Machine Comprehension using SQuAD and Deep Learning

Josh King
jking9@stanford.edu

Ajay Mandlekar
amandlek@stanford.edu

Filippo Ranalli
franalli@stanford.edu

Codalab Group
cs224n-fil-josh-ajay
user submission: franalli

Abstract

There are many state-of-the-art deep learning architectures that have been used to develop reading comprehension systems using the Stanford Question Answering Dataset (SQuAD). In this work, we provide an overview of some of the state-of-the-art models and compare their performance across several variations of encoder and decoder architectures and different attention mechanisms. We take inspiration from these state-of-the-art models and develop our own hybrid architectures, which have performed very similarly to previous models in the literature such as Match-LSTM and Dynamic Coattention when excluding extensive hyperparameter searches. Across all the architectures experimented with, our custom Stacks-on-Stacks (SoS) model yielded the best results of 0.537 F1 and 0.407 EM on the SQuAD test set.

1 Introduction

Machine Comprehension (MC) is a challenging problem in artificial intelligence that consists of finding the answer (assumed to be contiguous) to a given question in a context paragraph of arbitrary length. This problem has generated considerable interest and led to the development of many novel deep learning architectures. Most models make use of an encoder-decoder mechanism analogous to those used in machine translation. These models start with pre-trained word embeddings that allow for a dense representation of the question and paragraph and generate an answer span for the location of the answer in the context paragraph. Most models generate two probability distributions over the paragraph words - one for the start index of the answer and the other for the end index of the answer, usually through a mechanism that conditions the end probabilities over the start probabilities such that $P = \{P(a_s), P(a_e \mid a_s)\}$.

A wide variety of models of increasing complexity will yield different F1 and EM (exact match) scores depending on the particular encoding and decoding mechanisms implemented. One of the simplest models generates separate encodings of the question and the paragraph through a single-directional RNN with an LSTM cell, and decoding with a multi-layer feed-forward neural net predicting the start and end probabilities with no conditional relation. Such a model results in validation F1 scores that are no higher than 0.30, but it made for a good starting point to ensure that the training pipeline was working correctly. More complex models that we implemented such as Match-LSTM [2], Dynamic Coattention Network [4], Multi-Perspective Context Matching [3], and our very own Stacks-on-Stacks (SoS) build on the same encoder-decoder principles, but add complexity to the model by capturing question-to-paragraph relations through various attention and filtering mechanisms, and vertically and horizontally stacking RNNs both in the encoder and decoder to capture conditional dependencies. The modularity of the encoder, decoder, filtering and attention components allowed us to experiment with different architectures, and this led to the development of the Stacks-on-Stacks (SoS) model.

2 Data

The SQuAD data on which the models were trained consists of roughly 87k triplets of questions, paragraphs and answer spans. The data is provided both in string format for the questions and paragraphs, and as indexes mapped by a vocabulary of 400,000 tokens. The start and end predictions in the answer spans are provided as positional integers indexing into the paragraphs. The data set was split into a 95% used exclusively for training purposes, and a remaining 5% used for validation. The model was finally tested on a set of 11k triplets. The training data lengths distributions in Fig.(1) are useful for determining what a reasonable cut-off for input paragraphs and questions may be. The questions lengths averaged at about 10 up to a maximum of 60, making 30-40 a reasonable cut-off. The paragraph lengths on the other hand averaged at 135 and maxed at 766, making 400 the ideal cut-off with over 95% of the data distribution below such threshold.

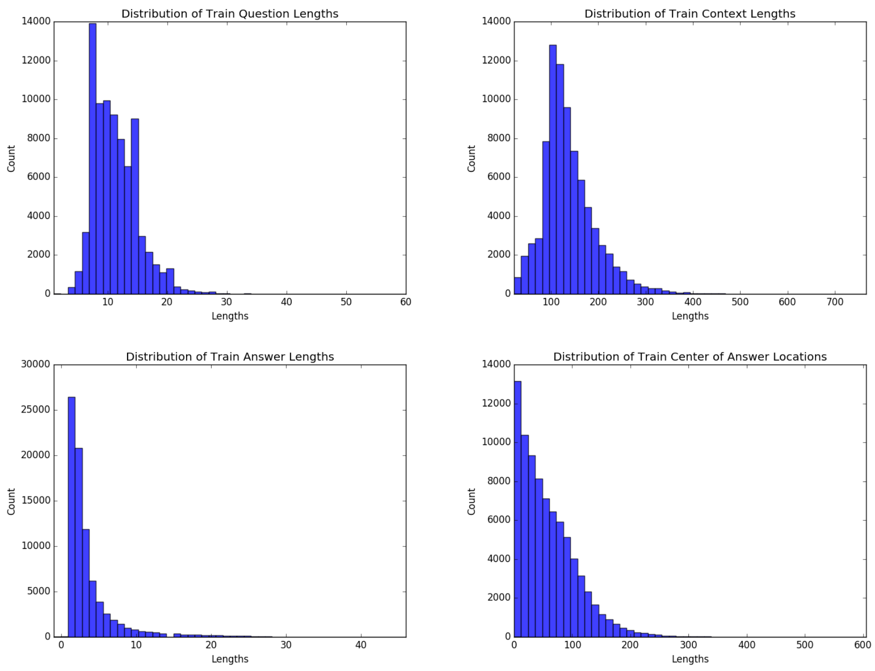

Figure 1: Training data distribution

3 Related Work

Note: we use d as the embedding size, l as the hidden state size, m as the maximum length of a context paragraph, and n as the maximum length of a question.

3.1 Match-LSTM

We used the Match-LSTM model encoder [2] and did not implement their decoder. The encoder consists of a basic LSTM encoding layer and the Match LSTM layer.

LSTM Encoding Layer

This layer takes the question embeddings $\mathbf{Q} \in \mathbb{R}^{d\times n}$ and the paragraph embeddings $\mathbf{P} \in \mathbb{R}^{d\times m}$ and encodes them using single-directional LSTMs. We collect the hidden states of each LSTM as the encodings, $\mathbf{H^q} \in \mathbb{R}^{l\times n}$ and $\mathbf{H^p} \in \mathbb{R}^{l\times m}$ respectively.

Match-LSTM Layer

This layer essentially uses a modified LSTM cell (a Match-LSTM cell) and runs this Match-LSTM over the paragraph encodings $\mathbf{H^p}$. At position i of the context paragraph, the Match-LSTM cell uses the current input $\mathbf{h_i^p}$, the previous state of the Match-LSTM $\mathbf{h_{i-1}^M}$, and all of the question encodings $\mathbf{H^q}$ in order to generate the next hidden state $\mathbf{h_i^M}$. First, the cell computes attention over the question encodings.

$$\overrightarrow{\mathbf{G}}_i = \tanh(\mathbf{W^q H^q} + (\mathbf{W^p}\overrightarrow{\mathbf{h}}_i^{\mathbf{p}} + \mathbf{W^M}\overrightarrow{\mathbf{h}}_{i-1}^{\mathbf{M}} + \mathbf{b^p}))$$
$$\overrightarrow{\alpha}_i = \text{softmax}(\mathbf{w}^T\overrightarrow{\mathbf{G}}_i + b),$$

where $\mathbf{W^q}, \mathbf{W^p}, \mathbf{W^M} \in \mathbb{R}^{l\times l}, \mathbf{b^p}, \mathbf{w} \in \mathbb{R}^l$ and $b \in \mathbb{R}$ are learned weights. We have omitted tiling of vectors in the equations above - broadcasting across addition operations is implied. The attention vector $\overrightarrow{\alpha}_i$ is then used to take a linear combination of question encodings to form a context vector, which is then concatenated with the original paragraph encoding input vector $\overrightarrow{h}_i^p$

$$\overrightarrow{\mathbf{z}}_i = \begin{bmatrix} \overrightarrow{\mathbf{h}}_i^p \\ \mathbf{H^q}\overrightarrow{\alpha}_i^T \end{bmatrix}$$

The concatenated input vector $\overrightarrow{\mathbf{z}}_i$ is then fed along with the previous hidden state $\mathbf{h_{i-1}^M}$ to a standard one-directional LSTM cell, which returns the next hidden state $\mathbf{h_i^M}$. This defines the Match-LSTM cell. The hidden states $\mathbf{H^M}$ of this Match-LSTM are then fed as inputs to the decoder. We do not use the same decoder architecture as the authors. Instead, we use a simple architecture which uses an LSTM and a linear layer over the corresponding hidden states to generate a start index probability distribution. The hidden states of this LSTM are fed as input to a second LSTM, whose hidden states are fed into a linear layer to generate the end index probability distribution. The predicted start and end indexes are the ones that maximize the probability distribution.

3.2 Dynamic Coattention

The Dynamic Coattention Network (DCN) [4] uses a coattention mechanism to encode the question and paragraph into one representation and uses a dynamic decoder that iteratively estimates the start and end indexes using an LSTM and a Highway Maxout Network (HMN). We implemented two versions of this model. In the first version we used the coattention encoder to produce encodings $\mathbf{U}$, but we used a simple decoder architecture that uses an LSTM and subsequent linear layer to generate the start index probability distribution and another LSTM and linear layer to generate the end index probability distribution. The LSTMs are vertically stacked so that the hidden states from the start LSTM decoder are fed as the input of the end LSTM decoder. In the second version of the DCN model, we used the coattention encoder to encoder the question and paragraph and a dynamic decoder similar to the original described in the paper, but using a 2-layer MLP instead of an HMN to reduce the implementation complexity of the model.

LSTM Encoding Layer

This layer takes the question embeddings $\mathbf{Q}$ and the paragraph embeddings $\mathbf{P}$ and encodes them using a shared single-directional LSTM. We collect the hidden states of the LSTM to form encodings for the paragraphs and the questions. We also add a sentinel vector to each encoding matrix which captures information that is not contained in any of the words. Finally, for the question encodings, we pass them through a single fully connected layer with a tanh activation. This is done to introduce additional variation between the question encodings and paragraph encodings. Thus, we end up with the question encodings $\mathbf{H^q} \in \mathbb{R}^{l\times(n+1)}$ and the paragraph encodings $\mathbf{H^p} \in \mathbb{R}^{l\times(m+1)}$.

Coattention Layer

We first compute an affinity matrix between the paragraph and question encodings $\mathbf{L} = \mathbf{H^p}^T \mathbf{H^q}$. We then normalize all of the columns of the affinity matrix using a softmax to produce attention scores over the paragraph encodings per question word, $\mathbf{A^q}$. We also normalize all of the rows of the affinity matrix using a softmax to produce attention scores over the question encodings per paragraph word, $\mathbf{A^p}$.

$$\mathbf{A^q} = \text{softmax}(\mathbf{L})$$
$$\mathbf{A^p} = \text{softmax}(\mathbf{L}^T)$$

Next, we generate context vectors that summarize the paragraph encodings per question word. This is just a linear combination of the paragraph encodings with weights given by the attention scores.

$$\mathbf{C^q} = \mathbf{H^p} \mathbf{A^q} \in \mathbb{R}^{l \times (n+1)}.$$

We also compute similar context vectors that summarize the question encodings per paragraph word via the matrix multiplication $\mathbf{H^q A^p}$. However, to achieve a co-dependent representation, we also consider the product $\mathbf{C^q A^p}$, which, for each paragraph word, is a linear combination of the question word context vectors with weights given by the attention scores. Concatenating these two context representations results in context vectors per paragraph word that are co-dependent on the attention over the question encodings and the paragraph encodings. Thus, we have

$$\mathbf{C^p} = \begin{bmatrix} \mathbf{H^q} \\ \mathbf{C^q} \end{bmatrix} \mathbf{A^p} \in \mathbb{R}^{2l \times (m+1)}.$$

The above is defined to be the coattention context. To complete the encoding, we concatenate the above coattention context vectors with the paragraph encodings and feed them as inputs to a Bidirectional LSTM.

$$\mathbf{X} = \begin{bmatrix} \mathbf{H^p} \\ \mathbf{C^p} \end{bmatrix} \in \mathbb{R}^{3l \times (m+1)}$$
$$\overrightarrow{\mathbf{u}}_t = \text{Bi-LSTM}(\overrightarrow{\mathbf{u}}_{t-1}, \overrightarrow{\mathbf{u}}_{t+1}, \mathbf{x}_t)$$

where $\mathbf{x}_t$ is column t in $\mathbf{X}$, corresponding to the coattention representation of paragraph word t. Our final coattention encodings are given by $\mathbf{U} \in \mathbb{R}^{2l \times m}$.

Dynamic Decoder Layer

We now describe the decoder architecture. This decoder proceeds iteratively using an LSTM. At iteration i, it starts with a previous estimate of the start index and end index probability distributions $\overrightarrow{\alpha}_{i-1}$ and $\overrightarrow{\beta}_{i-1}$, and a previous LSTM state $\overrightarrow{\mathbf{h}}_{i-1}$. By maximizing over the distributions, we obtain estimates for the start and end index of the answer, s_{i-1} and e_{i-1}. We then use these indices to index into the encodings $\mathbf{U}$ and feed $\left[\overrightarrow{\mathbf{u}}_{s_{i-1}}; \overrightarrow{\mathbf{u}}_{e_{i-1}}\right]$ as the input to the LSTM cell to update the hidden state to $\overrightarrow{\mathbf{h}}_i$. Then, for every word index j, we feed $\left[\overrightarrow{\mathbf{u}}_j; \overrightarrow{\mathbf{h}}_i; \overrightarrow{\mathbf{u}}_{s_{i-1}}; \overrightarrow{\mathbf{u}}_{e_{i-1}}\right]$ into two simple 2-layer MLPs (with tanh non-linearity) to generate new start and end probabilities $\alpha_{i,j}$ and $\beta_{i,j}$ for word j. These new distributions carry over to the next iteration. We run this iterative process for 4 iterations and output the final distributions.

3.3 Multi-Perspective Context Matching

We implemented a simplified version of the Multi-Perspective Context Matching architecture [3] (although according to their Ablation results, there shouldn't be much of a difference in results). The model consists of a filter layer, an LSTM encoding layer, a matching layer, and the decoding layer.

Filter and LSTM Encoding Layers

This layer weights all of the paragraph embeddings using relevancy scores over the question embeddings in order to filter out more important paragraph words. First, a relevancy matrix is calculated using $r_{i,j} = cosine(\mathbf{q}_i, \mathbf{p}_j)$ where $cosine$ denotes the cosine distance (dot product between L2-normalized vectors).

Next, weights $r_j = \max_i r_{i,j}$ are computed for every paragraph word j, and the paragraph embeddings are then re-weighted to become $\tilde{\mathbf{p}}_j = r_j \mathbf{p}_j$. The question embeddings $\mathbf{Q}$ and the filtered paragraph embeddings $\tilde{\mathbf{P}}$ are fed as inputs to a single shared Bi-LSTM. We collect the hidden states of the question and paragraph respectively to obtain the question encodings $\mathbf{H^q} \in \mathbb{R}^{l \times n}$ and the paragraph encodings $\mathbf{H^p} \in \mathbb{R}^{l \times m}$.Note that there are two matrices, one for the forward states and one for the backward states.

Multi-Perspective Context Matching Layer

This is the key layer of the model. This layer computes matching vectors to measure similarity between the paragraph and question encodings. We work with a simplified matching layer that just uses vanilla cosine similarity between states. This matching layer has no trainable parameters. This implementation was chosen primarily for simplicity. For any given paragraph encoding vector $\mathbf{h^p}$ and question encoding vector $\mathbf{h^q}$, the matching score is given by $m = cosine(\mathbf{h^p}, \mathbf{h^q})$. The matching layer takes the encodings $\mathbf{H^p}$ and $\mathbf{H^q}$ as input. For every paragraph word j it computes a corresponding matching vector as follows. We first compute a matching score between $\mathbf{h^p}_j$ and the final question state. Next, we compute matching scores between $\mathbf{h^p}_j$ and all of the question states. We take the maximum matching score and the average matching score. We do this for the forward states and the backward states, resulting in a 6-dimensional matching vector per paragraph word. We feed these matching vector representations into the decoder.

Decoder Layer

The decoder is simple - it takes the matching vectors and runs a Bi-LSTM. The hidden states of the Bi-LSTM are fed into two feed-forward networks that generate the start and end index probability distributions.

4 New Methods

4.1 Baseline

The first model we tried was a simple baseline consisting of an LSTM encoder, a linear attention layer, and a linear decoder. Baselinetakes in the paragraph and questions indexed into an embedding matrix as $\mathbf{P} \in \mathbb{R}^{m \times l}$ and $\mathbf{Q} \in \mathbb{R}^{n \times l}$. It then runs $\mathbf{P}$ and $\mathbf{Q}$ through an LSTM encoder with hidden size l to create $\mathbf{H}^p \in \mathbb{R}^{m \times l}$ and $\mathbf{H}^q \in \mathbb{R}^{n \times l}$ respectively. An affinity matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is computed between $\mathbf{H^q}$ and $\mathbf{H^p}$ by computing $\mathbf{A} = softmax(\mathbf{H^q H^{p\intercal}})$. This gives the attention scores over the question encodings per paragraph word. We then create a context vector for each paragraph word by taking a linear combination of the question encodings, with each weighted by its attention score, $\mathbf{C^p} = \mathbf{AH^q}$. We compute a new representation, $\mathbf{C}$ of the paragraph by concatenating the context vectors per paragraph word with its encoding and running the concatenated vectors through a linear layer, i.e. $\mathbf{C} = [\mathbf{C^p}; \mathbf{H^p}]W + b$ where $W \in \mathbb{R}^{m \times l}$ and $b \in \mathbb{R}^l$ are learned parameters.

We take the new encoded representation of the paragraph, given by $\mathbf{C}$, and pass it through a simple decoder. We use a linear layer to generate the start index logits, $s = \mathbf{CW^s}$, where $\mathbf{W^s} \in \mathbb{R}^l$. We also run $\mathbf{C}$ through an LSTM, take the hidden states of the LSTM, denoted by $\mathbf{C}'$, and run those through another linear layer to generate the end index logits, $e = \mathbf{C'W^e}$, where $\mathbf{W^e} \in \mathbb{R}^l$. These logits are mapped into probabilities with a softmax function, as is standard for the models we've discussed. The answer span is then given by $a = (argmax(s), argmax(e))$.

4.2 Stacks-on-Stacks

Due to the surprisingly good performance of Baselinewe decided to make slight modifications to various parts of Baseline. We modified the encoder to be a Bi-LSTM that was horizontally stacked, i.e. the final output states of the question encoder were fed in as the initial states of the paragraph encoder. We also made the decoder a similarly stacked Bi-LSTM. We call this model Stacks-on-Stacks or SoS, (see Figure 2).

SoS starts with the paragraph and question embeddings $\mathbf{P} \in \mathbb{R}^{m \times l}$ and $\mathbf{Q} \in \mathbb{R}^{n \times l}$. It then runs $\mathbf{Q}$ through a Bi-LSTM encoder with hidden size l to create $\mathbf{H^p} \in \mathbb{R}^{n \times l}$. Then $\mathbf{P}$ is fed through a separate Bi-LSTM with hidden size l and with initial state $\mathbf{H^q_n}$ producing $\mathbf{H^p} \in \mathbb{R}^{m \times l}$, the final hidden state of the question encoding. An affinity matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is computed between $\mathbf{H^q}$ and $\mathbf{H^p}$ by computing $\mathbf{A} = softmax(\mathbf{H^q}\mathbf{H^{p^T}})$. This matrix gives attention scores over the question encoding states per paragraph word. We then create context vectors for each paragraph word by taking linear combinations of the question encoding states, with each question state weighted by its attention score. Thus, the context vectors are given by $\mathbf{C^p} = \mathbf{AH^q}$. Then, we compute a new representation, $\mathbf{C}$ of the paragraph by concatenating the context vector of each paragraph word with its encoding and passing the concatenated vector through a linear layer, i.e. $\mathbf{C} = [\mathbf{C^p}; \mathbf{H^p}]W + b$ where $W \in \mathbb{R}^{m \times l}$ and $b \in \mathbb{R}^{l}$ are learned parameters.

We then run $\mathbf{C}$ through a Bi-LSTM with hidden size l to yield $\mathbf{S_i} \in \mathbf{R}^l$ for each paragraph index i. We also run $\mathbf{C}$ through a separate Bi-LSTM with hidden size l and initial state $\mathbf{S_m}$, the final hidden state of the start index vector, to yield $\mathbf{E_i} \in \mathbf{R}^l$ for each paragraph index i. We then compute the start and end logits $s = \mathbf{S} \cdot \mathbf{W^s}$ and $e = \mathbf{E} \cdot \mathbf{W^s}$ where $\mathbf{W^s}, \mathbf{W^e} \in \mathbb{R}^l$ are learnable parameters. These logits are mapped into probabilities with a softmax function, as is standard for the models we've discussed. The answer span is then given by $a = (argmax(s), argmax(e))$.

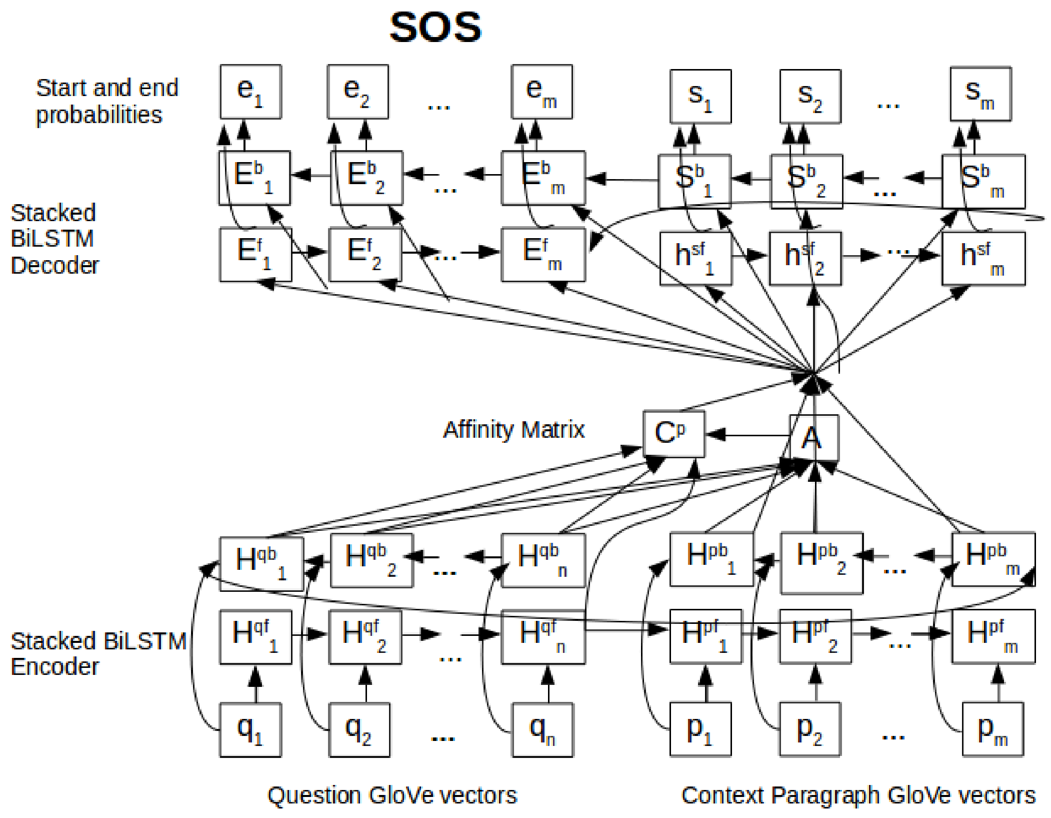

Figure 2: Stacks-on-Stacks Architecture

5 Methods

These models need word embeddings in order to generate a dense representation for the questions and paragraphs. The word embeddings can be trained using Word-2-Vec or GloVe which featurize context-level and word-level meaning for each word. The word embeddings used for this task were trimmed GloVe vectors of dimensionality $d \in \{50,100,200,300\}$, and are not trained further by the MC model. This is because training the embeddings would increase the model complexity by about 10 million parameters, dramatically increasing the training speed without concrete learning advantages. All our models were initially trained with $d = 100$, and subsequently with $d = 300$ for a 1-2% boost in the validation F1.

Furthermore, we combined learning rate annealing with embedding batch normalization, the latter of which has been proven to help the model train faster and allow faster learning [1]. Batch normalization is given by the following equation:

$$Normalized\_Batch_i = \gamma \frac{Batch_i - \mu_i}{\sigma_i + \epsilon} + \beta$$

Where γ and β are trainable parameters initialized with a value of one, μ_i and σ_i are the variance and standard deviation for each batch and ϵ is a small number for numeric stability. On Match-LSTM, batch normalization helps by about 2-3%, whereas no clear advantages are achieved on the other models we implemented.

5.1 Masking

It is imperative to apply masking, coupled with padding, in several locations of the model. First and foremost, each question and paragraph is padded with the <PAD> token up to the maximum trimmed sequence length. The actual length is then passed into the RNN's in the encoder and the decoder to ensure the network is only unrolled up to such length. Ultimately, masking is applied again to the start and end predictions in the decoder to ensure no probabilities are assigned to padded values.

5.2 Dropout

We used dropout in all of our RNNs at each output layer, and we found this to be a hyperparameter the models are very sensitive to. We experimented with values of dropout ranging from 0.10 to 0.40.

5.3 Learning Parameters

We used the TensorFlow Adam Optimizer. We used a learning rate of .001 for the majority of our models, even though we experimented with various learning rates and annealing. We also used gradient clipping to clip our gradients to a clip to a normalization of 5.0. We kept the embedding size as 200 for all our models.

5.4 Model Splicing

We tried implementing many different model architectures and splicing different parts and pieces of models together with varying degrees of success. The purpose of this section is to enumerate some parts of the model that we experimented with.

- Training the word embeddings (not recommended).
- Adding the filter layer from [3] to other models.
- Using LSTM Cells vs. GRU Cells in the Encoder and Decoder.
- Using Bidirectional vs. Single-directional RNNs in the Encoder and Decoder.
- When using Bidirectional RNNs, whether to concatenate forward and backward hidden states together or simply add them up.
- Sharing the LSTM weights when generating the question and paragraph encodings.

- Using bilinear attention over the question encodings to build a conditional encoding of the paragraph.
- Replacing the attention layer in Baselinewith the coattention mechanism from [4] and the Match-LSTM from [2].
- Using simple fully-connected decoders, vertically-stacked LSTM decoders (outputs feed into inputs), horizontally-stacked LSTM decoders (final state feeds into initial state), and dynamic decoders as in [4].

6 Results

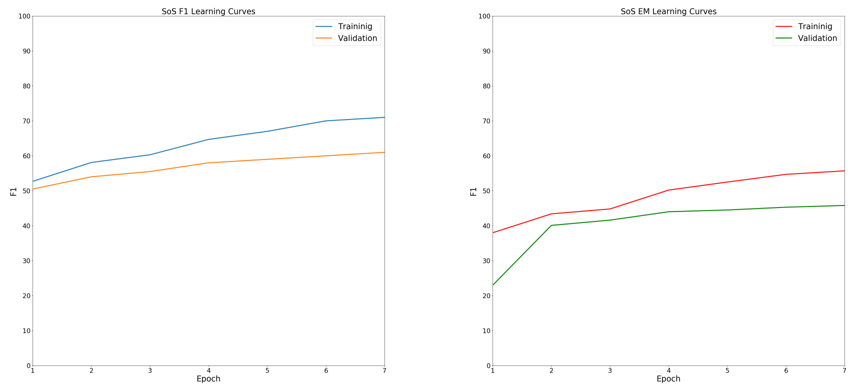

Figure 3: SoS Training Curves and Result Table

Models	Train F1	Val F1	Dev F1	Train EM	Val EM	Dev EM
SoS	0.71	0.6047	0.5311	0.557	0.458	0.3937
Multi-Perspective w/ vanilla cos similarity	0.46	0.4882	0.426	0.463	0.331	0.282
Match-LSTM Encoder w/ stacked LSTM decoder	0.72	0.57	0.5021	0.501	0.433	0.3822
DCN w/ 2-Layer MLP	0.68	0.461	0.4228	0.472	0.313	0.259
SoS with co-attention layer	0.62	0.5671	0.5012	0.46	0.4183	0.3567
Baseline 2	0.578	0.4562	0.429	0.4136	0.3058	0.293
DCN encoder w/ SoS decoder	0.618	0.544	0.4914	0.473	0.394	0.3528

Table 1: Result Summary

7 Conclusion

We evaluated the performance of many different model architectures for the Machine Comprehension problem using the SQuAD dataset. We also experimented with hybrid model architectures and this led to the model that we fondly refer to as Stacks-on-Stacks , or SoS. We believe that there is a very intuitive reason for why the SoS model achieves decent performance despite a simple architecture. The encoding step is very similar to what a human does - an initial reading of the question following by a context-aware reading of the paragraph. The attention step is also similar - it amounts to a second reading of the question and paragraph in order to narrow down focus to certain sections of the passage. The final step is to use the new understanding of the paragraph in order to select the answer, which is what the decoder does.

We saw that our SoS model outperformed the more conventional architectures such as Match-LSTM, DCN, and MPCM. It is our belief that these models are much more complex than our simple SoS model and as a result, they require much more careful hyperparameter tuning to reproduce the results cited in [2], [4], and [3]. This leads us to conclude that simpler model architectures like SoS have several benefits over more complex models, such as ease of implementation and the fact that minimal tuning is needed to achieve decent performance. In future work, we would spend more time on hyperparameter tuning for the more complex models that we implemented. We would also experiment more with different hybrid models - there are many possibilities for developing new architectures that we would have liked to explore given more time. Finally, experimenting with ensemble methods would also be a worthwhile endeavor.

References

[1] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *eprint arXiv:1502.03167*, 2015.

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

[3] Zhiguo Wang, Haitao Mi, Wael Hamza, and Radu Florian. Multi-perspective context matching for machine comprehension. *arXiv preprint arXiv:1612.04211*, 2016.

[4] Caiming Xiong, Victor Zhong, and Richard Socher. Dynamic coattention networks for question answering. *arXiv preprint arXiv:1611.01604*, 2016.