A QUESTION ANSWERING SYSTEM USING ENCODER-DECODER, SEQUENCE-TO-SEQUENCE, RECURRENT NEURAL NETWORKS

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

BO LI

May 2018

The Designated Committee Approves the Project Titled

A QUESTION ANSWERING SYSTEM USING ENCODER-DECODER, SEQUENCE-TO-SEQUENCE, RECURRENT NEURAL NETWORKS

by

BO LI

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2018

Dr. Chris Pollett	Department of Computer Science
Dr. Suneuy Kim	Department of Computer Science
Dr. David Taylor	Department of Computer Science

ABSTRACT

A QUESTION ANSWERING SYSTEM USING ENCODER-DECODER, SEQUENCE-TO-SEQUENCE, RECURRENT NEURAL NETWORKS

by BO LI

Question answering is the study of writing computer programs that can answer natural language questions. It is one of the most challenging tasks in the field of natural language processing. The present state-of-art question answering systems use neural network models. In this project, we successfully built a question answering system using an encoder-decoder, sequence-to-sequence, recurrent neural network. In total, five different models were tried. The first model we implemented was a previously studied model called the Match Long Short Term Memory (LSTM) & Answer Pointer model. Our second, third, fourth and fifth models were designed by making changes to the first model to try to determine if all the components of this model were necessary. By comparing the results of these five different models, we found that two non-trivial weakenings of this model had minimal effect on the accuracy of the model in determining answers. On the other hand, when these weakening were both present the accuracy became substantially worse.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER 1

INTRODUCTION

Question answering is the study of writing computer programs that can answer natural language questions. Question answering systems are widely used among search engines, personal assistant applications on smart phones, voice control systems and various other applications. Question answering systems can be categorized into two types - open domain and close domain. For an open domain system, the questions can be about almost everything; whereas, for a close domain system, the questions are about a specific domain. In this project, we focused on open domain question answering. To simplify the research topic, we focused on a scenario where a specific passage is assigned to a question and the answer is a segment of the passage. The Stanford Question Answering Dataset (SQuAD) used in this project is appropriate for such scenario. SQuAD includes questions asked by human beings on Wikipedia articles [RZLL16]. The answer to each question is a segment of the corresponding Wikipedia passage. In total, SQuAD contains more than 100,000 question-answer pairs on more than 500 articles.

In recent years, more and more state-of-art natural language processing results are produced by using neural network models. Among various neural network architectures, the encoder-decoder sequence-to-sequence recurrent neural networks were chosen for this project. These networks encode an input sequence to some vectors and then decode them to an output sequence. For question answering, the input sequence includes a passage and a question and the output sequence is the answer.

We successfully built a question answering system using five different models. Model 1 is Match-LSTM & Answer Pointer model designed by Wang and Jiang[WJ16]. This model has a typical encoder-decoder sequence-to-sequence recurrent network architecture and has a network size which is not too big to train. Model 2, 3, 4 and 5 are designed by deleting some parts or modifying some parts of Model 1. By comparing the results of five different models, two interesting observations that might improve Model 1 were found. The detailed explanation will be given in Chapter 3 and Section 5.4.

CHAPTER 2

BACKGROUND

2.1 Word Feature Vector

The concept of word feature vector was first described by Bengio, Yoshua and Ducharme in [BDVJ03]. A word feature vector represents a word according to its relationship with other words in a vocabulary. The distance from one word feature vector to any other word feature vector tells how likely the two words appear in a same context.

The word feature vector matrix for the vocabulary of a given text are learned by training a neural probabilistic language model on the text. Denote V as the vocabulary, w_t as a word from V, and the matrix C as the word feature vectors of all words in V. Each instance of the training set is a sequence of words $w_1; :::; w_T$ which is a segment of the text. The purpose of a neural probabilistic language model is to learn a model f such that

$$f(w_t; :::; w_{t\ n+1}) = \hat{P}(w_t | w_{t\ 1}; :::; w_{t\ n+1}):$$

The computation of $f(w_t; :::; w_{t\ n+1})$ is divided into two parts: First, each w is mapped to a word feature vector by selecting the corresponding row in C to get

$$x = (C(w_{t\ 1}); :::; C(w_{t\ n+1})):$$

Second, we get $f(w_t; :::; w_{t\ n+1})$ through

$$y = b + W \quad x + U \quad \tanh(d + H \quad x)$$

and

$$f(w_t; \ldots; w_{t \quad n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}.$$

The loss function to minimize is

$$L = \frac{1}{T} \sum_t \log f(w_t; \ldots; w_{t \quad n+1}).$$

Using word feature vectors together with neural network models enables learning dependencies on long sentences. In the neural probabilistic language model, the word feature vectors are used to predict the next word after a sequence. However, the usage of word feature vectors is far beyond this. Using word feature vectors to represent words is common when applying neural network models on natural language processing tasks. This is how we used word feature vectors in this project.

2.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) [RHW86] are used for modeling sequential data. Figure 2.1 shows a simple recurrent network with no outputs. x is the input. h is the state. is the hyperparameter. The relation between h and x is

$$h_t = f(h_{t \quad 1}; x_t; \quad).$$

An example of f is

$$h_t = \mathrm{sigmoid}(W_h h_{t \quad 1} + W_x x_t + b).$$

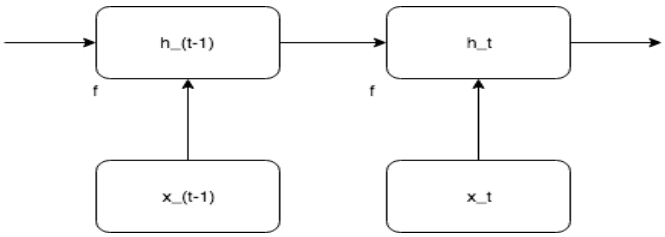

Figure 2.1: A simple recurrent neural network

Despite the suitability of applying RNNs to sequential data, the vanishing gradient problem exists. Here the vanishing gradient means the gradients become smaller and smaller as their values are propagated forward in a network. When this happens, the network learns slowly or even stops learning. The main solution to the vanishing problem is to use a more complex learning unit. In 1997, Hochreiter invented Long Short Term Memory (LSTM) cell [HS97] which reduces the vanishing problem. A LSTM has a memory cell to remember long term context and uses a forget gate, a input gate and a output gate to control how much information to flow into and out of the current unit. Aside from LSTM, Cho et al. invented Gated Recurrent Unit (GRU)[CVMG+14] which has a simplified structure but similar function with LSTM.

In this project, I used LSTM and GRU equally as learning unit. Among various RNN structures, I mainly used two types. The first type is a RNN with recurrent connections between hidden states as shown in Figure 2.1. The sequence of states are needed. The second type is also a RNN with recurrent connections between hidden states. However, the last state is needed.

2.3 Bidirectional Recurrent Neural Networks

The RNNs in Section 2.2 operate from left to right. As such, the h_t only contains context information from x_1 to x_t, but does not contain context

information from x_{t+1} to the end. However, in most sequence-to-sequence tasks, h_t should contain the information of the whole sequence. Bidirectional RNNs make this possible. In a bidirectional RNN, one cell operates from left to right, and another cell operates from right to left. As illustrated in Figure 2.2, at time t, using both h_t and g_t can get context information of the whole sequence.

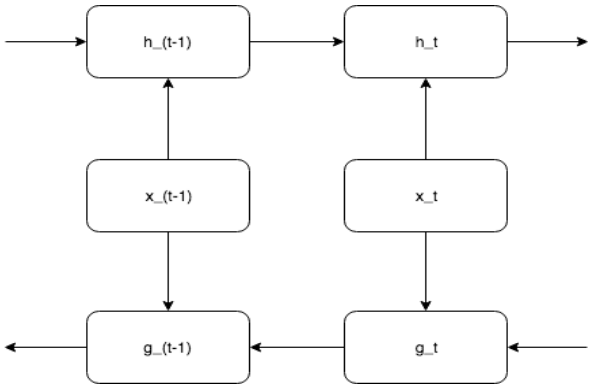

Figure 2.2: A simple bidirectional recurrent neural network

In this project, we used bidirectional RNNs in encoding part.

2.4 Encoder-Decoder Sequence-to-Sequence Architecture

Sequence-to-sequence means the input to a model is a sequence and the output from the model is also a sequence. An encoder-decoder architecture can be applied to do this task. The process of understanding the input sequence is considered as encoding the input sequence to some vectors $Crypto$. The process of generating output is considered as decoding the $Crypto$. Figure 2.3 shows the concept of encoder-decoder sequence-to-sequence architecture . x is the input, h is the state in encoding process, y is the output, and g is the state of decoding process.

The question answering task in this project is a sequence-to-sequence task. However, some additional techniques must be equipped to the basic architecture in

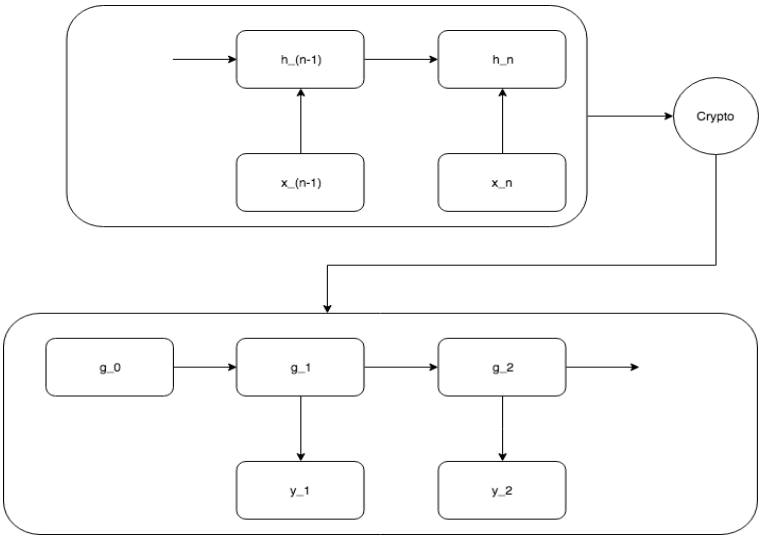

Figure 2.3: Concept of encoder-decoder sequence-to-sequence architecture

Figure 2.3. Each input actually includes two sequences - a question and a passage. As such, in the encoding process, some method is required to make each passage aware of the corresponding question and encode the two together. The attention mechanism discussed in Section 2.5 is one such method. At the same time, each output sequence is an answer which is represented by two indices for the input passage sequence. A special decoding technique called pointer network discussed in Section 2.6 is needed.

2.5 Attention Mechanism

The attention mechanism in neural networks was first described by Bahdanau et al. in the application of neural machine translation[BCB14]. In a neural machine translation task, an encoder-decoder sequence-to-sequence model encodes each input sentence to some vectors and decodes the vectors to a sentence in another language with the same content. The attention mechanism is used to enable the decoding process aware of the encoding states $h_1; :::h_n$. As shown in Figure 2.4, y is the output, g is the state, and c is the attention vector. We have

$$g_i = f(g_{i\ 1}; y_{i\ 1}; c_i):$$

The attention vector c_i is produced by using $g_{i\ 1}$ to "query" the encoding states $h_1; :::h_n$ through

$$c_i = \sum_j {}_{i;j} h_j$$

$${}_{i;j} = \exp e_{i;j} = \sum_j \exp e_{i;j}$$

$$e_{i;j} = attention(h_j; g_{i\ 1}):$$

An example of the *attention* function is $e_{i;j} = tanh(W_h h_j + W_g g_{i\ 1} + b)$.

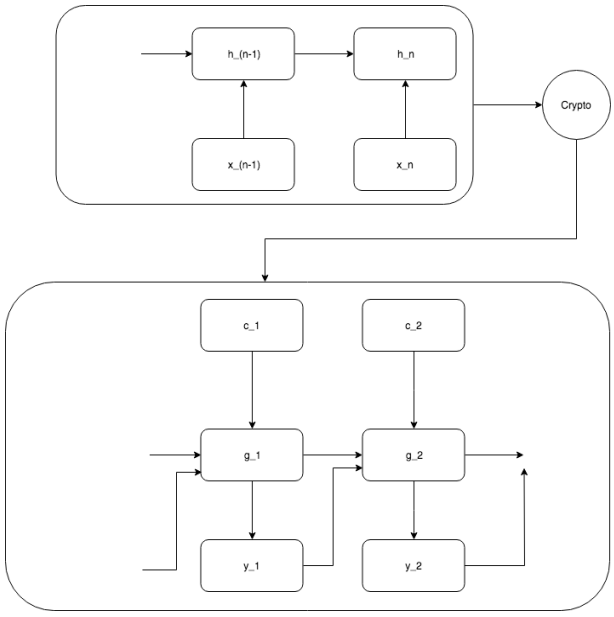

Figure 2.4: Attention mechanism in machine translation

The attention mechanism is a way to "be aware of a sequence". Since being aware of more context is a basic need of natural language processing tasks, it is reasonable to use the attention mechanism in other tasks besides machine translation.

In this project, the passage is required to "be aware of the question" in encoding process. At the same time, the answer is required to "be aware of the encoding states of passage and question". The detailed formulas are given in Chapter 3.

2.6 Pointer Network

The pointer network[VFJ15] was invented by Vinyals et al. in 2015. Using the pointer network enables the decoder to output tokens from input sequence. The attention mechanism is used with the pointer network. However, aside from getting an attention vector, the attention weight vector is considered as a probability distribution which indicates how likely each token in the input sequence is the current output. That is,

$$y_i = x_k$$

where

$$k = \mathrm{argmax}_j(\ _{i;j}):$$

Note that compared with the machine translation architecture in Figure 2.4, in the pointer network, y_i is not fed into the next decoding state.

In this project, the pointer network was used in the decoding part of several models.

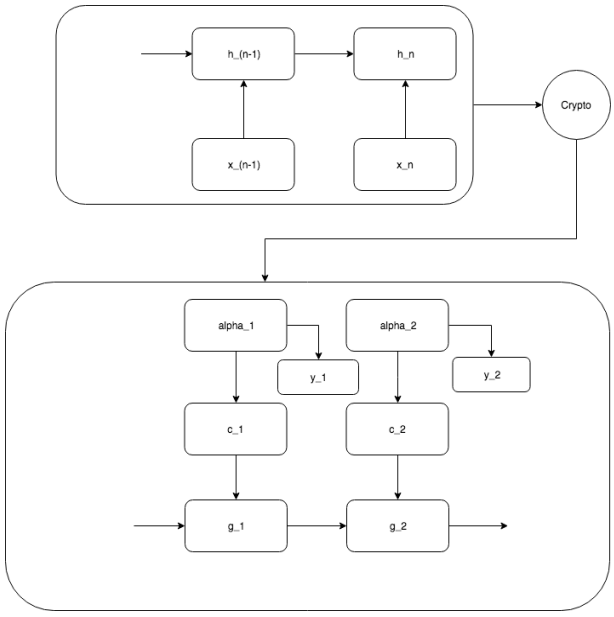

Figure 2.5: Concept of pointer network

CHAPTER 3

DESIGN

In this chapter, I will explain the five models one by one. Recall that Model 1 is the Match LSTM & Answer Pointer model which has a typical encoder-decoder sequence-to-sequence recurrent network architecture designed by Wang and Jiang[WJ16]. Model 2, 3, 4 and 5 are designed by making changes to Model 1.

3.1 Model 1

Wang and Jiang proposed an encoder-decoder sequence-to-sequence architecture for the question answering task on SQuAD dataset [WJ16]. Each instance of training data includes one passage, one question and one answer. The passage is a sequence of tokens, the question is a sequence of tokens, and the answer is a sequence of two indices indicating the start and end positions in passage. Recall that each answer is part of the corresponding passage in SQuAD. Below is an example instance of training data:

> Passage: The city had a population of 1,307,402 according to the 2010 census , distributed over a land area of **372.1** square miles (963.7 km2) . The urban area of San Diego extends beyond the administrative city limits and had a total population of 2,956,746 , making it the third-largest urban area in the state , after that of the Los Angeles metropolitan area and San Francisco metropolitan area . They , along with the RiversideSan Bernardino , form those metropolitan areas in California larger than the

San Diego metropolitan area , with a total population of 3,095,313 at the 2010 census .

Question: How many square miles does San Diego cover ?

Answer: 372.1

Before feeding training data into the model, tokens in passages and questions are vectorized to word feature vectors. As such, one pre-trained word feature vector matrix is an additional dataset required.

The vectorized training data is fed into the encoder. The encoder includes two layers - the preprocessing layer and the bidirectional match LSTM layer. In the preprocessing layer, a LSTM runs over each passage word feature vector sequence and outputs a sequence of LSTM states. The same LSTM is used to encode each question word feature vector sequence to a sequence of LSTM states.

$$H^p = \overrightarrow{LSTM}(P)$$

$$H^q = \overrightarrow{LSTM}(Q)$$

where

$$P \in R^{d \times p} : \text{passage}$$

$$Q \in R^{d \times q} : \text{question}$$

$$H^p \in R^{l \times p} : \text{encoded passage}$$

$$H^q \in R^{l \times q} : \text{encoded question}$$

$$p : \text{length of passage}$$

$$q : \text{length of question}$$

$$l : dimension\ of\ LSTM\ states$$

$$d : dimension\ of\ word\ feature\ vector$$

In the bidirectional match LSTM layer, a LSTM equipped with passage-question attention, which is called match LSTM, is used to encode each sequence of passage states and the corresponding sequence of question states together to a sequence of match LSTM states. To be specific,

$$\overset{l}{G} = tanh(W^q H^q + (W^p h_i{}^p + W^r h_i \overset{l}{}_{1}{}^r + b^p) \quad e_q)$$

$$\overset{l}{}_i = softmax(w^t \overset{l}{G}_i + b \quad e_q)$$

where

$$W^q; W^p; W^r \ 2\ R^{l\ \ l}$$

$$b_p; w\ 2\ R^l$$

$$b\ 2\ R$$

$$h_i^p\ 2\ R^l : one\ column\ of\ H^p$$

and

$$\overset{l}{z}_i = \begin{matrix} 2 & 3 \\ 6 & h_i{}^p \\ 4 & \\ & H^q \ \overset{l}{}_i{}^T \end{matrix} \begin{matrix} 7 \\ 5 \end{matrix} 2\ R^{2l}$$

$$\overset{l}{h}_i{}^r = LSTM(\overset{l}{z}_i; h_i \overset{l}{}_{1}{}^r):$$

After iterating between getting attention weight vector $\overset{l}{}_i$ and getting match LSTM state $h_i{}^r$ p times, we get $[h_1{}^r; :::; h_p{}^r]$. Concatenate them to get

$$\overset{l}{H}{}^r = [h_1{}^r; :::; h_p{}^r]\ 2\ R^{l\ \ p}:$$

Then go over H^p from right to left to get H^r. Concatenate $\overleftarrow{H}^r$ and H^r to get the final output of encoding process

$$H^r = \begin{bmatrix} \overleftarrow{H}^r \\ H^r \end{bmatrix} \in R^{2l \times p}.$$

The decoding process includes only one layer - the answer pointer layer. This layer is motivated by the pointer network. Wang and Jiang proposed two ways to design this layer. Here I only explain the boundary way. In this way, each output of the decoding process includes two probability distributions. The first probability distribution tells how likely each token in passage to be the start of the answer. The second probability distribution tells how likely each token in passage to be the end of the answer. To be specific,

$$F_k = tahn(V H^r + (W^a h^a_{k-1} + b^a) \otimes e_p)$$

$$_k = softmax(v^t F_k + c \otimes e_p)$$

where

$$V \in R^{l \times 2l}$$

$$W^a \in R^{l \times l}$$

$$b_a; v \in R^l$$

$$c \in R$$

$$h_{k-1}{}^a \in R^l : ith\ state\ of\ answer\ LSTM$$

and answer LSTM is

$$h_k{}^a = LSTM(H^{r\ T}_k; h^a_{k-1})$$

By iterating between the attention mechanism and the answer LSTM two times, we could get the output of the decoding process - $_0$ and $_1$.

Now we can get the loss function. Let a_s denote the ground truth start index of the answer and a_e denote the ground truth end index, we have

$$p(a|H^r) = p(a_s|H_r)p(a_r|H_r) = \quad _{0;a_s} \qquad _{1;a_e}$$

where

$$_{k;j} = j\,th\ token\ of \quad _k$$

To train the model, the loss function

$$J(\ ) = \quad \frac{1}{N} \sum_{i=1}^{N} \log p(a^n|H^r)$$

is minimized.

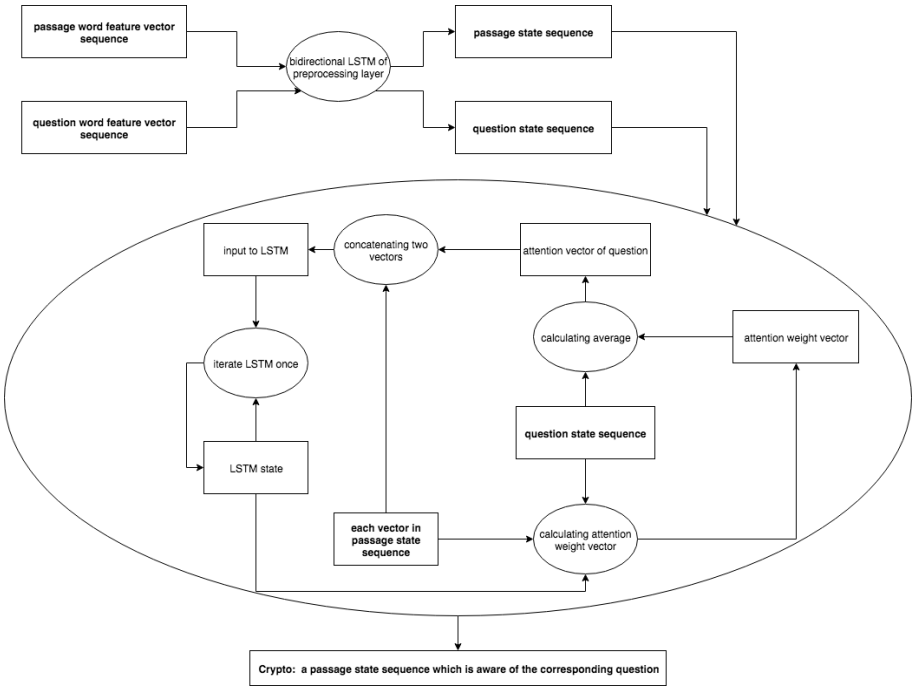

Figure 3.1: Encoder part of Model 1

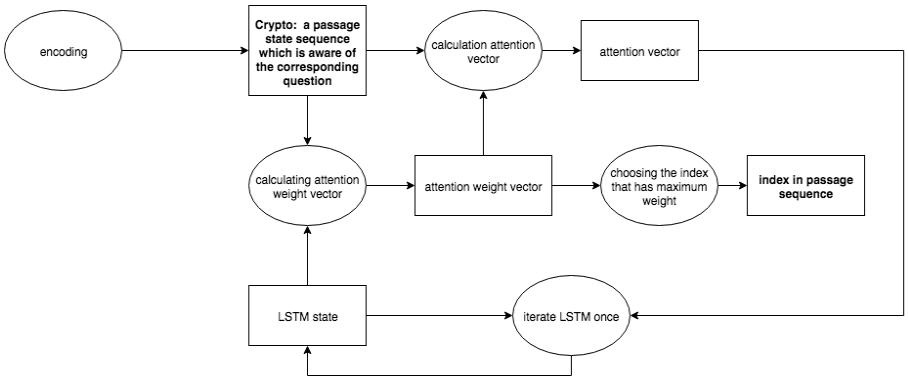

Figure 3.2: Decoder part of Model 1

3.2 Model 2

The difference from Model 2 and Model 1 is in the decoding process. In Model 2,

$$h_k^a = H^r \; {}_k^T:$$

That is, instead of the current state of answer LSTM, the previous attention vector is used to query the current attention weight vector.

3.3 Model 3

The difference between Model 3 and Model 2 is that in Model 3 the $W^r h_i \; {}_1^{!r}$ in the bidirectional match LSTM layer is removed. This modification aims at checking whether $h_i \; {}_1^{!r}$ carries some redundant context information. After this change,

$$\dot{G}^! = tanh(W^q H^q + (W^p h_i^p + b^p) \quad e_q)$$

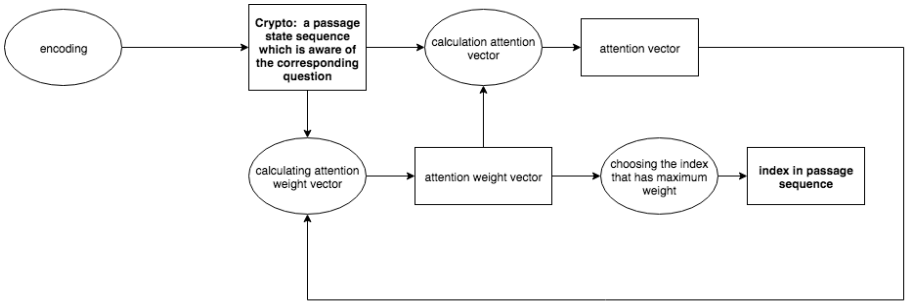

Figure 3.3: Decoder part of Model 2

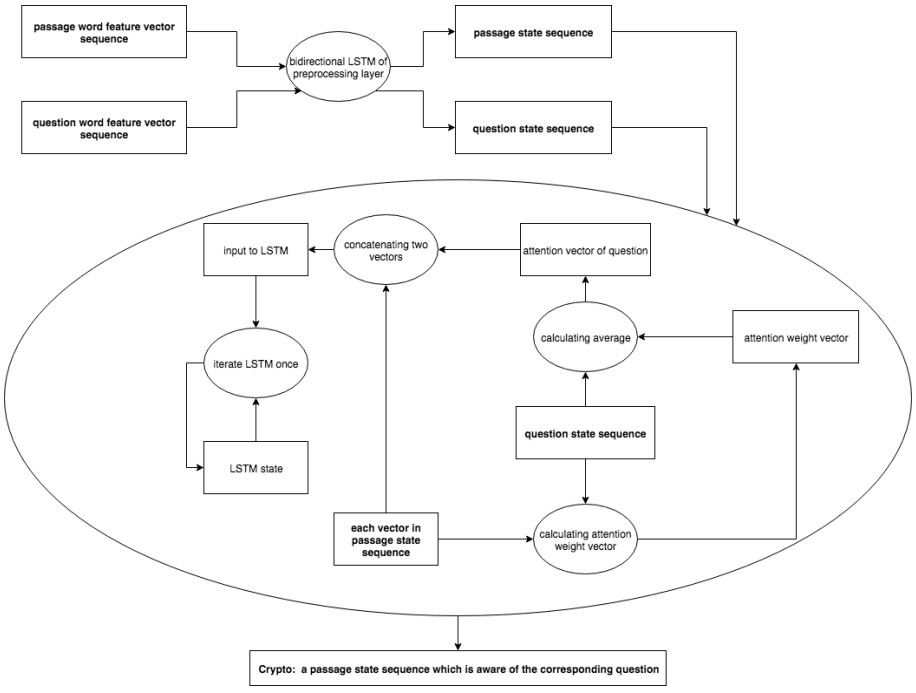

Figure 3.4: Encoder part of Model 3

3.4 Model 4

The difference between Model 4 and Model 2 is that in Model 4 the the preprocessing layer is removed. This modification aims at checking whether the preprocessing layer carries some redundant context information.

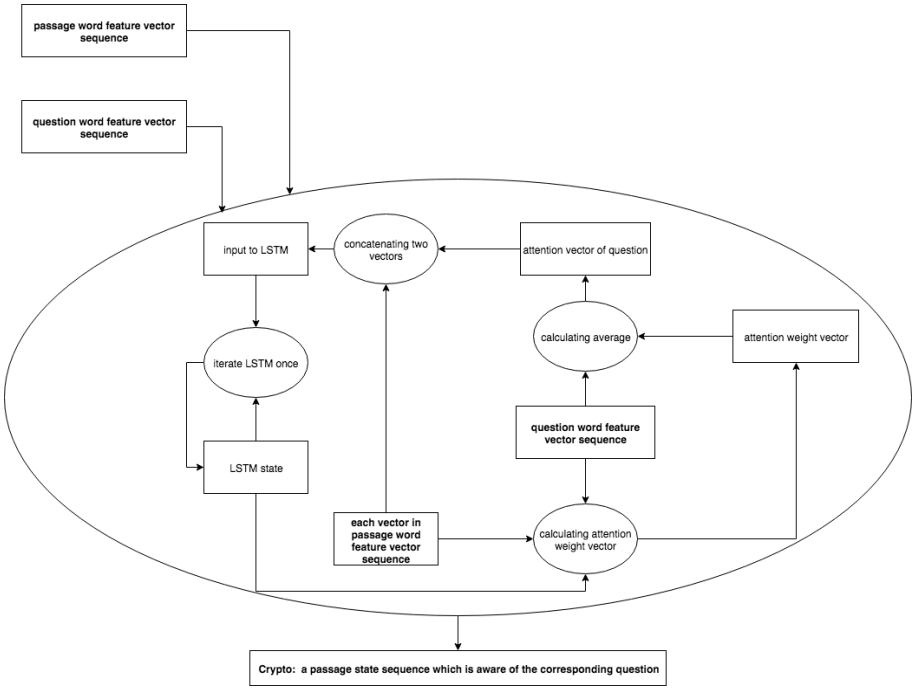

Figure 3.5: Encoder part of Model 4

3.5 Model 5

The difference between Model 5 and Model 2 is that in Model 5 both the preprocessing layer and $W^r h_i \overset{I}{\underset{1}{:}}{}^r$ in the bidirectional match LSTM layer are removed. This aims at checking whether context information carried by both is included in some other parts of Model 2.

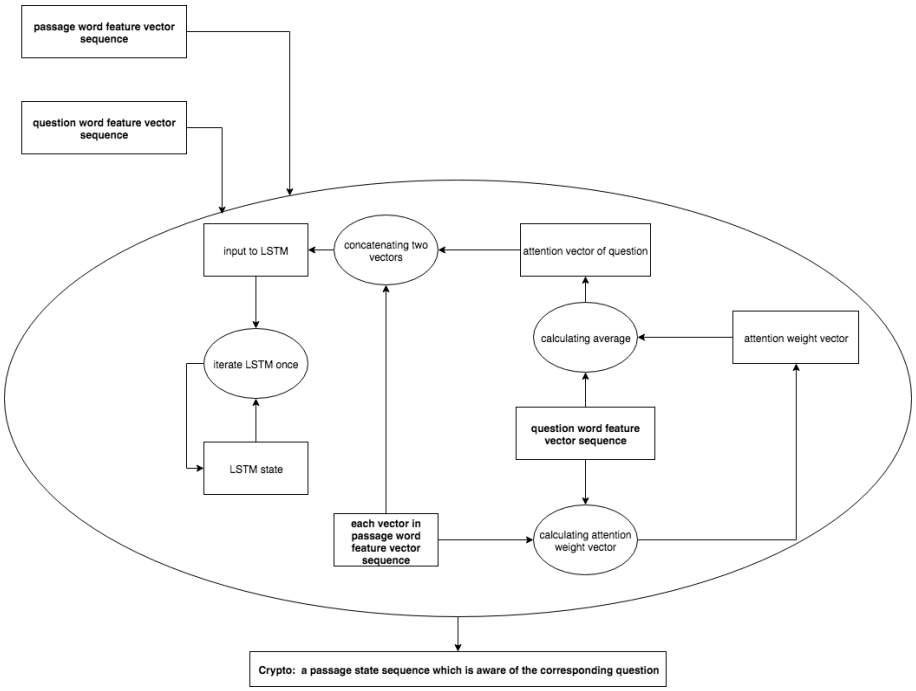

Figure 3.6: Encoder part of Model 5

CHAPTER 4

IMPLEMENTATION

4.1 Adjusting Models for Batch Training

When training a model, all of the training data is fed into the model to update the parameters. In one specific model, the number of times to iterate the encoding process is fixed. However, different passages have different lengths and different questions also have different lengths. As such, adjusting all passages to a same length and adjusting all questions to another same length are necessary.

For sequences longer than a fixed length, a part of the sentence is cut out. For sequences shorter than a fixed length, a faking pad token is used to pad them. In practice, each passage is adjusted to *passage_padding_length* and is paired with a mask vector *passage_mask* which has size *passage_padding_length*. Each question is adjusted to *question_padding_length* and paired with a mask vector *question_mask* which has size *question_padding_length*. Each entry of mask vector is either zero or one. Zero indicates that the current token does not exist in the original sequence. One indicates the opposite.

When implementing a model, every effort is made to prevent the model from distracted by tokens that do not exist. This makes the model in implementation different from the theoretical one. Take Model 1 as an example. In the preprocessing layer, after getting a sequence of states, the mask vector is used to reset the values of positions that do not exist to zero in an additional step

$$H^p = H^p \quad (\textit{passage\_mask} \quad l)$$

$$H^q = H^q \quad (question\_mask \quad l):$$

In the match LSTM layer, the attention weights of positions that do not exist are also set to zero in an additional step

$$^{l}_{i} = softmax((w^t \overset{l}{G}_i + b \quad e_q)) \quad question\_mask:$$

Similar to the preprocessing layer, we have

$$H_r = H_r \quad (passage\_mask \quad 2l):$$

In the answer pointer layer, similar to the match LSTM layer, we have

$$_k = softmax((v^t F_k + c \quad e_p)) \quad passage\_mask:$$

4.2 Tensor ow Graphs

Tensorflow is an open source software for numerical computation. It is most widely used for implementing neural networks. The central idea of Tensorflow is describing a complex numeric computation as a graph. `Variables` are "trainable" nodes. `Placeholders` are nodes whose values are fed in run time. Taking Model 1 as an example, `Variables` should be used to represent all the parameters of the encoding and decoding layers and `Placeholders` should be used to represent passages, questions, and answers. Building a computation graph using Tensorflow is very convenient. First, the API provides some building blocks which speed up coding a lot. For example, to implement a special LSTM cell with attention mechanism, we could use the RNNCell interface instead of coding every formula from scratch. Second, we do not need to calculate gradients by hand. To train a graph, some APIs of Tensorflow are called to get a train operation. Then the training data is fed through `Placeholders` to run the train operation. When the train operation is run, the `Variables` are updated.

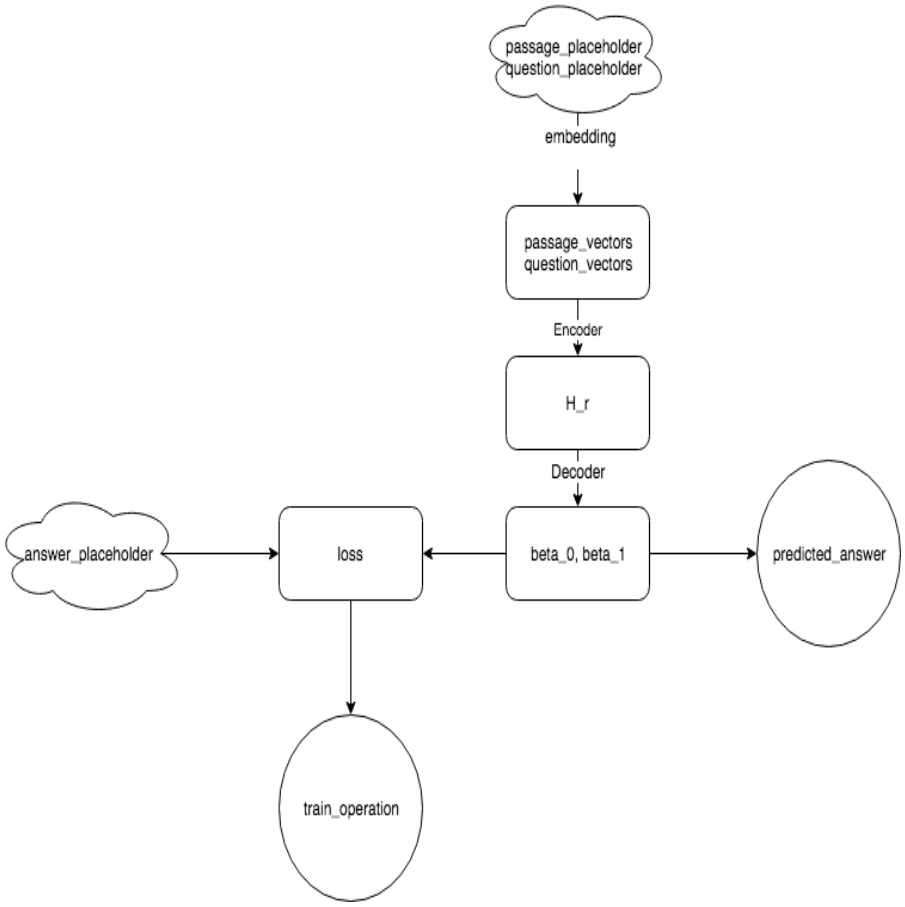

Figure 4.1: Concept of the Tensorflow graphs used in this project

Figure 4.1 describes the concept of the Tensorflow graphs for five different models used in this project. The cloud shapes represent the `Placeholders`. The `Variables` are included in `Encoder` and `Decoder`. The rectangles represent nodes in the calculation process. The circles represent the final outputs used in training or testing.

We now describe how to build a Tensorflow graph: In the embedding step, the model transforms word tokens into word feature vectors using a given word feature vector matrix. This matrix is part of the graph and is constant. The `Encoder` encodes passage word feature vectors and question feature vectors to H^r. The `Decoder` decodes H^r to two probability distributions $_0$ and $_1$. Recall that $_0$ tells how likely each token in a passage is the start of the answer and $_1$ tells how likely each token in a passage is the end of the answer. The loss is calculated using $_0$, $_1$ and ground truth answers fed through answer `Placeholders`. From the loss, a train operation is calculated.

The run time works as follow: In the training and validation process, the passages, questions and answers are fed into the graph. Running the train operation with data fed into through `Placeholders` will update all the `Variables`. In the testing process, only passages and questions are fed into the graph. The predicted answers are determined from $_0$ and $_1$.

4.3 Implementation Pipeline

For a specific model, in the train and validation process, some Tensorflow graphs that have same structure but different values for `Variables` are saved. The validation loss is used to choose the best graph. Then the best graph is used to do testing.

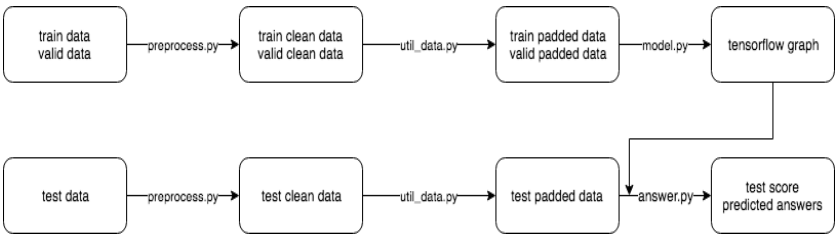

Figure 4.2: Implementation pipelines

CHAPTER 5

EXPERIMENTS

5.1 Data

The Stanford Question Answering Dataset (SQuAD) is used to do experiments. The GloVe word feature vectors[PSM14] are used to initialize words.

In the first step, a Python natural language processing library `nltk` is used to tokenize raw data into passage word token sequences, question word token sequences and answer spans. Each answer span includes a start index and an end index. In total, our training set contains 78,839 instances, our validation set contains 8,760 instances, and our test set contains 10,570 instances.

Table 5.1: Data sets

Set Name	Number of Instances
Train	78,839
Validation	8,760
Test	10,570

The vocabulary is then made from the word token sequences from the training set and validation set. After the vocabulary is made, each word token is turned into its corresponding index in the vocabulary. Two special indices are used to represent the unknown token and the padding token. Before feeding index sequences into the Tensorflow graph, the index sequences are padded to fixed lengths, as mentioned in Section 4.1. The whole training data set is then split into mini batches to support stochastic gradient decent.

After the vocabulary is made, a smaller word feature vector matrix is made from the original GloVe word feature vectors. The smaller word feature vector matrix only includes vectors of words in the vocabulary. The unknown token is assigned an average of all the vectors of known tokens. The padding token is assigned a zero vector. The index of each token in the smaller word feature vector matrix is same as that in the vocabulary.

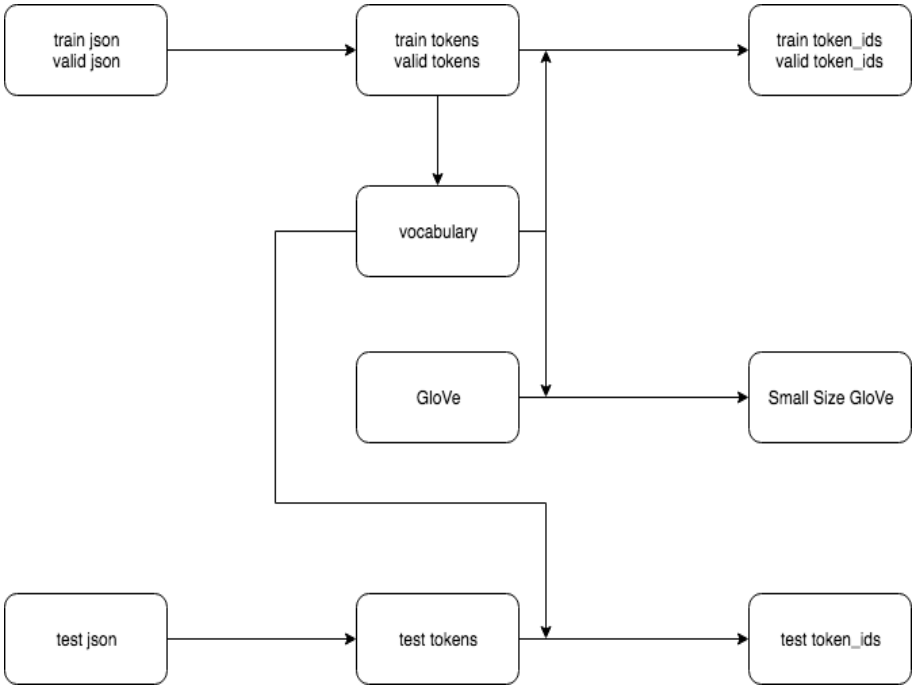

Figure 5.1: Data pipelines

5.2 Settings

Figure 5.2 shows the distribution of passage lengths. Figure 5.3 shows the distribution of question lengths. Based on the distributions, 400 is set as *passage_padding_length* and 30 is set as *question_padding_length*. We use the GloVe word feature vectors with size 100. The size of the LSTM state in the

preprocessing layer, which is l in theoretical model, is set at 64. The regularization scale of L2-regularization is set at 0.001. The batch size is set at 32. The adam optimizer is set using the default settings of Tensorflow. The normalization boundary to clip gradients is set at 5. 200 sample instances from training set are used to estimate training accuracy. 200 sample instances from validation set are used to estimate validation accuracy. The learning rate is selected through several experiments shown in Figure 5.4. Based on these experiments, the learning rate is set at 0.002.

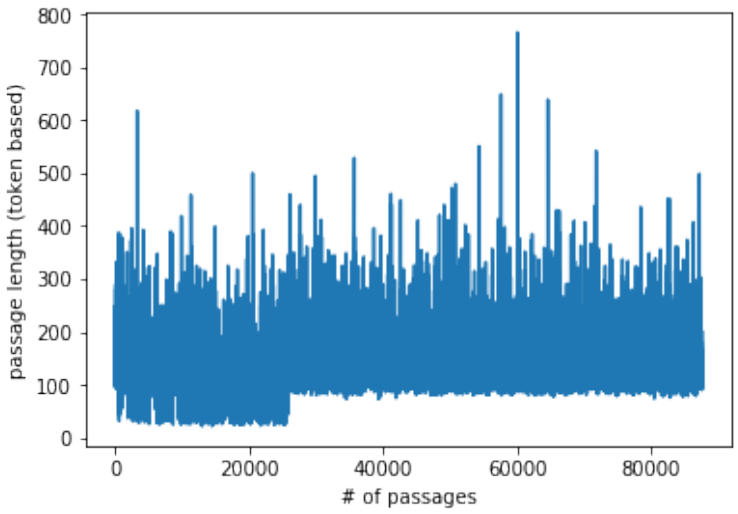

Figure 5.2: Distribution of passage lengths

The F1 score and the exact match score are used to evaluate the performance of each model. F1 treats a predicted answer and a ground truth as bag of words and calculate a harmonic average of precision and recall; exact match measures the percentage of predictions and ground truths that are exactly the same. The testing data contains several ground truth answers for one passage-question pair. The best score is chosen as the final score.

A machine that has Tesla K80 12 GB Memory, 61 GB RAM and 100 GB SSD

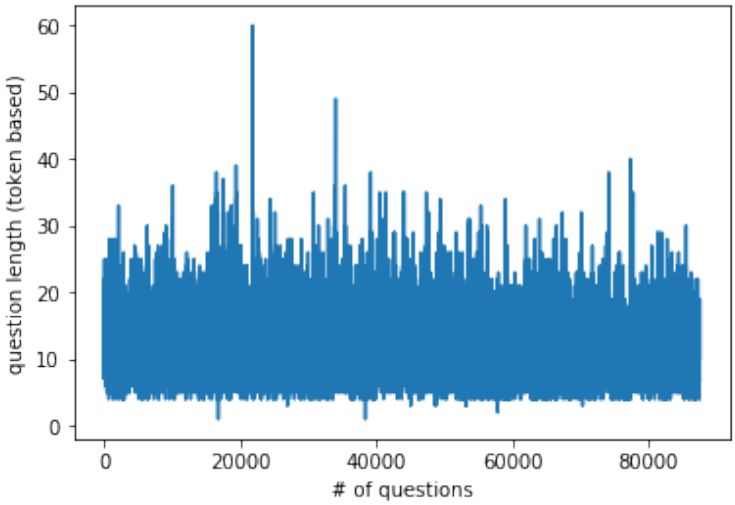

Figure 5.3: Distribution of question lengths

Table 5.2: Experimental settings

Hyperparameter Name	Value
Word Feature Vector Dimension (d)	100
Hidden State Size (l)	64
L2_regularization Scale	0.001
Hidden State Size (l)	64
Batch Size	64
Passage Length	400
Question Length	30
Clip Norm	5
Learning Rate	0.002

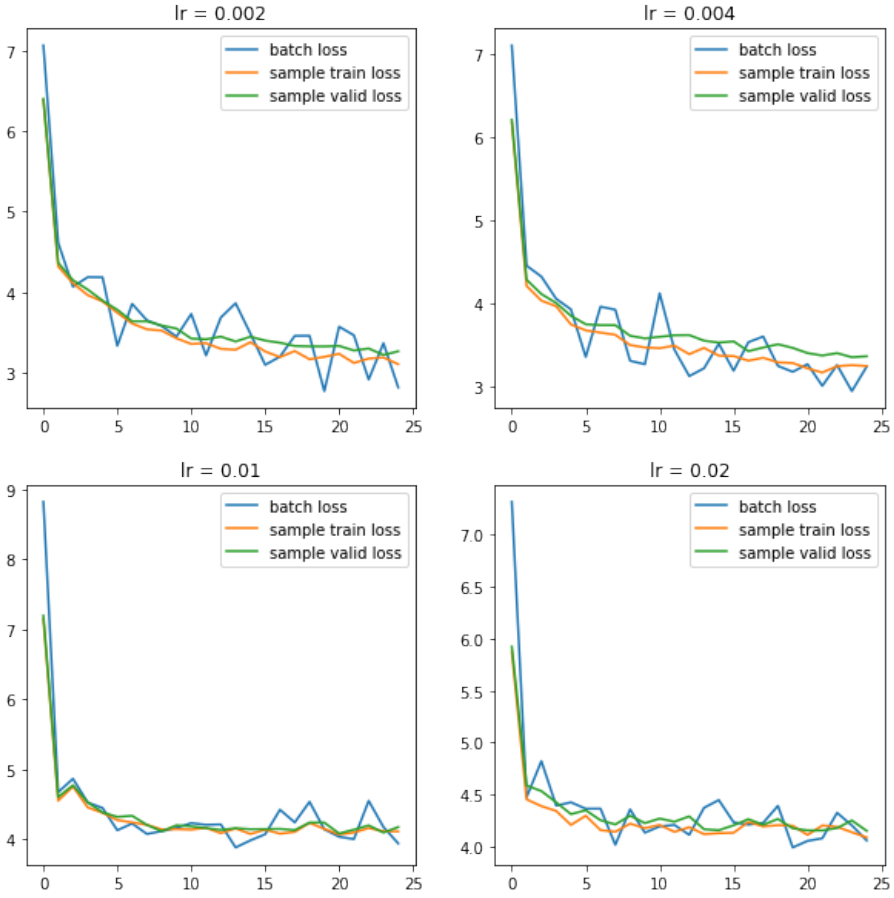

Figure 5.4: Performance of different learning rates

is used to train the models.

5.3 Results

5.3.1 Training Process

Figure 5.5, 5.6, 5.7, 5.8 and 5.9 show the training process of five different models. One epoch contains roughly 25 * 100 mini batches. The training loss and training scores are calculated every 100 mini batches using the 200 sample instances from training set. We do the same for validation loss and validation scores. Training one epoch takes roughly 100 minutes. A thorough training of each model requires around 10 epochs and takes around 17 hours.

As indicated by Figure 5.5, Model 1 converges after three epochs. The training F1 score converges to around 0.5, the training exact match score converges to around 0.35, the validation F1 score converges to around 0.3, and the validation exact match score converges to around 0.2.

As indicated by Figure 5.6, Model 2 keeps learning in 10 epochs. The training F1 score converges to around 0.6, the training exact match score converges to around 0.5, the validation F1 score converges to around 0.4, and the validation exact match score converges to around 0.3.

As indicated by Figure 5.7, Model 3 converges after 10 epochs. The training F1 score converges to around 0.65, the training exact match score converges to around 0.45, the validation F1 score converges to around 0.4, and the validation exact match score converges to around 0.3.

As indicated by Figure 5.8, Model 4 converges after four epochs. The training F1 score converges to around 0.6, the training exact match score converges to around 0.4, the validation F1 score converges to around 0.4, and the validation

exact match score converges to around 0.2.

As indicated by Figure 5.9, Model 5 keeps learning in the 10 epochs. The training F1 score converges to around 0.5, the training exact match score converges to around 0.35, the validation F1 score converges to around 0.3, and the validation exact match score converges to around 0.2.

5.3.2 Testing Results

Table 5.3: Testing results

Model	Test Exact Match	Test F1
Reference Paper	64.7	73.7
Model 1	23.4	33.6
Model 2	33.0	45.8
Model 3	33.0	46.2
Model 4	33.0	45.6
Model 5	24.3	33.9

Table 5.3 shows the testing results of all models. The reference paper gets F1 score 73.7 and exact match score 64.7 [WJ16]. Model 1 gets F1 score 33.6 and exact match score 23.4. It does not reproduce the scores of the reference paper. Model 2, 3 and 4 behave similarly with F1 score around 46 and exact match score around 33. Model 5 behaves worse than the other three experiments with F1 score 33.9 and exact match score 24.3.

5.4 Analysis

Comparing the test results of Model 1 with that of the reference paper, the difference is quite surprising. To find out why my implementation does not reproduce the results of the original paper, further debugging and parameter tuning are needed.

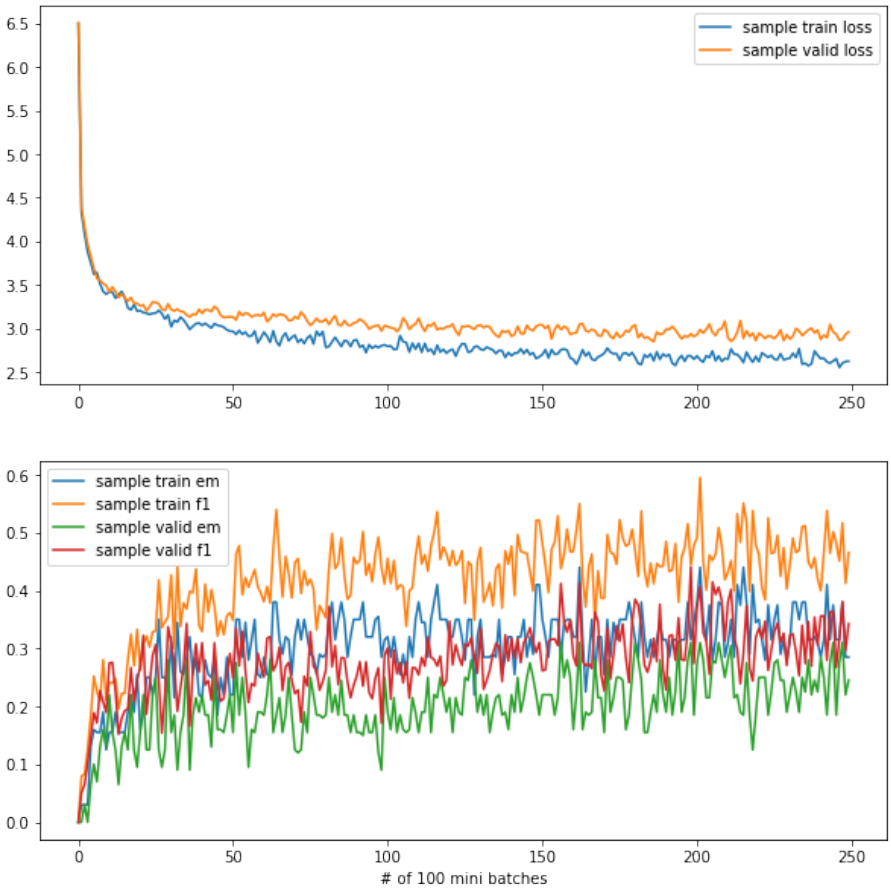

Figure 5.5: Training process of Model 1

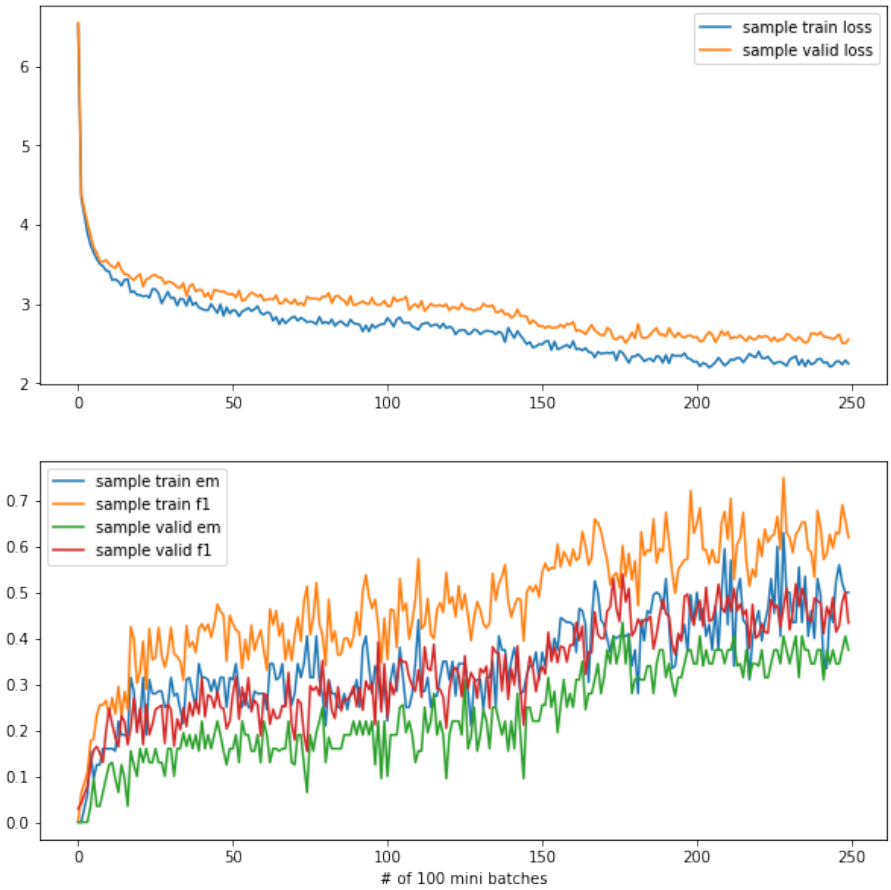

Figure 5.6: Training process of Model 2

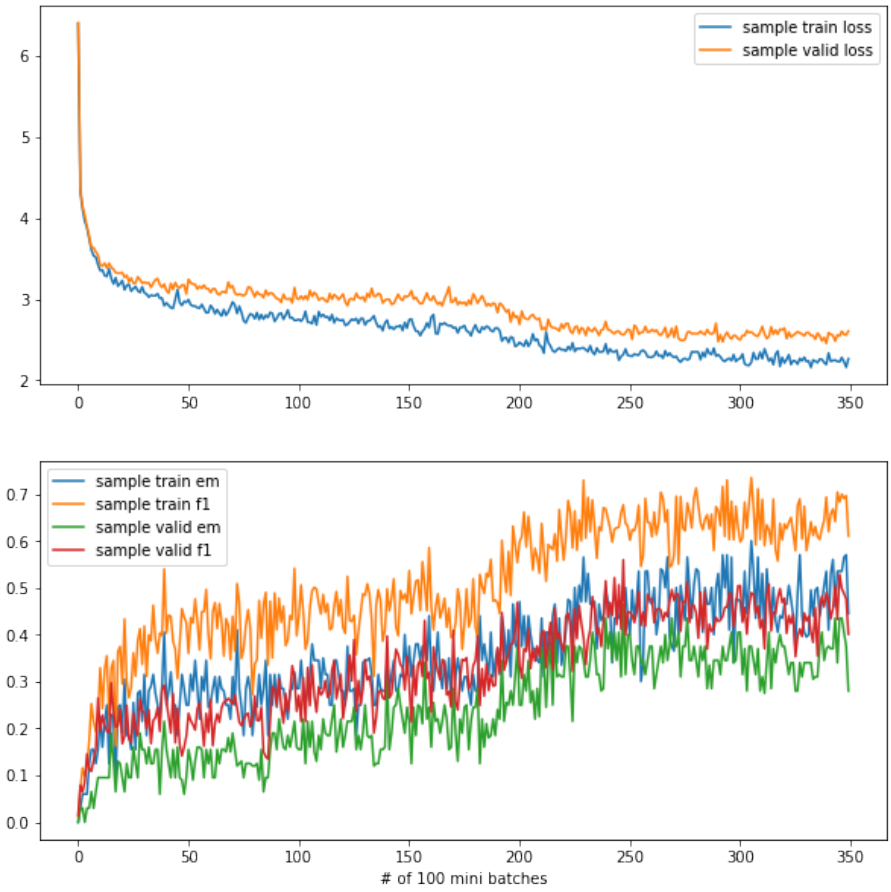

Figure 5.7: Training process of Model 3

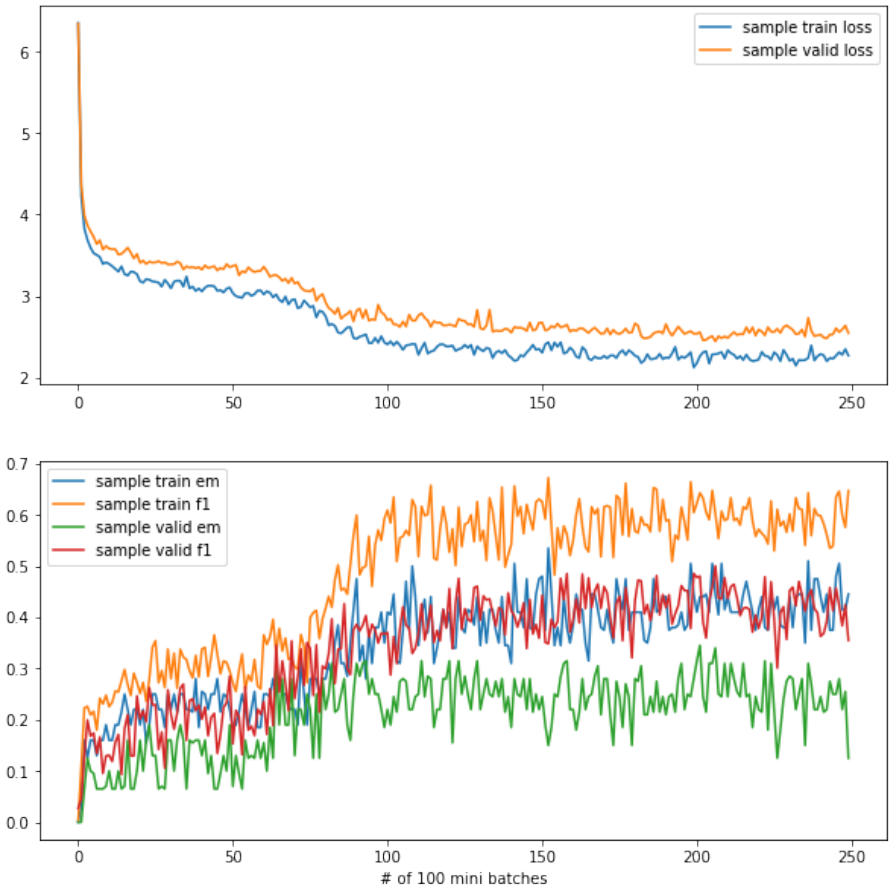

Figure 5.8: Training process of Model 4

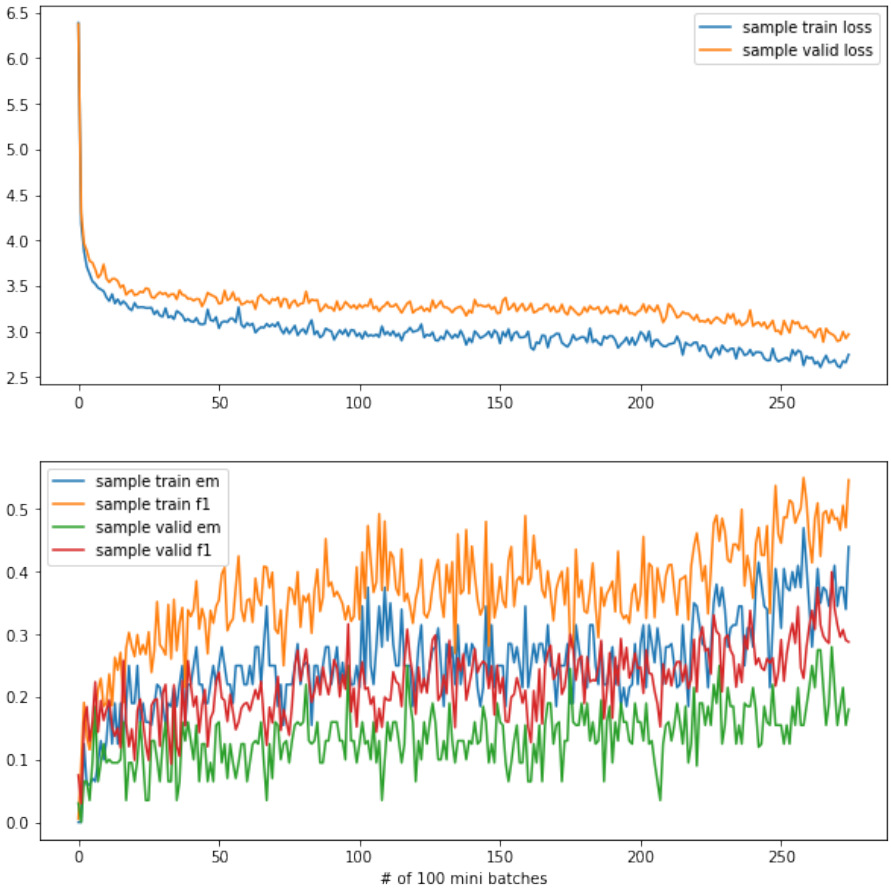

Figure 5.9: Training process of Model 5

Comparing the testing results of Model 2 and Model 1, the scores of Model 2 are better. As mentioned in Section 3.2, Model 2 uses previous attention vector to query the current attention weight vector. However, Model 1 uses the current answer LSTM state to query the attention weight vector. The answer LSTM state is produced by applying a non-linear transformation on the attention vector. It turns out not including the non-linear transformation gives better results.

Model 2, 3 and 4 behave similarly. This means removing either the preprocessing layer or the h_r in the bidirectional match LSTM layer does not decrease test results. A reasonable guess is the two parts provide duplicate context information.

Model 5 performs worse than Model 2, 3 and 4. This means removing both the preprocessing layer and the h_r in the bidirectional match LSTM layer decreases testing results. A reasonable guess is the context information provided by these two parts is not provided in other parts of Model 1. As such, one of the two must be kept.

CHAPTER 6

CONCLUSION

This project presented a thorough implementation of a question answering system. Five different models were tried and several interesting observations were found.

Further work is required to find out why Model 1 failed to reproduce the testing results of the reference paper. At the same time, more parameter tuning work is required to make the experiments more precise. Last but not the least, making novel architectures to bypass the state-of-art results is always a good way to move the question answering research forward.

BIBLIOGRAPHY

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio, *Neural machine translation by jointly learning to align and translate*, arXiv preprint arXiv:1409.0473 (2014).

[BDVJ03] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin, *A neural probabilistic language model*, Journal of machine learning research **3** (2003), no. Feb, 1137–1155.

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio, *Learning phrase representations using rnn encoder-decoder for statistical machine translation*, arXiv preprint arXiv:1406.1078 (2014).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber, *Long short-term memory*, Neural computation **9** (1997), no. 8, 1735–1780.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher Manning, *Glove: Global vectors for word representation*, Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), 2014, pp. 1532–1543.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams, *Learning representations by back-propagating errors*, nature **323** (1986), no. 6088, 533.

[RZLL16] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang, *Squad: 100,000+ questions for machine comprehension of text*, arXiv preprint arXiv:1606.05250 (2016).

[VFJ15] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly, *Pointer networks*, Advances in Neural Information Processing Systems, 2015, pp. 2692–2700.

[WJ16] Shuohang Wang and Jing Jiang, *Machine comprehension using match-lstm and answer pointer*, arXiv preprint arXiv:1608.07905 (2016).