function.

$$f(x)_j = \frac{e_j^x}{\sum_{k=1}^{K} e^{x_k}} for j = 1 \ldots K$$

This function turns the output of K function into a probability distribution and is used for multiclass classification (Goodfellow, Bengio, and Courville, 2017).

4.0.3 Training

The aim of training is to minimize the loss over different training examples. Loss functions include Huber mean squad error (MSE), Kullback-Leibler divergence, Hinge classification and categorical cross-entropy. As I explained previously in section[3.1], the task is a discrete classification problem, in which the classes are mutually exclusive and the neural network model generates the probability distribution of the start index and the end index from C[a_s] to C[a_e] of the answer in the context paragraph. Thus I will focus on categorical cross entropy. The reason it is used instead of MSE is that the latter heavily penalises wrong results due to the way the error is calculated. Categorical cross entropy is defined as:

$$L(f(x_i), y_i) - \sum_{j}^{C} y_i \log f(x_i)$$

The parameters of a neural network can be updated using different methods, such as evolutionary algorithms or expectation maximization, but I will focus on gradient descent using backpropagation.

Gradient Descent methods

Gradient descent (ascent) is an iterative method to find a local minimum (maximum) function. It computes the gradient of the function to update the weights.

$$w_{i+1} = w_i - \lambda_i \nabla f(w_i), i = 0, 1, \ldots$$

The main advantage of the gradient descent method is the memory requirement. We only need the first derivative, which is O(n) parameters. For very large datasets, an approximation of the gradient descent called stochastic gradient descent (SGD) is applied. Where we can use a randomly chosen single sample or a small batch instead of the whole dataset. The disadvantage is of this method is that it generally has a slow convergence speed. We can set λ as a learning rate parameter and many functions use it to improve convergence.

I will mention second order algorithms for the sake of completeness, and discuss why they are not usable in neural networks. A commonly mentioned optimisation technique is Newton's method, which is derived from the quadratic approximation of the target function using Taylor's series expansion as follows:

$$w_{i+1} = w_i - \lambda_i H^{-1} \nabla f(w_i), i = 0, 1, \ldots$$

,

where H is the Hessian or the matrix of all second derivatives and λ is the step size or learning rate. The faster convergence is a great advantantage of this method. However, the disadvantages are amplified in neural networks due to the large number of parameters. For example, when you determine the Hessian, you find that its corresponding inverse for each layer is computationally expensive, making training very slow despite faster convergence. Another disadvantage is that the Hessian has $O(n^2)$ parameters to store, which is unfeasible for large networks. Thirdly, the Hessian makes it impossible to distribute the neural network among different CPUs or machines for large scale learning. Current trends tend toward calculations which can be computed locally in parallel.

4.0.4 Backpropagation

Backpropagation is the use of the chain rule of derivatives on the composed function to update the weights at each layer (Rumelhart, Hinton, and Williams, 1986). Firstly, we do a *forward pass* by putting in the data in the network and propagating it to the last layer. Then, we calculate the error by comparing the output with the true labels (in my case I use categorical cross-entropy). I subsequently update the weights, starting with the last layer, and going from right to left, to the input. The change in a given hidden layer depends on its own derivative and its output weighted by the succeeding layer's derivatives.

4.0.5 Optimizer

Common improvements to gradient descent methods include tuning the learning rate. A simple method is exponential decay, where we gradually lower the learning rate between iterations, hence if a dataset is sparse this can be a problem because some parameters will converge very slowly. Further algorithms such as adaptive subgradient methods (Adagrad) try to completely remove the learning rate. Alternatively, another can be added to make it easier to compute. I will mention both in the following list:

- SGD method with momentum adds a portion of the previous output to the gradient. Intuitively, the momentum helps the gradient to accelerate in the correct direction.
- Adaptive moment estimation *Adam* (Kingma and Ba, 2014) is another method to compute the average of the gradient, or squared gradient. At each iteration h it add updates from previous iterations h_{-n}, also called momentum, with a correction of the bias **b**. Adamax is a variation *Adam*, which maximises the normalisation of the gradient without a need to correct the bias.

4.1 Challenges

4.1.1 Hyperparameters

Neural network method generally contain a number of hyperparameter decisions. From network architecture to learning rate and optimisation, each decision could have an impact on the resulting generalization and training model. Common deep learning approaches for hyperparameter search, such as cross validation, tend to be infeasible for very large neural networks where training can takes weeks. Batch normalisation is a method to make a neural network more robust against variations from initial parameters (Ioffe and Szegedy, 2015). This normalisation of the input layer results in a normal distribution, which addresses the problem where the distribution of layer inputs changes along with the previous layer parameters during training (making learning more hard). Normalising the layer input makes it less dependent on previous layer changes and allows it to use higher learning rates. Other hyperparameters such as the learning rate, can be previously estimated by the optimizer. I found this to be empirically true in section 7.1 as it improves performance of the model. For batch size, a common decision is to choose the largest one that fits into the memory after setting the layer architectures. Most architecture decisions come from experience (Goodfellow, Bengio, and Courville, 2017). However, current research is focused on automatically generating neural network architectures for the given task using reinforcement learning (Paulus, Xiong, and Socher, 2017), (Zhong, Xiong, and Socher, 2017).

4.1.2 Vanishing and Exploding gradients

I have already mentioned the vanishing gradient problem in section 4.0.2. An opposite problem is the exploding gradient, where the value of the weight update tends to infinity, making further learning impossible. This is commonly detected at the very early stage as the value of the loss function quickly approaches infinity as well.

A simple solution to address this problem is the use of gradient clipping (Goodfellow, Bengio, and Courville, 2017), where normalisation of the gradient is applied if its L2 norm reaches the threshold.

4.1.3 Overfitting

Neural networks tend to overfit. It is easier for them to recall a certain feature at a specific point than learn the structure of the given problem. Overfitting is usually detected by having a validation set, and looking at the classification error and loss during training. One option to prevent overfitting is early stopping. We stop training the network when the validation rate does not improve. A commonly cited form of regularization is Dropout, where we randomly turn off a portion of neurons during training. This forces the network to be more robust and not to concentrate on a single neuron output, preventing overfit. It has been said (Mikolov et al., 2013) that dropout can be understood as averaging several models trained in an ensemble layer. I experiment extensively with this method in this study. Batch normalisation is also cited as a form of regularization (Ioffe and Szegedy, 2015).

4.1.4 Modelling power

The power of neural network is *representation learning*. This is easy to see with convolutional neural networks for computer vision, in which each filter learns to detect a simple shape such as a line or a circle. These filters are then the input of the next layer, which detects more complex shapes and so on, until finally it is fully connected and the final layer does the classification. This is considerably challenging for NLP problems, but many approaches were proposed and implemented. I test two of these, word and POS embedding using using LSTM. I will discuss these approaches in Section 5, as well as the benefit of amplified raw data by using linguistic features.

4.2 Review LSTM

LSTM (Hochreiter and Schmidhuber, 1997) is a particular type of gated RNN, which processes sequences of data and is designed to address the vanishing gradients problem. It is the first RNN architecture to introduce *gating mechanism*. LSTM uses gate vectors at each gate to control the data along the sequence, which improves the modelling of long term dependencies.

There are many types of LSTM architecture, such as the one introduced by Gers and Schmidhuber, which adds peephole connections and allows the gate layers to look at the cell gate. Another variation is GRU, from (Cho et al., 2014), which combines the forget and input gates in a single state and merges the cell and hidden gates; (Rocktäschel et al., 2015), introduced Match-LSTM, using attention mechanism to match weight vectors at the final state.

For the purposes of this study, I use original LSTM for the lexicon embedding layer, and a variation for the subsequent layers. One of the challenges of this study is to identify a method that generalises well without overfitting the model. In the next section, I describe the Match-LSTM model. In Chapter 5, I explain the different approaches to create a lexicon layer.

4.3 Match-LSTM

Match-LSTM is a variation of the original LSTM model for predicting textual entailment. Given two pieces of text, *a hypothesis* and *a premise*, it attempts to determine whether the premise entails the hypothesis. At a high level this can be applied to Question - Answering problems by considering the *question* as our premise, and we are attempting to identify a *passage* from our context-paragraph as the hypothesis. Specifically, the model denotes an input sequence and uses it as follows:

$$X = (x_1, x_2, ..., x_N)$$

,

where $x_k \in \mathbb{R}^l (1 \leq k \leq N)$. At each position k, there is a set of internal vectors, including an input gate: i_k, a forget gate: f_k, an output gate o_k, and a memory cell c_k. All these vectors are used together to generate a d-dimensional hidden state vector h_k as follows:

$i_t = \sigma(W^i x_k + V^i h_{k-1}) + \mathbf{b}^i$	Input gate
$f_t = \sigma(W^f x_k + V^f h_{k-1}) + \mathbf{b}^f$	forget gate
$o_t = \sigma(W^o x_k + V^o h_{k-1}) + \mathbf{b}^o$	output gate
$c_t = f_k \odot c_{k-1} + i_k \odot \tanh(W^c x_k + V^c h_{k-1} + \mathbf{b}^c)$	final memory cell
$h_k = o_k \odot \tanh(c_k)$	final hidden state

TABLE 4.1: Gate equations of the Long Short Term Memory cell.

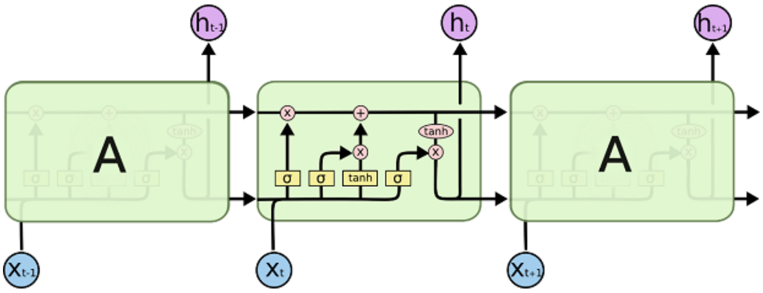

FIGURE 4.2: Diagram to show a Long Short Term Memory cell.

- The **forget gate** decides whether to keep the incoming data or not.

- The **input gate** decides what values in the cell we want to update.

- The **cell** is the network which generates the potential values.

- The **output gate** decides which part of the cell state to output.

In Table 4.2 above, σ is a sigmoid function, $\odot$ is the element wise multiplication of two real value vectors and all $\mathbf{W}^* \in \mathbb{R}^{dxl}$, $\mathbf{V}^* \in \mathbb{R}^{dxd}$, and $\mathbf{b}^* \in \mathbb{R}^d$ are weight matrices and vectors to be learned.

The *sigmoid activation function*, also called logistic function, transforms each value x into a range of [0,1], defined as follows:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

I explain below how I will modify the existing Match-LSTM to be used in the MRC Question - Answering problem of this study.

For Natural Language Inference (NLI)

For predicting textual entailment in natural language inference (NLI) problems, they have two sentences:

$$X^s = x_1^s, x_2^s, ..., x_j^s$$

$$X^t = x_1^t, x_2^t, ..., x_k^t$$

where X^s is the premise, and X^t is the hypothesis, and each x is an embedding vector of the corresponding word.

The goal of NLI is to predict a label y that indicates the relationship between X^s and X^t .

For Question - Answering MRC

I implement a modified version of NLI. For the MRC Question - Answering problem, it can be seen as a question and a paragraph:

$$H^q = h_1^q, h_2^q, ..., h_j^q$$

$$H^p = h_1^p, h_2^p, ..., h_k^p$$

Where H^q is the question, and H^p is the paragraph, and each h is an embedding vector of the corresponding word.

The goal of MRC Question - Answering is to predict a span y from the context paragraph H^p that indicates the answer to H^q.

4.3.1 Attention Mechanism

The main idea of using word-by-word attention mechanism in Match LSTM is to introduce a number of attention weighted combinations of the hidden states of the premise, which is the question for us, where each combination matches a particular word in the hypothesis. This is the context paragraph for our problem.

4.3.2 Training

The objective of training is to minimise the loss function across the different training examples. As I explained previously in section[3.1], I modelled the task as a discrete classification problem. Creating three classses: [adv1] context-paragraph, [adv2] question, [adv3] answer index In this problem, the classes are mutually exclusive and the neural network model generates the probability distribution of the start and end index from $C[a_s]$ to $C[a_e]$ of the answer in the context-paragraph. Loss functions include Huber mean squad error (MSE), Kullback-Leibler divergence, Hinge classification, and categorical cross-entropy. The model is optimised by minimising categorical cross-entropy, which is defined as follows:

$$-\sum_{n=1}^{N} \log p(a_n \mid P_n, Q_n)$$

For each training example I minimise the errors of the answer a_n, for both, the Question Q_n, and the paragraph P_n.

4.3.3 Regularization

(Srivastava et al., 2014) introduced Dropout regularization as an effective technique to reduce overfitting in deep neural networks. By randomly dropping some connections in the network during training, they reported significant improvements in test accuracy in various deep learning tasks. My goal is to share contextual information between the lexicon layer and the Match-LSTM layer without overfitting the model. Following Mikolov's interpretation, I also experiment with Dropout by applying dropout probability to the outputs of the LSTM cells in the lexicon layer, and the output of the LSTM cells in the Match-LSTM layer.

Chapter 5

Word representation

Word representation is a key component of the neural network for NLP, also called word embedding. There are a number of successful approaches, related to the distributional hypothesis of Magnus Sahlgren. This hypothesis states that *"Linguistic items with similar distributions have similar meanings"* and the idea, in which a word is well described by its context, is the main principle of the algorithm Latent Semantic Analysis (LSA), Word2vec (Mikolov et al, 2013) and Glove (Pannignton et al, 2016) algorithms.

5.1 Latent Semantic Analyis (LSA)

LSA is a powerful statistical tool, which has two main steps. The first is the construction of a term-document matrix M, with size $n \times m$. The second step corresponds to the singular value decomposition (SVD), where matrix M will be decomposed to three different matrices, as follows:

$$M_k = U_k \times S_k \times V_k^t$$

,

where S is a diagonal matrix, and U, V^t are two orthogonal matrices. The model was introduced to reduce the dimensionality of the simple co-occurrence vector method. However, the computational cost of SVD scales quadratically for the derivative matrix, resulting in poor performance for billions of words in a document.

5.2 word2vec

Word2vec is a shallow word embedding model proposed by Mikolov et al, (2013) (Mikolov et al., 2013). The main difference with previous approaches is to learn law dimensional vectors from the beginning. It predicts words based on their context by using one of two distint neural networks, continuous-bag-of-words: CBOW and Skip-Gram.

CBOW attempts to predict the next word from the surrounding words, using three neural network layers, which requires a large corpus of text. The final step of this model is the comparison between its output and the word itself in order to correct its representation based on the back propagation of the error gradient. The main purpose of CBOW is to maximise the following equation:

$$\frac{1}{V}\sum_{t=1}^{V} \log p(m_t \mid m_{t-\frac{c}{2}}...m_{t+\frac{c}{2}})$$

,

where V corresponds to the vocabulary size, and c to the window size of each word.

Skip-Gram is the opposite to CBOW in that it attempts to predict the surrounding words from the current word. In fact, the input layer corresponds to the word and the output layer corresponds to the corpus. The final step of Skip-Gram is the comparison between its input and each word of the context in order to correct its representation, based on the back propagation of the error gradient. It looks to maximise the following equation:

$$\frac{1}{V}\sum_{t=1}^{V}\sum_{j=t-c,j\neq t}^{t+c}\log p(m_j \mid m_t)$$

,

where V corresponds to the vocabulary size, c corresponds to the window size of each word.

According to Mikolov (Mikolov et al., 2013), each of these models has its own advantage. Skip-Gram generalises infrequent words well On the other hand, CBOW is faster and works well with frequent words. However, for CBOW learning the output vectors can be an expensive computational task. In this context, two algorithms are used to address this problem.

The first is The *Negative Samplin*g algorithm. The goal of this algorithm is to limit the number of output vectors that need to be updated. Hence, only a sample of these vectors are updated based on a noise distribution. This is a probabilistic distribution, which is used in the sampling process.

The second algorithm is the *Hierarchical Softmax*, which is based on a Huffman Tree. This is a binary tree, which represents all terms based on their frequencies. According to (Mikolov et al., 2013), each algorithm has a different output depending on the nature of the training data. Negative sampling performs better with low dimensional vectors and works better with frequent words. While hierarchical Softmax performs better with infrequent words.

5.3 Glove

Global Vector (Glove) (Pennington, Socher, and Manning, 2014) is the most common word representation for MRC problems on SQuAD Question - answering datasets. This algorithm is based on word occurrences in a textual corpus.

Glove is based on two main steps. The first step corresponds to the construction of the co-occurrence matrix X from a training corpus, where X_{ij} is the frequency of the word i co-occurring with the word j. The second step is the factorisation of X in order to obtain the vectors. Pannington et al show that, compared with raw probabilities, ratios help to reduce noise by identifying relevant words from irrelevant words.

This algorithm takes advantage of global count statistics instead of only local information such as Word2vec. To resolve the problem of the computational complexity of LSA, it predicts the surrounding words of every word instead of counting co-occurrence directly, therefore computationally it is more efficient.

5.4 Discussion

In conclusion, Glove combines the best of both LSA and Word2vec by using SVD and SkipGram algorithms. Hence, it is more computationally efficient, resulting in fast training and good performance for both small and large corpus. Several research papers have used pre-trained Glove for MRC problems, including Question - Answering. However, comparison studies of Glove and Word2vec performance draw contrasting conclusions. Studies carried out by Pannington, which created Glove, have found that Glove performs better than Word2vec in every task (word analogies, word similarity, word segmentation, and named entity recognition). However, a recent independent comparative study of word, embedding in both English and Arabic languages (Naili, Chaibi, and Ghezala, 2017), showed that for the task of word segmenation - no matter which language was used - Word2vec outperformed Glove. Moreover, Glove needed more data than Word2vec in the latter study to detect the semantic meaning of words.

The initial purpose of the study was to incorporate POS in an existing baseline model. However, in the course of the research when I discovered the two above contrasting studies I adjusted the objectives of the initial proposal, and decided also to test a different algorithm for the word embedding. I trained Word2vec on the latest Wikipedia articles and obtained a 300d vector for 880 billion tokens. I also used pre-trained Word2vec on Google News corpus, given by Google for research purposes with 100 million tokens. In my experiments, Word2vec trained on Wikipedia corpus outperformed Glove. This gave me the input to evaluate the POS model using Word2vec. The performance of Word2vec with Google News was very poor compared with both the above mentioned corpus, so I did not consider it further.

5.5 Lexicon encoding layer

The purpose of the lexicon encoding layer is to extract information from Q and P at the word level and normalise for lexical variants. A typical technique is a concatenation of its word embedding with other linguistic embedding.

$$\mathrm{p}_1, ..., \mathrm{p}_m = \mathrm{RNN}\mathbf{p}_1, ..., \mathbf{p}_m,,$$

where $\mathbf{p}_i$ is expected to encode useful context information (such as derived from POS tags, and named entities (NER) around the token p_i. I choose original LSTM (Hochreiter and Schmidhuber, 1997), and take $\mathbf{p}_i$ as the concatenation of each layer's hidden units at the end. As discussed above, LSTM was explicitly designed to overcome the vanishing gradient problem by applying RNN, which is the main reason I chose this model. There are two common methods to encode linguistic features. Both, binary and dense encoding, are explained below.

5.5.1 binary encoding

Binary encoding is defined as a matrix, C, of real value vectors. That is, $\mathbf{C} \in \mathbb{R}^{d \times c}$ represents the embedding of each word of the context-paragraph, with dimensionality d. This representation is padded to reach length of the question. Similarly, question Q is embedded in a matrix $\mathbf{Q} \in \mathbb{R}^{d \times q}$.

A classic technique for categorical problems is applying one-hot vector algorithm to POS-tagging of the context-paragraph p and the question q. This gives a binarised representation of unique POS-tags for each token, for both the question and the context-paragraph. These representations are successively concatenated onto **C** and **Q** to construct our final vector representation.

5.5.2 Dense encoding

Another recent technique introduced by (Chen et al, 2017) is dense POS encoding. As with binary encoding, all tokens $\mathbf{p}_i$ are represented in a context-paragraph p as a sequence of feature vectors $\tilde{p}_i \in \mathbb{R}^d$, and then pass as the input to an RNN.

Each linguistic feature has its own embedding, concatenated to its word embedding. This method increases considerably the parameters of the model. Furthermore, dimensionality is a problem as I am processing question and context-paragraph independently to capture extra information from the question.

5.5.3 Binary vs dense encoding

Binary encoding can be seen from two different angles. The first is an extension of word embedding, where each token feature is represented by its own dimension. The second is the number of parameters, reducing the computational side of the problem.

Dense encoding reduces the sparsity of word representation. However, it increases the dimensionality of the model, compared with binary encoding.

5.5.4 Features

I consider a number of category features influenced by research papers ((Rajpurkar et al., 2016), (Liu et al., 2017), Chen et al).

- Word tokens is a feature function output 1 if x_i is equal to a certain word. For the implementation I tokenise all words with NLTK.

Word in Context:

- POS tagging is a process whereby we assign a part-of-speech to a token based on its definition and context. POS-tag includes noun, verb, adjective, adverb, question words, etc. This allows us to generalise and infer entities from the sentence structure (see Figure 3.1)
- Named Entity Recognition (NER) allows us to extract real world entity information from unstructured texts to understand what the sentence is about, i.e. person, organisations, sport-events, etc. I didn't use the common NLTK NER chuncker to map the corpus. I instead opted for the new release IOB-Tagging system, also known as CoNLL representation where a chunker is a tagged classifier from POS.

- Exact match is a metric function defined as $f_{exact_match}(p_i) = \mathbb{I}(p_i \epsilon Q)$ this checks whether a passage token p_i matches the original, lowercase or lemma form of any question token.

- I also include a basic boolean feature, signalling whether or not the word is a nil value.

Chapter 6

Implementation

In this section I describe the implementation steps, the system architecture and discuss the challenges of the LSTM I have adopted.

The model is to be trained and evaluated using the SQuAD dataset. The corpus is to be preprocessed using Python's NLTK tagging system, based on perceptron learning network. I implement an LSTM three layer architecture in Torch neural network framework. Wikipedia database dump is to be processed with Python3. Word2vec is to be trained on the Wikipedia preprocessed corpus to produce word embedding results with Gensim implementation. I experiment with both pre-trained Glove and Word2vec, of 300 dimensions for both, encoder and decoder. The linguistic POS feature for each word is to be incorporated along with word embeddings through concatenation. All models are to be optimised using Adamax.

6.1 Specification of the solution in UML

The model consists of three different LSTM layers, illustrated in 6.1, and described as follows:

1. The first layer is a *lexicon encoding layer* that maps words by words to their embedding independently for the question and the passage. This layer is illustrated in 6.3
2. The second layer is the *Match attention layer*. Each Match-LSTM layer is described in section 6.1.2.
3. The final layer is an *Answer-point layer* and is described in 6.1.3.

6.1.1 Lexicon encoding layer

An overview of the Lexicon encoding with a simple LSTM layer is shown in Figure 6.1. I use two separate LSTM cells for the question, and the paragraph. I choose this method for the following reason:

- Most SQuAD questions expect noun answers. The frequency of POS in the context-paragraph shown in Histogram 6.2 shows the correlation between these and length-answer on Histogram 3.4. Intuitively, I am providing a question comprehension to identify and detect the POS of the span answer, equivalent to the pre-trained part of the context-paragraph, allowing the LSTM layer to use *contextual information* about the words it is learning.

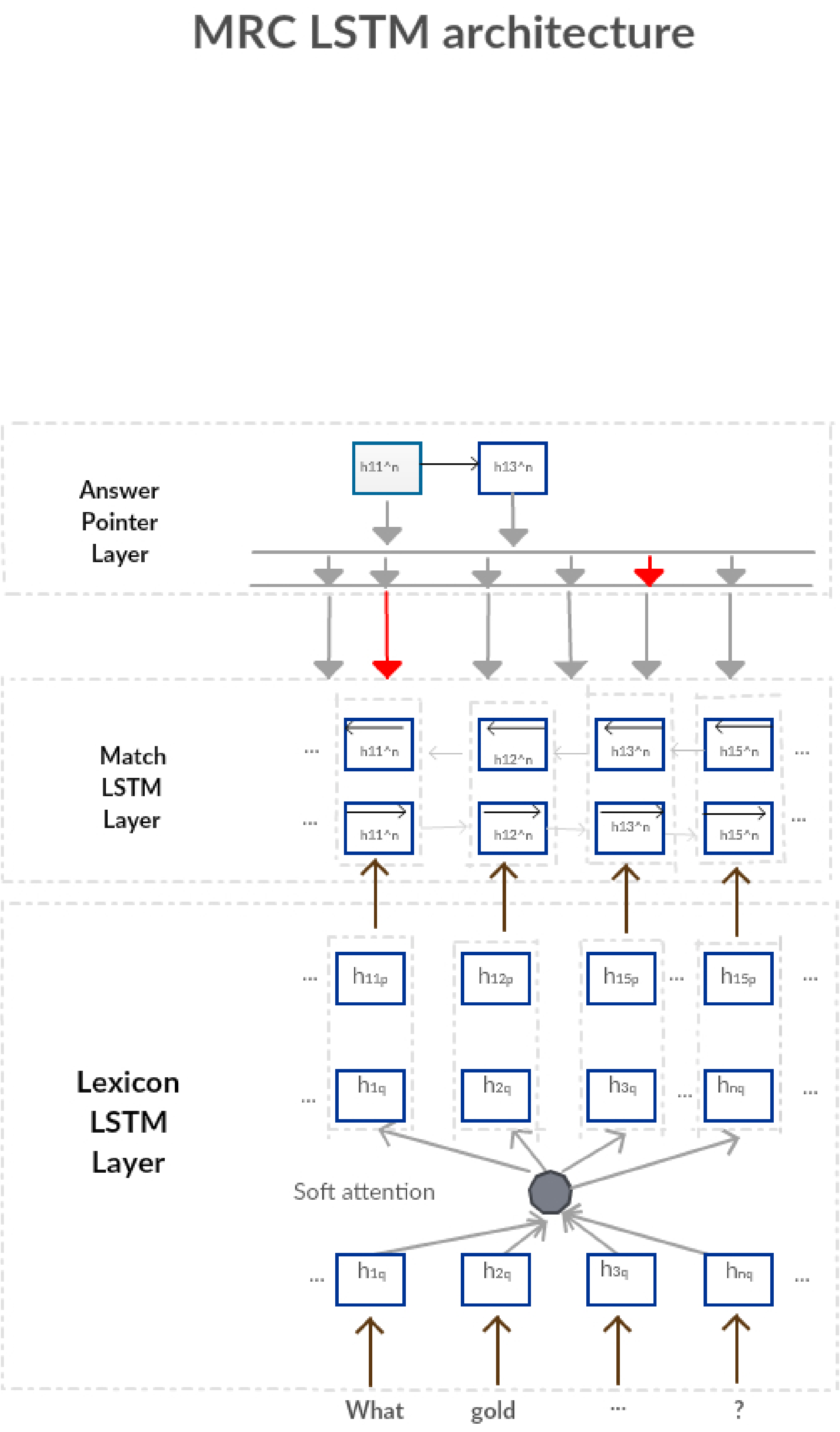

FIGURE 6.1: LSTM Architecture

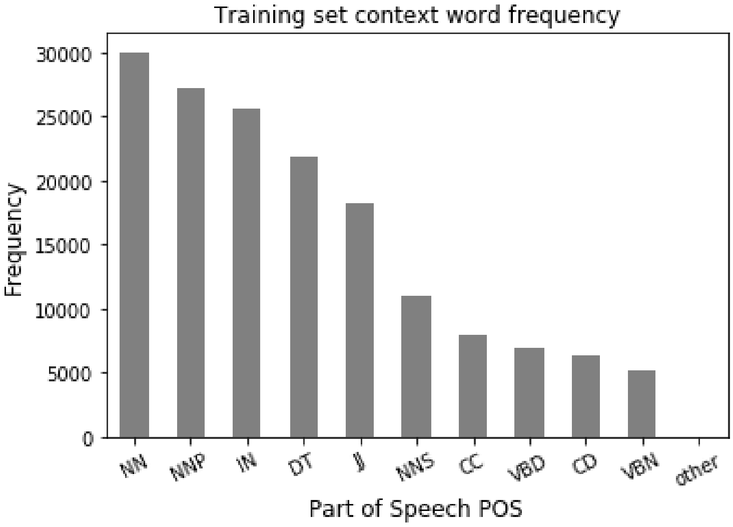

FIGURE 6.2: Histogram to show POS-tags vs frequency for a context-paragraph in the SQuAD dataset.

For the POS tagging I classify each input word based on its part of speech (POS). Following Ghodsi (2017), the POS vector is a concatenation one-hot vector for each word embedding.

The POS embedding is composed by associating each of them with its own dimension, with the value of 1 if the condition holds for word w_i and 0 otherwise. The output of this layer is a unique vector for each word on the question $Q = [q_1, ..., q_N]$, and the paragraph $P = [p_1, ..., p_M]$.

POS-embedding is implemented in function `tr:buildVacab2Emb(opt)`

The word embedding is a fixed vector for each individual word, which is pretrained with Gensim model for Word2vec. The Wikipedia file is preprocessed with `preprocess-wiki.py`, and trained with `train-word2vec-model.py`

6.1.2 LSTM input layer

I use two separate LSTM cells, one for the question, and one for the context-paragraph with extra information from the question, as follows:

$$H^p = \overrightarrow{LSTM}(\mathbf{P}), H^q = \overrightarrow{LSTM}(\mathbf{Q}) \tag{6.1}$$

The resulting matrices $H^p \in \mathbb{R}^{d \times P}$, and $H^q \in \mathbb{R}^{d \times Q}$ are hidden representations of the context-paragraph and the question. A UML of the Lexicon layer shows the input - output of the LSTM cell.

The LSTM layer is under models at: `LSTM.lua`, a section of the code word embedding is showed in Appendix B.

6.1.3 Match-LSTM layer

The Match LSTM layer goes through each word in the paragraph sequentially and computes an attention vector, which measures the similarity between it and each word in the question. The attention vector is computed as follows:

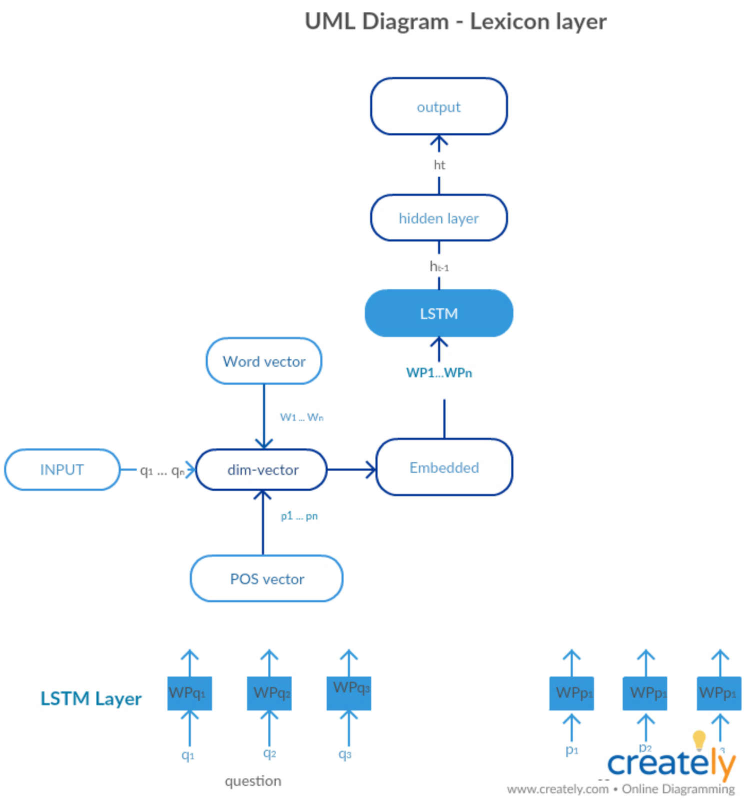

FIGURE 6.3: UML diagram to show the LSTM Lexicon Layer

$$\overrightarrow{G}i = \tanh(W^q H^q + (W^p h_i^p + W^r \overrightarrow{h}_{i-1}^r + b^p) \otimes e_Q)$$

$$\overrightarrow{\alpha}_i = softmax(w^T \overrightarrow{G}_i + b \otimes e_Q) \tag{6.2}$$

$W^q \in \mathbb{R}^{d \times q}$, $W^p, W^r \in \mathbb{R}^{d \times d}, b^p, w \in \mathbb{R}^d$, and $b \in \mathbb{R}$ are parameters to be learned. $\overrightarrow{h}_{i-1}^r \in \mathbb{R}^d$ is the hidden vector of the one directional Match-LSTM (which I explain in detail below) at position i-1, and the outer product $\otimes e_Q$ operator produces a matrix, separating the vector on the left Q times.

The resulting attention weight $\overrightarrow{\alpha}_{i,j}$ above indicates the degree of matching between the i^{th} token in the passage with the j^{th} token for the question. The attention weight vector $\overrightarrow{\alpha}_i$ is used to contain a weighted version of the question and combines it with the current token of the passage to form a vector $\overrightarrow{z}_i$.

This vector is fed into a standard one-directional LSTM to form the Match-LSTM as follows:

$$\overrightarrow{h}_i^r = \overrightarrow{LSTM}(\overrightarrow{z}_i, \overrightarrow{h}_{i-1}^r), \tag{6.3}$$

where $\overrightarrow{h}_i^r \in \mathbb{R}^d$.

To increase the encoding representation for each token in the context-paragraph, the same Match-LSTM is used in the reverse direction, defined as follows:

$$\overleftarrow{G}_i = \tanh(W^q H^q + (W^p h_i^p + W^r \overleftarrow{h}_{i+1}^r + b^p) \otimes e_Q),$$

$$\overleftarrow{\sigma}_i = softmax(w^T \overleftarrow{G}_i + b \otimes e_Q) \tag{6.4}$$

The parameters here (W^q, W^p, W^r, b^p, w, and b) are the same as used in Equation 6.2. $\overleftarrow{z}_i$ is defined in a similar way, and $\overleftarrow{h}^r_i$ is the hidden representation at position i produced by the Match-LSTM in the reverse direction.

This allows $\overrightarrow{H}^r \in \mathbb{R}^{l\times P}$ to represent the hidden states [$\overrightarrow{h}^r_{11}, \overrightarrow{h}^r_{12}, \cdots, \overrightarrow{h}^r_p$], and $\overleftarrow{H}^r \in \mathbb{R}^{l\times P}$ to represent the hidden states [$\overleftarrow{h}^r_{11}, \overleftarrow{h}^r_{12}, \cdots, \overleftarrow{h}^r_p$].

Finally, the two matrices $H^r \in \mathbb{R}^{2l\times P}$ are defined as the concatenation of the two to be passed to the Answer-pointer layer, as follows:

$$H^r = \begin{pmatrix} \overrightarrow{H}^r \\ \overleftarrow{H}^r \end{pmatrix} \tag{6.5}$$

The Match-LSTM layer is on directory models at: `LSTMwwatten.lua`

6.1.4 Answer-pointer layer

Finally, the input of the Answer-pointer layer is the information encoded from the Match-LSTM steps. My implementation uses the boundary model, in which the final output is two probability distributions, β_1, and $\beta_2 \in \mathbb{R}^p$, where p is the length of the context-paragraph, β_1 is the probability for the start-index and β_2 is the probability distribution for the end-index.

The Answer-pointer layer is under models at: `pointNet.lua`

6.2 Datasets

I evaluate different experiments on SQuAD dataset. **SQuAD** Stanford Question - Answer contains 107,785 questions-answers pairs, on a set of 536 Wikipedia articles where the answer to each question is a span of text from the corresponding reading passage. No candidate answers are provided. As described in Section 2.1, SQuAD is more realistic, and challenging than other large scale datasets such as MCTest. Advances in neural network architecture have been implemented since the launched of the dataset.

SQuAD is preprocessed in function `SQuAD`, at `preprocess.py`

6.3 Challenges with LSTM

Training LSTM architecture is expensive. For 20 epochs it took 180 minutes per epoch, for a total of 5,100 minutes per process for 5 processes. For my experiments I run the entire architecture on a NVIDIA GTX 1070 16GB GDDR5, on Linux. For a POS version of the model, I verify its functionality by overfitting on the very small subset of the dataset, running the code on a Macbook PRO. This provides a guarantee that the model is actually capable of learning.

6.4 External anotators

I use Jupyter notebook from conda for plotting histograms and data analysis. I use Gensim to preprocess the Wikipedia articles corpus, and to obtain the 300d vector for each unique token in the corpus. I use Creately to create UML diagrams.

Chapter 7

Experiments and evaluations

The main experimental question I want to answer is: Is feature representation of words a technique that can significantly improve LSTM models? To answer, I experiment with different embedding algorithms by incrementally adding features. I then feed the model's hyperparameter applying dropout after each layer, and compare the results. I opt for this approach to see the impact of each feature.

7.1 Evaluation

To be congruent with the official SQuAD dataset, F1 score and Exact Match score are used to evaluate the model. For each question, *precision* is calculated as the number of correct words divided by the number of words in the predicted answer. *Recall* is calculated as the number of correct words divided by the number of words in the ground truth answer. The F1 score is computed per question and then averaged across all questions. EM score - the higher the better - is the number of questions that are answered in the exact same words as the ground truth divided by the total number of questions.

I experiment with different pre-trained word embedding, shown in Table 7.1. From the official SQuAD leaderboard there is a trend to use *glove* algorithm for word embedding. I randomly partition context-paragraph and Question pairs in the dataset into train (80%), development (10%), and test set (10%).

7.2 Baseline results

I first try pre-trained Word2vec on Google News. However due to its poor performance I decide to train Word2vec on Wikipedia. It took five hours to process the Wikipedia corpus and more than seven hours to obtain a pre-trained Word2vec file with a 300 dimensional vector for a corpus of 880B tokens, including numbers, formatting date and hours, and phrases. I will upload the file as part of the code for further evaluation and tests.

Algorithm	Corpus	vocabulary
Word2vec	Latest Wikipedia	888B
Word2vec	Google-news	100B
Glove	Common crawl	840B

TABLE 7.1: List of word embeddings used and their sizes to evaluate the LSTM model.

Table 7.2 shows that Word2vec pre-trained on Wikipedia corpus outperforms the model outcome with Glove.

Model	Algorithm	Corpus	F1 %	EM %	Parameters
Human performance	-	-	91.2	82.3	
My Match-LSTM	Word2vec	Wikipedia	63	57.2	20Epoch
	Word2vec	Google-news	35.2	28.3	
	Glove	Common crawl	61	53	
My Match-LSTM	Word2vec	Latest Wikipedia articles	72.5	62.9	30Epoch
	Word2vec	Google-news	35.2	28.3	
	Glove	Common crawl	71.3	61.1	

TABLE 7.2: Performance comparison for the word embeddings used on the LSTM baseline model

I observe that 300 dimensional Wikipedia vectors trained With Word2vec is composed of over 30% of phrases. These are mostly nouns, such as proper nouns or entities. Nouns and Proper Nouns (NN and NNP) are also the most frequent POS-tags on SQuAD dataset. Words from SQuAD not in Word2vec are always higher than those compared with Glove. Generally, this parameter performs less well during training. To get the model to generalise better, I have set unknown words to a random vector. However, the nature of these vectors is not known. I cite that another key advantage of Skip-Gram, 5.2) is that it generalises well infrequent words. To conclude, I can say that the above result, together with the ability of Word2vec to preserve child nodes (due to the hierarchical Softmax function on Skip-Gram model), suggests that the model generalises better than Glove for this specific task, answering my first question.

7.3 Lexical embedding

I tag each token with its unique POS for context-paragraph and question classes for a total of 31 POS-taggings. The most common are shown in Table 7.3

POS tagger	Description
$\ldots(r'[0-9])$	cardinal numbers – CD
$\ldots(r'.*able',' JJ')$	adjectives – JJ, JJS, JJR, POS
$\ldots(r'.*ness',' NN')$	nouns formed from adjectives – NN
$\ldots(r'.*ly',' RB')$	adverbs – RB, RBS, WRB, WDT, WP,
$\ldots(r'.*s',' NNS')$	plural nouns – NNS
$\ldots(r'.*ing',' VBG')$	gerunds – VBG
$\ldots(r'.*ed',' VBD')$	past tense verbs – VB,VBD,VBD,VBZ, VBN
$\ldots(r'.*',' NN')$	nouns (default) – NNP, NNP, NNPS

TABLE 7.3: List of POS used to tag the SQuAD dataset.

I trained three different embeddings, for a total SQuAD vocabulary of 117K tokens. The table below shows the difference between words in SQuAD and those in vector files.

Algorithm	Words that don't appear in SQuAD
Word2vec Wikipedia	24713
Glove	29662
Word2vec Google	46321

TABLE 7.4: List of out of vocabulary words that do not appear in the SQuAD dataset.

I address the out of vocabulary problem by setting it to a random vector for a fixed window size.

Model	Algorithm	Corpus	F1 %	EM %
My-POS	word2vec	Wikipedia	73	63.9
Match-LSTM	Glove	Common Crawl	71.3	61.1
Baseline	Word2vec	Wikipedia	72.5	62.9

TABLE 7.5: Performance comparison for the word embeddings used on the LSTM POS model

7.4 POS Analysis

A sample of the vocabulary for embedding has the following form:

24 : "The DT"
25 : "Conference NNP"
26 : "AFC NNP"
27 : "Denver NNP"
28 : "Broncos NNP"
29 : "defeated VBD"
30 : "NFC NNP"
31 : "Carolina NNP"
32 : "Panthers NNP"
33 : "24–10 CD"
34 : "earn VB"
35 : "their PRP"
36 : "third JJ"
37 : "title NN"
38 : "played VBN"
39 : "on IN"
40 : "February NNP"
41 : "7 CD"

A sample of the output of the above POS encoding is shown at the end of Appendix B Both models failed to predict the correct answer. However, the ground truth answer is included in the POS prediction answer, which suggests further embedding, and more advance encoding. On the other hand, an analysis of the score F1 and EM in histograms 7.1 and 7.2 shows that distance during training between the POS model is less than the baseline model. This demonstrates the power and potential of LSTM architectures. Further studies are needed with more advanced architectures to reduce this distance, hence be able to capture syntactic information.

Number of epochs

For the results on 7.1 and 7.2, the number of epochs is calculated by the number of iterations between batch-size and number of processes. I set a total number of epochs to a maximum of 30 and 20, which is the result of number of $batchSize \times numProcesses$.

Regularization

To prevent the model overfitting the training data, I test different dropout probability values along with weight decay to the matrices H^p, H^q, and H^r, setting up DropoutP to 0.4.

To further experiment with initialisation, I set the dropout with a value near zero. This solution considerably increases the number of parameters in the system (by around 11%), but also increases F1 and EM.

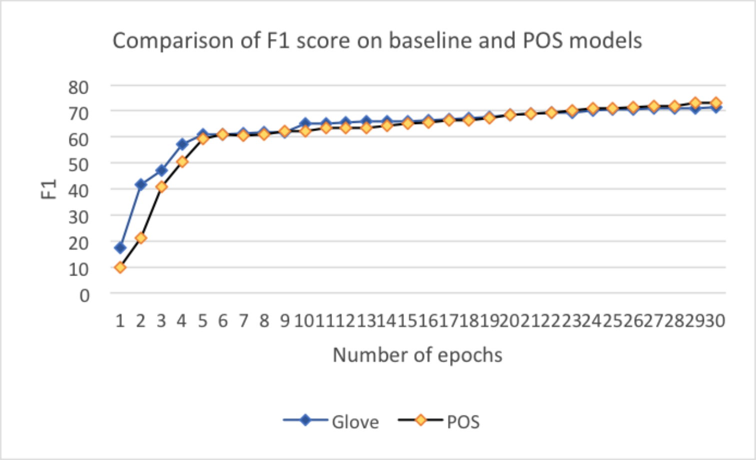

FIGURE 7.1: Comparison of the F1 score for baseline against POS during training.

7.5 Model Hyperparameters

I experiment with different parameters. For the POS model, the best results are obtained by setting a learning rate of 0.002. I decay the learning rate by 0.95 after the 10th epoch of training. The Dropout probability is set to 0.4 at the end of each LSTM layer. The number of full passes through the training data is set to 30 epochs. The word embedding is not updated during the training, and unknown words are initialised to zero. Incorporating POS outperforms the above model without excessively compromising computational time. I experiment with both regularization methods, dropout and batch size to prevent overfit of the model. In Table 6.1, a set of optimal parameters are layed down.

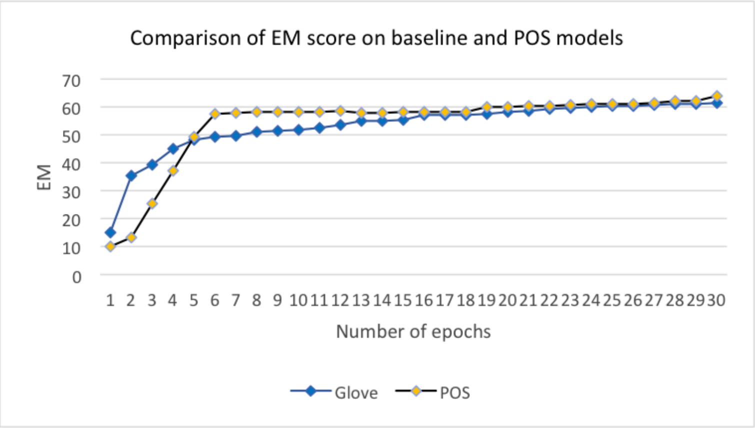

FIGURE 7.2: Comparison of the EM score for baseline against POS during training.

Hyperparamters	Baseline model	POS model
Optimizer	torch.optim.Adamax Learning rate: 0.002 decay ratio: 0.95	idem Learning rate: 0.002 decay ratio: 0.995
Regularization	DropoutP: 0.4	DropoutP: 0.4
Number of batches	6	6
Number of processes	5	5
Number of epochs	30	30
Fixed context size	300	331
Hidden state size	150	150

TABLE 7.6: List of hyperparameters used in the models.

Chapter 8

Conclusion

In this study, I investigated the impact of word representation on the performance of LSTM deep neural network models; two alternative representations were validated and compared against the SQuAD MRC Question-Answering dataset. I believe that my work has showed that deep learning models can be improved by changing the initialization of the model alongside its parameters to avoid overfitting, and ensure better generalisations.

For the SQuAD dataset, where the answer to the question is in the context-paragraph, I found that using the correct word vectors increased performance, which explains some of the behaviour of the whole architecture. For example Word2vec optimised with negative sampling could have a completely different output compare with hierarchical softmax. However, to achieve a good generalisation for more challenging datasets such as SQuAD2, where answers might not be in the context-paragraph, more sophisticated embeddings may be necessary.

I tested two word vectors and Word2vec outperformed Glove for both baseline and POS model for the chosen task. This is somewhat surprising, given that scientific articles mentioned on the SQuAD website use pre-trained Glove embeddings to evaluate their deep learning architecture against SQuAD. A major result of my work is the suggestion that the use of Word2vec could yield increased word embedding performance. I also tested pre-trained Word2vec with negative sampling on Google News for 100 million tokens. The overall evaluation for F1 and EM was very poor, with 35.2%, and 28.3% respectively.

This research confirms that deep-learning models need to be fed with huge amounts of data to achieve generality. Further investigation with different lexicon embeddings is necessary to understand deep-learning behaviour. Linguistic features such as POS tagging outperformed the baseline model at very small computational cost, without compromising the model by increasing parameters. Semi-unsupervised linguistic embedding, such as dense encoding, which were recently proposed by (Liu et al., 2017) could have a considerable impact on the current model. Further works is needed to be able to reach acceptable, human performance is not human-level on this type of tasks.

I intend to implement (Liu et al., 2017) "linguistically dense" encoding for further research. I also will evaluate my results on different datasets for MRC Question - Answering task.

Appendix A

List of code

Usage of POS model:
– cd main
– th mainDt.lua

Data:
– glove
– squad
– word2vecGoogle
– word2vecWikipedia

Preprocessing data:
– Preprocess.py
– converter.py
– processwiki.py
– trainword2vecmodel.py

Evaluation:
– Evaluate-v1.1.py
– txt2js.py

Main:
– init.lua
– mainDt.lua
– main.lua

LSTM Architecture:
– BoundaryMPtr.lua

LSTM Models:
– Embedding.lua
– LSTM.lua
– LSTMwwatten.lua
– pointNet.lua

Building vocabularies:
– loadFiles.lua
– utils.lua

Torch config CPU:
– DMax

– CAddRepTable

A.1 Section of the POS encoding

```
-- Creating Vocabulary to Embedding
function tr:buildVacab2Emb(opt)
-- ...
local emb = torch.randn(#ivocab, opt.wvecDim) * 0.05 -- Initialising
    Vocabulary embedding
-- ...
List of POS-taggers
local all_pos = {"NN","NNP","IN","DT","JJ","NNS","CC","VBD","CD","VBN",
    "RB","VBZ","VB","TO","VBP","PRP","VBG","PRP$","POS","WDT","MD","WRP
    ","NNPS","JJS","JJR","WP","EX","RBS","RBR","RP","WP$"}
-- ...
while true do
local line = file:read()
if line == nil then break end
local vals = stringx.split(line, '␣')
local glove_word = vals[1] -- Embedding Vocab
for ip = 1, #all_pos do
local pos = all_pos[ip]
local wp = glove_word .. '␣' .. pos -- POS Vocabulary
if vocab[wp] ~= nil then
--print(vocab[wp])
local index = vocab[wp]
for i = 2, #vals do

emb[index] [i-1] = tonumber(vals[i])  -- Copying embedding vector
end
-- Requirements to embedding
emb[index][ #vals + ip -1 ] = 1
embRec[index] = 1
count = count + 1
if count == #ivocab then
break
-- ...
```

A.2 Section of The LSTM architecture

```
-- ...
       -- Input of Embedding layer taking the whole vocaulary, and
          dimensionality.
       self.emb_vecs = Embedding(self.numWords, self.emb_dim)
       self.emb_vecs.weight = tr:loadVacab2Emb(self.task):float()
       if self.emb_partial then
              self.emb_vecs.unUpdateVocab = tr:loadUnUpdateVocab(
                 self.task)
       end
       -- Regularization
       self.dropoutl = nn.Dropout(self.dropoutP)
       self.dropoutr = nn.Dropout(self.dropoutP)

       self.optim_state = { learningRate = self.learning_rate }

       self.criterion = nn.ClassNLLCriterion()
       -- LSTM layer
       self.llstm = seqmatchseq.LSTM({in_dim = self.emb_dim, mem_dim =
          self.mem_dim, output_gate = false})
       self.rlstm = seqmatchseq.LSTM({in_dim = self.emb_dim, mem_dim =
          self.mem_dim, output_gate = false})
       -- Match-LSTM layer
       self.att_module = seqmatchseq.LSTMwwatten({mem_dim =
          self.mem_dim, att_dim = self.att_dim})
       self.att_module_b = seqmatchseq.LSTMwwatten({mem_dim =
          self.mem_dim, att_dim = self.att_dim})
       -- Pointer Net layer
       self.point_module = seqmatchseq.pointNet({in_dim = 2*
          self.mem_dim, mem_dim = self.mem_dim, dropoutP =
          self.dropoutP/2})
```

A.3 Section of F1 and EM score function

```
-- ... Normalizing data ...
def f1_score(prediction, ground_truth):
       prediction_tokens = normalize_answer(prediction).split()
       ground_truth_tokens = normalize_answer(ground_truth).split()
       common = Counter(prediction_tokens) & Counter(
           ground_truth_tokens)
       num_same = sum(common.values())
       if num_same == 0:
              return 0
       precision = 1.0 * num_same / len(prediction_tokens)
       recall = 1.0 * num_same / len(ground_truth_tokens)
       f1 = (2 * precision * recall) / (precision + recall)
       return f1


def exact_match_score(prediction, ground_truth):
       return (normalize_answer(prediction) == normalize_answer(
           ground_truth))

... Computing scores, Evaluating, Saving test ...
```

A.4 Hyperparameters script-section

```
-- ... model definition

-- Parameters
opt.batch_size = 6 -- 1
opt.num_processes = 5 -- 1
opt.max_epochs = 30 -- 1
opt.seed = 123 -- 10
opt.reg = 0
opt.learning_rate = 0.002
opt.lr_decay = 0.95
opt.num_layers = 1
opt.m_layers = 2
opt.dropoutP = 0.40

-- Models
opt.model = 'boundaryMPtr' -- 'sequenceMPtr'
opt.task = 'squad' -- 'snli'
opt.preEmb = 'word2vecWikipedia' --'word2vecGoogle' --'glove'
opt.grad = 'adamax' -- 'Adam'
opt.expIdx = 0
opt.initialUNK = false -- True to set all UNK to zero (without activate
    random function)
opt.emb_lr = 0
opt.emb_partial = true

-- Dimensionality
opt.wvecDim = 331 -- 300
opt.mem_dim = 150 -- 150
opt.att_dim = 150 -- 150

-- ... Other parameters
```

A.5 Preprocessing JSON data

```
# A section of the preprocessing file
              ...
              for qa in p["qas"]:
                     question = word_tokenize(qa["question"])
                     question_tag = nltk.pos_tag(question)

              qtags = []
              for words, postag in question_tag:
                     qtags.append(words +'␣'+ postag)

              if filename == 'train':
                     for a in qa['answers']:
                            answer = a['text'].strip()
                            answer_start = int(a['answer_start'])
                     ...
                     ptags = []
                     prev_context_words = word_tokenize( p["context"
                         ][0:answer_start ] )
                     prev_tag = nltk.pos_tag(prev_context_words)
                     for w, ppos in prev_tag:
                            ptags.append(w + '␣' + ppos)

                     ltags = []
                     left_context_words = word_tokenize( p["context"][
                         answer_start:] )
                     left_tag = nltk.pos_tag(left_context_words)
                     for w, lpos in left_tag:
                            ltags.append(w + '␣' + lpos)
                     ...
```

Appendix B

Model Output

B.1 Baseline output with Glove

Hyperparameters with number of epoch: 20

```
expIdx 0
maxEpochs 20
seed 123
batchSize 4
reg 0
grad adamax
numprocesses 5
model boundaryMPtr
initialUNK false
dropoutP 0.4
numWords 124164
task squad
numlayers 1
log log information
attdim 150
embpartial true
visualize false
emblr 0
preEmb glove
learningrate 0.002
mlayers 2
memdim 150
wvecDim 300
lrdecay 0.95
```

Output of F1 and EM scores for POS model with 20 epochs

```
squad: 1: "f1": 39.324524410142246, "exactmatch": 31.100283822138124,
squad: 2: "f1": 40.524646244101001, "exactmatch": 32.190383922138121,
squad: 3: "f1": 43.164624410041146, "exactmatch": 33.128390211130128,
squad: 4: "f1": 45.324524410142116, "exactmatch": 35.838221381243825,
squad: 5: "f1": 47.114221101424322, "exactmatch": 36.213382745072128,
squad: 6: "f1": 47.100423465509348, "exactmatch": 37.100163222170011,
squad: 7: "f1": 49.324524410142246, "exactmatch": 39.100283822138124,
squad: 8: "f1": 49.924418643123932, "exactmatch": 39.902324456043491,
squad: 9: "f1": 49.931117643434221, "exactmatch": 41.324524410142246,
squad: 10: "f1": 51.33152441014116, "exactmatch": 41.724524410554233,
squad: 11: "f1": 53.33154524410116, "exactmatch": 43.500183822563111,
squad: 12: "f1": 53.34452441666711, "exactmatch": 43.500183825631112,
squad: 13: "f1": 55.52432480141111, "exactmatch": 45.810183511281223,
squad: 14: "f1": 55.62133570141212, "exactmatch": 45.870183822313612,
squad: 15: "f1": 57.32452441014226, "exactmatch": 47.100282731132865,
squad: 16: "f1": 57.37152464343422, "exactmatch": 47.101412721132855,
squad: 17: "f1": 59.39111575677127, "exactmatch": 47.101412721132855,
squad: 18: "f1": 59.41111575677127, "exactmatch": 51.100283822138124,
squad: 19: "f1": 60.62452441011424, "exactmatch": 51.100283822138124,
squad: 20: "f1": 61.21452441041233, "exactmatch": 53.100283822138124,
```

Hyperparameters with number of epoch: 30

```
expIdx 0
maxEpochs 30
seed 123
batchSize 6
reg 0
grad adamax
numprocesses 5
model boundaryMPtr
initialUNK false
dropoutP 0.4
numWords 124164
task squad
numlayers 1
log log information
attdim 150
embpartial true
visualize false
emblr 0
preEmb glove
learningrate 0.002
mlayers 2
memdim 150
wvecDim 300
lrdecay 0.95
```

Output of F1 and EM scores for 30 epochs

```
squad: 1: "f1": 41.324524410142246, "exactmatch": 32.100283822138124,
squad: 2: "f1": 43.524646244101001, "exactmatch": 32.190383922138121,
squad: 3: "f1": 43.164624410041146, "exactmatch": 33.128390211130128,
squad: 4: "f1": 45.324524410142116, "exactmatch": 35.838221381243825,
squad: 5: "f1": 47.114221101424322, "exactmatch": 36.213382745072128,
squad: 6: "f1": 47.100423465509348, "exactmatch": 37.100163222170011,
squad: 7: "f1": 49.324524410142246, "exactmatch": 39.100283822138124,
squad: 8: "f1": 49.924418643123932, "exactmatch": 39.902324456043491,
squad: 9: "f1": 49.931117643434221, "exactmatch": 41.324524410142246,
squad: 10: "f1": 51.33152441014116, "exactmatch": 41.724524410554233,
squad: 11: "f1": 53.33154524410116, "exactmatch": 43.500183822563111,
squad: 12: "f1": 53.34452441014224, "exactmatch": 43.500183825631112,
squad: 13: "f1": 55.52432480141111, "exactmatch": 45.810183511281223,
squad: 14: "f1": 55.62133570141212, "exactmatch": 45.870183822313612,
squad: 15: "f1": 57.32452441014226, "exactmatch": 47.100282731132865,
squad: 16: "f1": 57.37152464344533, "exactmatch": 47.101412721132855,
squad: 17: "f1": 59.39111575677127, "exactmatch": 47.101412721132855,
squad: 18: "f1": 59.41111575677127, "exactmatch": 51.100283822138124,
squad: 19: "f1": 60.62452441011424, "exactmatch": 51.211283822138124,
squad: 20: "f1": 61.27342441041233, "exactmatch": 53.100283822138124,
squad: 21: "f1": 63.33152441014116, "exactmatch": 55.500183822563111,
squad: 22: "f1": 65.33154524410116, "exactmatch": 57.111001838333111,
squad: 23: "f1": 65.34452441014224, "exactmatch": 57.500183825631112,
squad: 24: "f1": 67.52432480141111, "exactmatch": 59.810183511281223,
squad: 25: "f1": 67.62133570141212, "exactmatch": 59.870183822313612,
squad: 26: "f1": 69.32452441014226, "exactmatch": 60.101325311328629,
squad: 27: "f1": 69.77152464343427, "exactmatch": 60.211412721123428,
squad: 28: "f1": 70.19111575677127, "exactmatch": 60.901412721132855,
squad: 29: "f1": 70.71123483245322, "exactmatch": 61.100283822138124,
squad: 30: "f1": 71.30452441011424, "exactmatch": 61.100283822138124,
```

B.2 Baseline output with Word2vec

Hyperparameters with number of epoch: 20

```
expIdx 0
maxEpochs 20
seed 123
batchSize 4
reg 0
grad adamax
numprocesses 5
model boundaryMPtr
initialUNK false
dropoutP 0.4
numWords 100000
task squad
numlayers 1
log log information
attdim 150
embpartial true
visualize false
emblr 0
preEmb word2vecWikipedia
learningrate 0.002
mlayers 2
memdim 150
```

```
wvecDim 300
lrdecay 0.95
```

Output of F1 and EM scores for 20 epochs

```
squad: 1: "f1": 11.910091125556623, "exactmatch": 10.011228387645677,
squad: 2: "f1": 35.924684624410156, "exactmatch": 17.190381392213841,
squad: 3: "f1": 48.164624410041109, "exactmatch": 40.128390211130187,
squad: 4: "f1": 55.113910018894487, "exactmatch": 44.971233817657755,
squad: 5: "f1": 55.664119900566675, "exactmatch": 47.314417760975556,
squad: 6: "f1": 55.001423465645993, "exactmatch": 48.914516660050777,
squad: 7: "f1": 56.111524124103337, "exactmatch": 49.100282227799956,
squad: 8: "f1": 56.123334431239398, "exactmatch": 50.999524456650411,
squad: 9: "f1": 57.278822343777223, "exactmatch": 51.190665777878922,
squad: 10: "f1": 57.91115245666881, "exactmatch": 52.001667999923501,
squad: 11: "f1": 58.22165558900413, "exactmatch": 52.221129900745613,
squad: 12: "f1": 59.98995101003228, "exactmatch": 53.100067700023333,
squad: 13: "f1": 59.47666439908126, "exactmatch": 53.977883511245216,
squad: 14: "f1": 61.02133570141211, "exactmatch": 54.541123009223123,
squad: 15: "f1": 61.17453221014772, "exactmatch": 55.331116695317237,
squad: 16: "f1": 61.97152464343421, "exactmatch": 56.422112721132567,
squad: 17: "f1": 62.00011575677114, "exactmatch": 56.911422881166166,
squad: 18: "f1": 62.10777468735114, "exactmatch": 57.000888456714702,
squad: 19: "f1": 62.92242356656673, "exactmatch": 57.101838670091589,
squad: 20: "f1": 63.11124410455672, "exactmatch": 57.322110034471166,
```

Hyperparameters with number of epochs: 30

```
expIdx 0
maxEpochs 30
seed 123
batchSize 6
reg 0
grad adamax
numprocesses 1
model boundaryMPtr
dropoutP 0.4
attdim 150
initialUNK false
log log information
numWords 124164
task squad
numlayers 1
lrdecay 0.95
embpartial true
visualize false
emblr 0
preEmb word2vecWikipedia
learningrate 0.002
```

```
mlayers 2
memdim 150
wvecDim 300
```

Output of F1 and EM scores for 30 epochs

```
squad: 1: "f1": 10.001788477778331, "exactmatch": 10.435553112921129,
squad: 2: "f1": 20.924646242324101, "exactmatch": 11.110383229221381,
squad: 3: "f1": 26.964624324100411, "exactmatch": 16.329914044023888,
squad: 4: "f1": 29.324524114101421, "exactmatch": 19.110934334433311,
squad: 5: "f1": 37.100283822138145, "exactmatch": 27.213382745072451,
squad: 6: "f1": 47.911711123235426, "exactmatch": 37.431229963485666,
squad: 7: "f1": 59.333386660017789, "exactmatch": 45.791244452121381,
squad: 8: "f1": 60.879340034123123, "exactmatch": 50.902365244560434,
squad: 9: "f1": 61.971119703434256, "exactmatch": 51.620024410156634,
squad: 10: "f1": 62.33152441014113, "exactmatch": 52.984524410554243,
squad: 11: "f1": 63.33154524410111, "exactmatch": 53.500183825631554,
squad: 12: "f1": 63.34452441014225, "exactmatch": 53.500183825631144,
squad: 13: "f1": 63.52432480141114, "exactmatch": 53.810183511281452,
squad: 14: "f1": 64.62133570141214, "exactmatch": 53.870183822313642,
squad: 15: "f1": 65.32452441014222, "exactmatch": 54.100282754311328,
squad: 16: "f1": 65.37152464343423, "exactmatch": 55.101412721147328,
squad: 17: "f1": 66.39111575677121, "exactmatch": 55.101476127211328,
squad: 18: "f1": 66.41111575677128, "exactmatch": 56.100283822145381,
squad: 19: "f1": 67.12452441011424, "exactmatch": 56.911283822135683,
squad: 20: "f1": 68.51452441041237, "exactmatch": 57.100283822138176,
squad: 21: "f1": 68.93152441014114, "exactmatch": 58.500183822563153,
squad: 22: "f1": 69.13154524410113, "exactmatch": 59.091110018383354,
squad: 23: "f1": 70.34452441014224, "exactmatch": 59.500183825631145,
squad: 24: "f1": 70.82432480141113, "exactmatch": 60.810183511281267,
squad: 25: "f1": 71.02133570141214, "exactmatch": 61.100183822313634,
squad: 26: "f1": 71.51452441014222, "exactmatch": 61.101325311328645,
squad: 27: "f1": 71.91715246434343, "exactmatch": 61.211412724511234,
squad: 28: "f1": 72.01111575677123, "exactmatch": 62.101412721134528,
squad: 29: "f1": 72.11123483245323, "exactmatch": 62.100283822145381,
squad: 30: "f1": 72.50087789993334, "exactmatch": 62.909960346574556,
```

B.3 POS output with Word2vec

Hyperparameters for POS

```
expIdx 0
maxepochs 30
seed 123
batchsize 6
reg 0
grad adamax
numprocesses 1
```

```
model boundaryMPtr
dropoutP 0.4
attdim 150
initialUNK false
log information
numWords 124164
task squad
numlayers 1
lrdecay 0.95
embpartial true
visualize false
emblr 0
preEmb word2vecWikipedia
learningrate 0.002
mlayers 2
memdim 150
wvecDim 331
```

Output of F1 and EM scores for POS model with 30 epochs

```
squad: 1: "f1": 20.924646244132101, "exactmatch": 13.110383922138321,
squad: 2: "f1": 40.964624410042311, "exactmatch": 25.323243934140023,
squad: 3: "f1": 50.324524410142321, "exactmatch": 37.110934345433311,
squad: 4: "f1": 59.100283822125381, "exactmatch": 49.213385327450721,
squad: 5: "f1": 60.911711123232156, "exactmatch": 57.431229921685666,
squad: 6: "f1": 60.333386660018911, "exactmatch": 57.791244212311381,
squad: 7: "f1": 60.879340012312363, "exactmatch": 57.902324456043234,
squad: 8: "f1": 61.997111970343412, "exactmatch": 57.920024410156346,
squad: 9: "f1": 62.331524410141156, "exactmatch": 57.984524410345542,
squad: 10: "f1": 63.33154524410113, "exactmatch": 58.100183824325631,
squad: 11: "f1": 63.34452441014242, "exactmatch": 58.500183845256311,
squad: 12: "f1": 63.43243248014141, "exactmatch": 57.810183515412812,
squad: 13: "f1": 64.36213357014212, "exactmatch": 57.870183822323136,
squad: 14: "f1": 65.32452441014232, "exactmatch": 58.100282731541328,
squad: 15: "f1": 65.37152464343422, "exactmatch": 58.101412725711328,
squad: 16: "f1": 66.39111575677129, "exactmatch": 58.101815323411328,
squad: 17: "f1": 66.41111575677127, "exactmatch": 58.102311234566113,
squad: 18: "f1": 67.12452441011424, "exactmatch": 59.911285438221383,
squad: 19: "f1": 68.51452441041237, "exactmatch": 59.980127575386893,
squad: 20: "f1": 68.93152441014114, "exactmatch": 60.120156483825631,
squad: 21: "f1": 69.13154524410117, "exactmatch": 60.191110018343833,
squad: 22: "f1": 69.13154524410113, "exactmatch": 59.091115001838354,
squad: 23: "f1": 70.34452441014224, "exactmatch": 59.500183825631545,
squad: 24: "f1": 71.01213357014123, "exactmatch": 61.100183865322316,
squad: 25: "f1": 71.51452441014221, "exactmatch": 61.101132531132816,
squad: 26: "f1": 71.91715246413434, "exactmatch": 61.211412721123141,
squad: 27: "f1": 71.91715246434343, "exactmatch": 61.211412724511234,
squad: 28: "f1": 72.01111575677121, "exactmatch": 62.101412721134528,
squad: 29: "f1": 72.11123483245323, "exactmatch": 62.100283822132181,
```

squad: 30: "f1": 73.01918778999331, "exactmatch": 63.909196034651751,

B.4 Output of predicted answers from the baseline and POS models.

Baseline-model	POS-model
50th Super Bowl	Carolina Panthers 24–10
50th Super Bowl	Carolina Panthers 24–10
)	Levi 's Stadium
Denver Broncos	Denver Broncos
golden	gold
February 7 , 2016	February 7 , 2016
2015 season	2015 season
February 7 , 2016	February 7 , 2016
Denver Broncos	Denver Broncos
Levi 's Stadium	Levi 's Stadium
San Francisco Bay Area at Santa Clara	California Santa Clara
2015 season	2015 season
San Francisco Bay Area at Santa Clara	San Francisco Bay Area at Santa Clara , California
Levi 's Stadium	Levi 's Stadium
2016	2016
Denver Broncos	Denver Broncos
Santa Clara	February 7 , 2016
Super Bowl 50	Super Bowl 50
50th Super Bowl	Denver Broncos
eight	eight
1995	1995
Arizona Cardinals 49–15	Arizona Cardinals 49–15
New England Patriots a chance to defend their title from Super Bowl XLIX	Super Bowl XLIX
Arizona Cardinals 49–15	Arizona Cardinals 49–15
Cam Newton	Cam Newton
Super Bowl XLIX	Super Bowl XLIX
12–4	a 12–4 record
...	...

TABLE B.1: Comparison of the answer prediction

Bibliography

Cho, Kyunghyun et al. (2014). “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078*.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2017). “Deep Learning”. In:

Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long short-term memory”. In: *Neural computation* 9.8, pp. 1735–1780.

Ioffe, Sergey and Christian Szegedy (2015). “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167*.

Joshi, Mandar et al. (2017). “Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension”. In: *arXiv preprint arXiv:1705.03551*.

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

Lei, Tao, Yu Zhang, and Yoav Artzi (2017). “Training rnns as fast as cnns”. In: *arXiv preprint arXiv:1709.02755*.

Liu, Xiaodong et al. (2017). “Stochastic answer networks for machine reading comprehension”. In: *arXiv preprint arXiv:1712.03556*.

Mikolov, Tomas et al. (2013). “Distributed representations of words and phrases and their compositionality”. In: pp. 3111–3119.

Naili, Marwa, Anja Habacha Chaibi, and Henda Hajjami Ben Ghezala (2017). “Comparative study of word embedding methods in topic segmentation”. In: *Procedia Computer Science* 112, pp. 340–349.

Nguyen, Tri et al. (2016). “MS MARCO: A human generated machine reading comprehension dataset”. In: *arXiv preprint arXiv:1611.09268*. URL: `https://arxiv.org/pdf/1611.09268.pdf`.

Nielsen, Thomas Dyhre and Finn Verner Jensen (2009). “Bayesian networks and decision graphs”. In:

Paulus, Romain, Caiming Xiong, and Richard Socher (2017). “A deep reinforced model for abstractive summarization”. In: *arXiv preprint arXiv:1705.04304*.

Pennington, Jeffrey, Richard Socher, and Christopher Manning (2014). “Glove: Global vectors for word representation”. In: pp. 1532–1543.

Rajpurkar, Pranav et al. (2016). “Squad: 100,000+ questions for machine comprehension of text”. In: *arXiv preprint arXiv:1606.05250*. URL: `https://arxiv.org/pdf/1606.05250.pdf`.

Ravanelli, Mirco et al. (2017). “Improving speech recognition by revising gated recurrent units”. In: *arXiv preprint arXiv:1710.00641*.

Richardson, Matthew, Christopher JC Burges, and Erin Renshaw (2013). “Mctest: A challenge dataset for the open-domain machine comprehension of text”. In: pp. 193–203.

Rocktäschel, Tim et al. (2015). “Reasoning about entailment with neural attention”. In: *arXiv preprint arXiv:1509.06664*. URL: `https://arxiv.org/pdf/1509.06664.pdf`.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986). “Learning representations by back-propagating errors”. In: *nature* 323.6088, p. 533.

Seo, Minjoon et al. (2016). “Bidirectional attention flow for machine comprehension”. In: *arXiv preprint arXiv:1611.01603*.

Srivastava, Nitish et al. (2014). “Dropout: a simple way to prevent neural networks from overfitting”. In: *The Journal of Machine Learning Research* 15.1, pp. 1929–1958.

Tapaswi, Makarand et al. (2016). “Movieqa: Understanding stories in movies through question-answering”. In: pp. 4631–4640.

Trischler, Adam et al. (2016). “Newsqa: A machine comprehension dataset”. In: *arXiv preprint arXiv:1611.09830*.

Wang, Shuohang and Jing Jiang (2015). “Learning natural language inference with LSTM”. In: *arXiv preprint arXiv:1512.08849*.

Wang, Wenhui et al. (2017). “Gated self-matching networks for reading comprehension and question answering”. In: 1, pp. 189–198.

Wang, Zhiguo et al. (2016). “Multi-perspective context matching for machine comprehension”. In: *arXiv preprint arXiv:1612.04211*.

Xiong, Caiming, Victor Zhong, and Richard Socher (2016). “Dynamic coattention networks for question answering”. In: *arXiv preprint arXiv:1611.01604*.

Yu, Lantao et al. (2017). “SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient.” In: pp. 2852–2858.

Zeiler, Matthew D et al. (2013). “On rectified linear units for speech processing”. In: pp. 3517–3521.

Zhong, Victor, Caiming Xiong, and Richard Socher (2017). “Seq2SQL: Generating structured queries from natural language using reinforcement learning”. In: *arXiv preprint arXiv:1709.00103*.