A Convolutional Network Approach to Machine Comprehension

Lisa Yan
Stanford University
yanlisa@stanford.edu
CodaLab: yanlisa

Abstract

Machine Comprehension is a daunting task, since it requires cross-encoding and exchanging information between a context paragraph and a given query in order to produce an answer span. In designing baselines for a machine comprehension model, each model training has a long turnover, which does not bode well when there is limited time to train. Long runtimes are often from implementing recurrent neural networks (RNNs), whose forward and backward passes do not make adequate use of the parallel compute units at hand. This paper discusses how to apply convolutional neural networks (CNNs) to the machine comprehension task. The author incorporates CNNs with existing bidirectional attention-flow mechanisms and compares the performance to RNN-based models. The model has been evaluated on the Stanford Question Answering Dataset (SQuAD).

1 Introduction

The machine comprehension (MC) task has recently become popular for many deep learning projects. While still a budding field, deep learning and neural networks have evolved to handle difficult tasks like machine comprehension and question answering end-to-end. As the MC task becomes more fleshed out, more datasets are published that challenge the ability of a machine to adapt. In particular, the Stanford Question Answering Dataset (SQuAD) poses the task of answering a question—or query—by parsing a given context—or paragraph—and returning a span of the paragraph that answers that question [1].

In recent work, the most popular deep learning architecture used to tackle the machine comprehension tasks has been recurrent neural network (RNN) architectures. These tasks are often phased as a three part problem: encode context within the paragraph and query independently, enforce attention that connects the paragraph to the query, and then encode the attention-aware context of the paragraph to answer the question. RNNs utilizing bidirectional long-short-term memory cells (BiLSTMs) are used for both encoding tasks to allow a single word to become aware of words before and after it.

However, while bidirectional RNN architectures like BiLSTMs provide an intuitively human approach for encoding context—after all, like a human, they scan forward and backward until they come across the answer—they have very slow performance, as they are not easily parallelizable. Training RNNs requires unrolling each step of the RNN for each iteration, and advanced parallelizable hardware like GPUs are somewhat underutilized. However, it is well known that a convolutional neural network (CNN) architecture operates *in-parallel* and therefore often performs faster than the in-series RNN architecture on GPU resources. CNNs are often used for computer vision tasks to extract higher-order, region-aware features for classification or object identification.

Therefore it is mainly intuition that motivates this project to use CNNs to produce a per-word, context-dependent representation for machine comprehension tasks. In this paper, we introduce a

hybrid CNN layer that implements what we define as a *convolutional pipeline*, merging together the pipelined RNN text-scanning architecture and CNN feature highlighting tasks in recent deep learning approaches for computer vision. We pass the output of each convolution to successively larger kernels and ultimately produce a vector *per word* that encodes information from both local and faraway contexts. We tackle the machine comprehension task with an attention model adapted from BiDAF [2], where all RNN layers are substituted with our new CNN layer.

We evaluate our machine comprehension model against BiLSTM-based BiDAF approaches on SQuAD and show that while we do not achieve performance higher than logistic regression, we at least perform better than the sliding window baseline with minimal hyperparameter tuning. Furthermore, for the same set of hyperparameters and same set of training epochs, a CNN-based model has comparative performance to BiLSTM and has much shorter training time per epoch.

2 Background

A few recent works focus on machine comprehension with SQuAD. [3, 4, 5, 2]. Almost all of these models have a three-phase processing pipeline prior to the output layer: first encode the paragraph and query separately, then apply an attention layer to the paragraph with respect to the query, then finally encode the attention-aware paragraph. The innovation of these models often comes in the attention layer, where one must make decisions on the appropriate method of combining the paragraph and query. A multi-perspective context approach uses cosine similarity [3], but we found the memory requirements of this model to be too exhaustive. We therefore use the Bi-Directional Attention Flow for Machine Comprehension (BiDAF) model, whose basic memory requirements fit within our computing resources. There are also choices for the output layer, which can output either labels per-word for their inclusion in the answer span [5], or two separate probability distributions of the start and end indices of the answer span [3].

In natural language processing tasks, RNN architectures are often preferred because they provide an intuitive model for how the neural network processes a body of text. However, there also exists various text-based tasks using CNNs, the most well-known of these architectures being character-aware embeddings [6]. CNN models exist for sentence classification [7, 8] and pair-wise embeddings of sentences [9]. However, because the machine comprehension (MC) task is relatively new, there is little work in CNNs applied to tasks which require attention [10]. Furthermore, typical applications of CNN architectures to text-based tasks pool together CNN layer output into a single vector representing the entire text passage. However, in order to use BiDAF and other attention-based MC tasks, we would like to keep per-word vectors.

3 Approach

We formally define the machine comprehension problem as follows: Given a (paragraph, question) pair defined as (P, Q), find the probability distributions $Pr(a_{st}|Q, P)$ and $Pr(a_{end}|Q, P)$ of the answer start index and end index, respectively. From these two probability distributions, generate an answer span that correctly addresses the question, or query Q, where the range of the answer comes from the paragraph, or context P. The context P and query Q have N and M words, respectively.

Our CNN-based machine comprehension model consists of six layers of multi-stage processing. After an initial processing on fixed word embeddings (Filter Layer), there are three hidden layers with learnable parameters (Contextual Representation Layer, Attention Flow Layer, and Modeling Layer) followed by a final output layer with learnable parameters. As shown in Figure 1, the main components of the architecture resemble those in the BiDAF model [2]; however, there are a few key differences in how we generate the context and query representations. We describe each component in detail below.

3.1 Word Representation Layer

The first step is to represent each word in P and Q with a d-dimensional embedding that encodes meaning of each individual word. We use fixed vector embeddings pre-trained with GloVe [11]. We

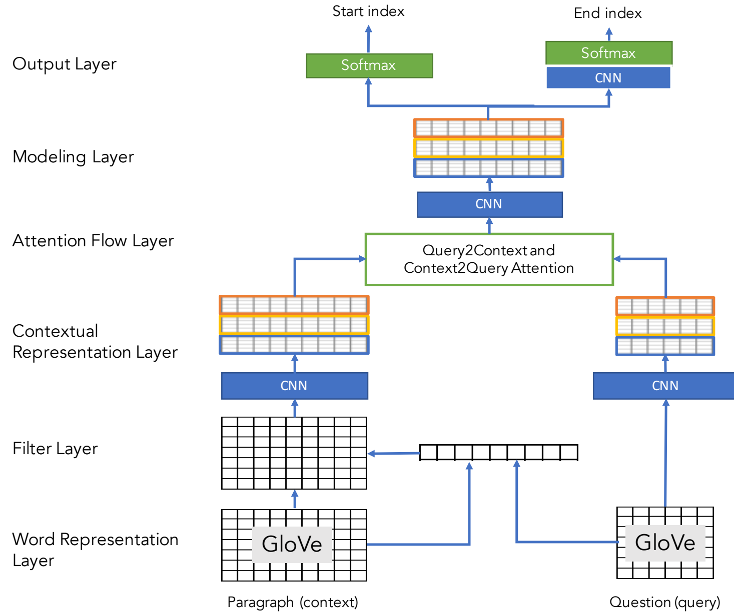


Figure 1: CNN BiDirectional Attention Flow Model (best viewed in color).

thus can represent the context paragraph P as a matrix $\hat{P} = [\hat{p_1}, \cdots, \hat{p_N}] \in \mathbb{R}^{d_e \times N}$ and the query question Q as $\hat{Q} = [q_1, \cdots, q_M] \in \mathbb{R}^{d_e \times N}$.

3.2 Filter Layer

We then filter out redundant information from the context by using a relevancy matrix, as described in Wang, et al. [3]. Each $r_{i,j}$ element in the relevancy matrix ($\in \mathbb{R}^{N \times M}$) is the cosine similarity between each pair of embedding vectors $(\hat{p_i}, q_j)$, where $\hat{p_i} \in \hat{P}$ and $q_j \in \hat{Q}$: $r_{i,j} = \frac{\hat{p_i}^T q_j}{||\hat{p_i}|| \cdot ||q_j||}$. We then calculate the relevancy degree r_i for each context word by $r_i = \max_j r_i, j$, and filter each each context word embedding $\hat{p_i}$ to produce $p_i = r_i \cdot \hat{p_i}$. The filtered context $[p_1, \cdots p_N] \in \mathbb{R}^{d_e \times N}$ is then passed onto the next layer. We chose to implement this layer because the ablation tests in Table 4 of Wang, et al. [3] showed the simple filtering provided a small boost in model performance. The query embeddings $\hat{Q}$ are not filtered.

3.3 Contextual Representation Layer

After the initial filtering, we come to our first hidden layer, which encodes the temporal interaction of the words in the paragraph context and produce a *contextual representation* for each word that incorporates information of the other words in the paragraph context. We do the same for each word in the query. In other machine comprehension models, this contextual representation layer is implemented with a bidirectional LSTM (BiLSTM) that produces forward and backward outputs of each word. However, BiLSTMs and other RNN models do not make adequate use of the state-of-the-art parallel computing, as each input must be rolled out. We therefore attempt a CNN-based approach to contextual representation.

We define a new *CNN layer* that allows each word to incorporate information from nearby and faraway. A simplified version of our generalized convolution layer is illustrated in Figure 2.

The CNN layer consists of a pipelined stage of K *convolutional steps*, in order of increasing kernel size $k \in \{2, \ldots, K\}$. At convolutional step k with input $D^{(k-1)} \in \mathbb{R}^{F_{k-1} \times T}$ (i.e., T words of dimension F_{k-1}) we apply F_k convolution filters (or kernels) to produce an output $\in \mathbb{R}^{F_k \times T}$, and we pipe the output into the next convolution step with kernel size $k + 1$. For example, suppose we use the smallest meaningful kernel size 2 and apply $F_2 = 25$ convolution filters to the paragraph

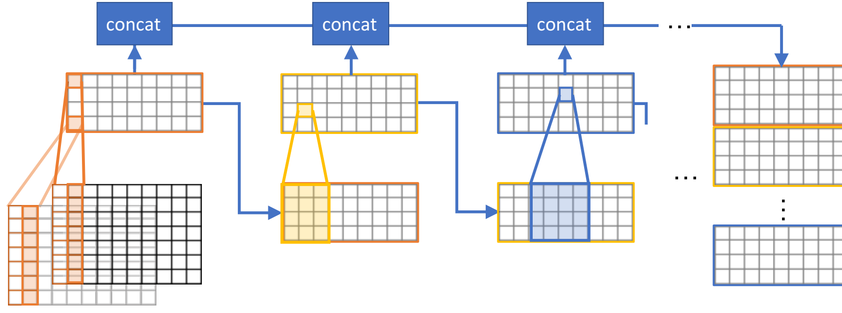


Figure 2: CNN Layer. A pipelined series of convolutions with increasing kernel size.

context matrix $[p_1 \dots p_N] \in \mathbb{R}^{d \times N}$. The output of this convolution step is a matrix in $\mathbb{R}^{25 \times N}$, which becomes the input for the next convolution step with kernel size 3.

Take $D^{(k-1)} \in \mathbb{R}^{F_{k-1} \times T}$ to be the input to a convolution step for kernel size k. The output $D^k \in \mathbb{R}^{F_k \times T}$ is calculated as follows:

1. For $j \in \{1, \dots, T\}$ take $C_{j,k}$ to be the k-sized window around the j-th input word, $C_{j,k} = D^{(k-1)}[j : j + k - 1 : *]$.
2. For each filter $W_{i,k} \in \mathbb{R}^{k \times T}$, where $i \in \{1, \dots, F^k\}$, calculate each element of the output matrix $D^k_{ij} = \tanh\big(\langle C_{j,k}, W_{i,k} \rangle\big)$, where $\langle A, B \rangle$ is the Frobenius inner product.
3. Perform regularization steps before passing D^k as the input to the next convolutional step.

These steps are repeated for each kernel size $k \in \{1, \cdots, K\}$. Note that the dimension of the input into the smallest kernel ($k = 2$) is $F_0 = d$, the dimension our per-word vectors from the previous layer. We produce the final CNN layer output by concatenating the outputs of each convolution step to produce a matrix $C \in \mathbb{R}^{\mathbf{f} \times T}$, where the resulting dimension $\mathbf{d}$ of each contextual representation is the number of filters used, or $\mathbf{f} = \sum_{k \in K} F_k$.

To create the contextual representation for the paragraph context and query, we pass the matrices $[p_1 \dots p_N] \in \mathbb{R}^{d \times N}$ and $[q_1 \dots q_M] \in \mathbb{R}^{d \times M}$ through the CNN layer to produce matrices $C^{(P)} \in \mathbb{R}^{\mathbf{f} \times N}$ and $C^{(Q)} \in \mathbb{R}^{\mathbf{f} \times M}$, respectively. In our model, we have made the hidden layer sizes consistent by setting $d = \mathbf{f}$.

The intuition behind this pipelined convolutional approach comes from image processing, where convolutions of varying kernel sizes are performed on a single image. We also borrow inspiration from the general pipelined RNN architecture: instead of having steps per word, we have steps per kernel size; instead of taking input independently , for each step The convolutional pipeline starts by intaking information from local context, or neighboring words; as the pipeline progresses, a wider and wider context window is considered. Often there is pooling in-between convolutions to reduce the dimensionality of the image, but in this case we must maintain the same number of words across convolutions. We also pipeline convolutions instead of concatenating them like in character-level CNN architectures [6] because we would like neighbors that are further than a kernel size away.

Therefore, a key point of this design is that the output of the CNN is a contextual representation *per-word*; that is, we do not aggregate to produce a single contextual representation vector for each context or query. This decision was made in order to support the existing Query2Context and Context2Query attention models that require per-word context representations for the query and context. Note that no highway networking is required[12], as the CNN layer incorporates information from earlier and later words; furthermore, highway networking only works across a single vector.

3.4 Attention Flow Layer

We share information between the paragraph and question contextual representations once more by implementing the two-way attention layer from BiDAF [2]. That is, we first produce a similarity matrix $S \in \mathbb{R}^{N \times N}$ between each paragraph word and query word. We then produce one

query vector per paragraph word i by weighting all query vectors by a softmax over the similarity column corresponding to the paragraph word, $S_i \in \mathbb{R}^M$. The resulting matrix $\tilde{U} \in \ \times \mathbb{N}$ is the context-to-query (C2Q) attention. The query-to-context (Q2C) attention matrix $\tilde{P} \in \mathbb{R}^{d \times N}$ is created in the same fashion as our relevancy degree from the filter layer, by giving the weighted sum of the most important words in the paragraph with respect to the query. Finally, we concatenate and multiple matrices together element-wise to create a matrix $G \in \mathbb{R}^{4d \times N}$, where $G_i = \left[{C_i}^{(P)}; \tilde{U}_i; {C_i}^{(P)} \circ \tilde{U}_i; {C_i}^{(P)} \circ \tilde{P}_i\right]$. For more details, please refer to Seo, et al. [2].

3.5 Modeling Layer

Once we have the matrix G containing the attention-refined paragraph representation, we then proceed with the modeling layer, which once again captures the interaction between the (attention-aware) words in the paragraph context. We have implemented this as both a BiLSTM and as another convolutional pipeline layer, and we discuss tradeoffs in our Experiments section.

The output of this layer is a matrix $M \in \mathbb{R}^{d \times T}$. This dimensionality comes naturally to the convolutional pipeline layer (recall we fix the output dimension to be $\mathbf{f} = d$). For our BiLSTM implementation, we produce forward and backward states $\in \mathbb{R}^{d/2}$ and then concatenate them together to produce a correctly-sized vector per word.

3.6 Output Layer and Loss

Finally, we produce the probability distributions $Pr(a_{st}|Q,P)$ and $Pr(a_{end}|Q,P)$ for the start and end indices, respectively, of our answer span. We take the input to be the modeling layer matrix, M. As in BiDAF [2], the probability distribution vector $\hat{y}_1 \in \mathbb{R}^N$ of the start index is a simple linear weighting by vector $w_1 \in \mathbb{R}^d$. We then pass M through a final CNN layer to produce $\tilde{M}$, which we then weight to find the end index's probability distribution vector, $\hat{y}_2 \in \mathbb{R}^N$. The loss is defined as the sum of the average cross-entropy loss functions of $(\hat{y}_1, \hat{y}_2)$ with the actual start/end indices, (y_1, y_2), over all W paragraph-question pairs.

$$\hat{y}_1 = \text{softmax}\big(w_1^\intercal M\big) \qquad \hat{y}_2 = \text{softmax}\big(w_1^\intercal \tilde{M}\big)$$

$$J(\theta) = -\frac{1}{W}\sum_{w=1}^{W}\big(y_1^{(w)} \log \hat{y}_1^{(w)} + y_2^{(w)} \log \hat{y}_2^{(w)}\big)$$

3.7 Answer Generation

To predict the substring corresponding to an answer, we select the start and end indices with the highest joint probability, where we assume independence; that is, we choose start and end indices s and e that satisfy $\max_{s,e}\left[\hat{y}_{1,s} \cdot \hat{y}_{2,e}\right]$. We also enforce valid spans ($s \leq e$) by using dynamic programming to search the quadratic space of probabilities.

4 Experiments

4.1 Experiment Settings

We evaluate our models with the SQuAD dataset, which includes over 87,000 training instances, over 10,000 development instances, and a large hidden test set . Using the default project assignment input, we split our train set into 95% for training and 5% for validation and save the entire development set as a secondary test set. We evaluate model performance with Exact Match (EM) and F1 scoring [1].

Figure 3a summarizes the parameters of our model. We use 100-dimensional word embeddings from GloVe pre-trained on Wikipedia and Gigaword [11] and use the default processing techniques provided in the starter code (meaning we only use embeddings from our train and local val set). We fix the paragraph length to be 500 words by truncating longer paragraphs and zero-padding shorter ones. The query length is set to the max length of all queries in our dataset. The dimension

Model parameters	**Size**
Input dimension, d_e	100
Paragraph length, N	500
Query length, M	max
Hidden layer dimension, d	200
CNN: # Kernels, K	8
CNN: # Filters/kernel, F_k	25

(a) Shared parameters

Model	Dropout	Batch size	Epochs
CNN	0.2	100	10
BiLSTM	0.2	100	10
Regular BiLSTM	0.2	100	2

(b) Runtime parameters

Figure 3: (a) Model parameters (b) runtime parameters for the BiLSTM and CNN models.

of each hidden layer is fixed to 200, and we fix each CNN layer to have 8 different kernel sizes ($k \in \{2, \dots, 9\}$), each with 25 filters.

We test two model designs; the first is a CNN model, which has CNNs at both the contextual representation layer and the modeling layer, and the second is a BiLSTM model, which has CNNs at the contextual representation layer and a BiLSTM at the modeling layer. In order to speed up the runtime of the modeling layer BiLSTM, we insert another matrix $W_M \in \mathbb{R}^{4d \times d}$ to scale down the large attention flow matrix G prior to computing the BiLSTM, consequently reducing parameters. We also run a third baseline model, a "regular" BiLSTM model which uses BiLSTMs at both of these layers. All models use a CNN layer for decoding the end index. We apply a dropout of 0.2 (keep 0.8), and use a sample batch size of 100 (Figure 3b). We train the CNN and BiLSTM models for 10 epochs each; the baseline BiLSTM model is used to compare runtime performance and thus is only trained for 2 epochs.

For each convolutional step in all CNN layers, we perform batch normalization [13] prior to applying the $\tanh$ nonlinearity, and perform dropout prior to passing the output to the next convolutional step within the CNN layer. For all other layers, we insert dropout before passing to the next layer.

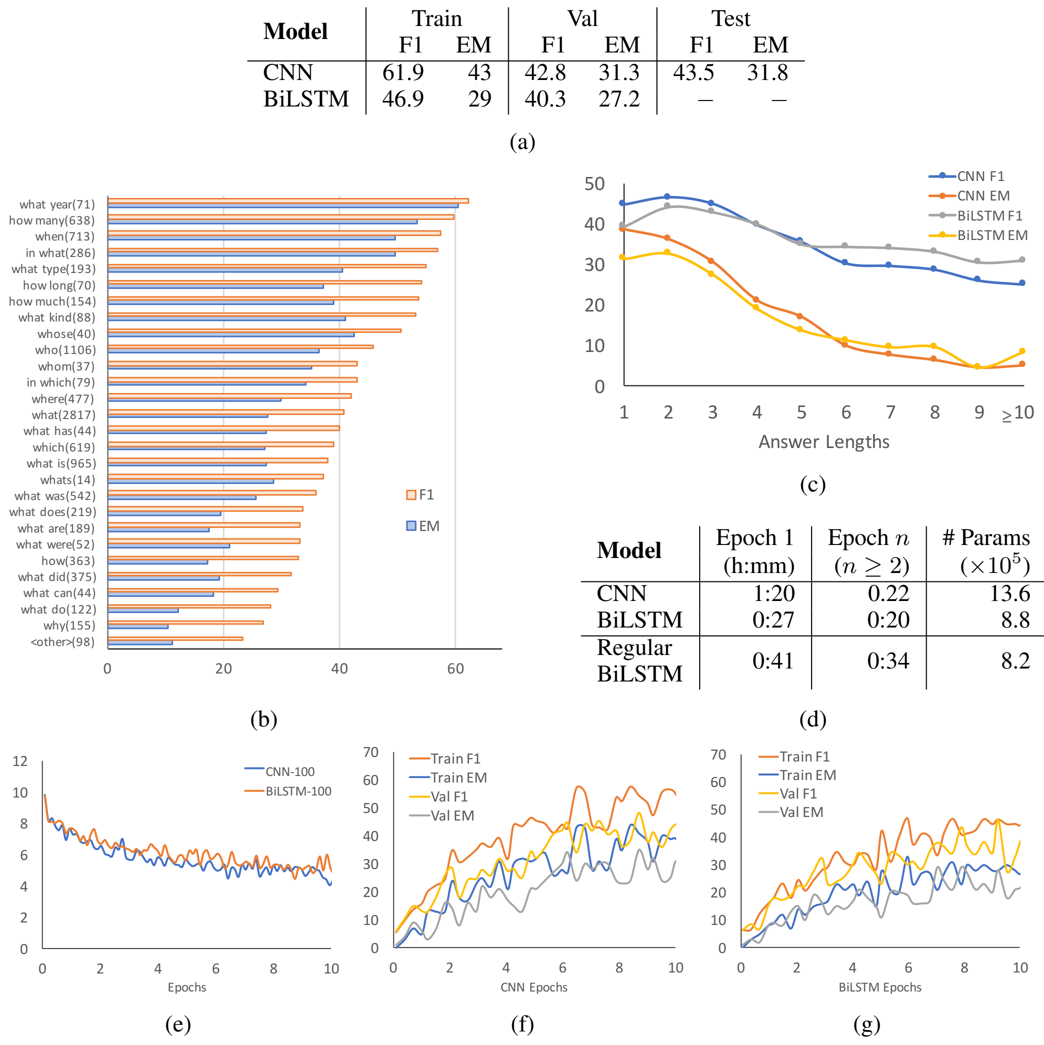
For all models, we minimize the cross entropy of the start and end probability distributions using the ADAM optimizer [14] with a learning rate of 0.0001 and global gradient norm clipping of 5. Note that Figure 3b shows that some models were only run for a few epochs; the purpose of running these models was to analyze the epoch runtime and number of parameters, and not to optimize our scores on the dataset.

4.2 Results and Analysis

We implement and run all of our experiments in TensorFlow 0.12.1 on a Microsoft Azure VM with a Tesla M60 GPU. We summarize the results in Figure 4. Figure 4a shows the SQuAD train, validation, and hidden test set performance of the CNN and BiLSTM models. We do not include the BiLSTM performance on the test set as we had limited submissions on CodaLab. These performance metrics are nothing to write home about, but they do perform better than the sliding window baseline [1]. Furthermore, we spent minimal time analyzing the structure of the SQuAD data and tweaking the hyperparameters, so we believe that data preprocessing techniques and further tuning will aid our models in becoming competitive.

Qualitative performance of the models on the validation set is shown in Figures 4b and 4c. Figure 4b shows that CNN performs best when the answer is numeric, and worst when the answer is more open-ended (when there is no question word/phrase, we classify as <other>). In addition, the EM performance is relatively close to F1 for questions asking for people or names. The BiLSTM performance is almost identical, most likely because the two models are trained on the same CNN contextual representation layer. Figure 4c shows that both the BiLSTM and CNN models perform worse as the answer span gets larger; however, BiLSTM performs marginally better, perhaps because of gating mechanisms at the attention layer.

Figure 3 illustrates the runtime of each model as compared with a "regular" BiLSTM model that uses an RNN contextual representation layer. Evidently, the CNN outperforms the BiLSTM by 2$\times$ and the regular BiLSTM by 4$\times$ per epoch. There is substantial setup time on the GPU prior to any training, which adds to the runtime of the first epoch. In particular, the more trainable parameters and parallel computing required, the longer the underlying libraries work to optimize the use of the GPU. However, as we train for longer epochs, this "start-up" time becomes negligible.



Model	Train		Val		Test	
	F1	EM	F1	EM	F1	EM
CNN	61.9	43	42.8	31.3	43.5	31.8
BiLSTM	46.9	29	40.3	27.2	–	–

(a)

Model	Epoch 1 (h:mm)	Epoch n ($n \geq 2$)	# Params ($\times 10^5$)
CNN	1:20	0.22	13.6
BiLSTM	0:27	0:20	8.8
Regular BiLSTM	0:41	0:34	8.2

(d)

Figure 4: Various illustrations of performance of the CNN and BiLSTM models (best viewed in color). (a) The F1 and EM scores for each model on the SQuAD train, validation, and hidden test sets; (b) CNN performance for different question types (frequencies of each question type shown in parentheses); (c) Runtime and parameters for our models vs. a full BiLSTM model; (d) CNN and BiLSTM performance for different answer lengths; (e) CNN performance over time; (f) BiLSTM performance over time; (g) CNN and BiLSTM loss decay over time.

Finally, we analyze the CNN and BiLSTM model performance over training epochs. As shown in Figure 4e, the loss decay over time of both the CNN and BiLSTM models closely follow each other. This is perhaps because other trainable parameters outside of the modeling layer—for example, the contextual representation layers—contribute more in the backpropagation step. The training performances of CNN and BiLSTM (Figures 4f and 4g, respectively) do not vary much. However, a gap between the training and validation performance starts to form early in the BiLSTM training. A similar gap does not occur in the CNN model, possibly because we increase the regularization with more dropout and batch normalization between convolutional steps.

5 Conclusion and Future Work

As our results show, the CNN approach greatly speeds up training time, but it is not completely apparent that they provide exactly the same performance opportunities as other models. It is necessary to perform ablation testing to fully understand the impacts of CNN on the machine comprehension

task, by incrementally replacing CNNs with BiLSTMs at each layer in our architecture, and not just at the modeling layer.

Furthermore, our training results are not stellar by any means and can definitely be improved with further hyperparameter searching. We spent a lot of time fiddling with the learning rate before remembering that the Adam optimizer automatically adapts learning rates during training. With more work time, we would like to analyze the SQuAD dataset structure more, as it seems the leading models perform heavy preprocessing on the SQuAD text to optimize and speed up the training process. Furthermore, in order to fully understand the impacts of CNN on such a task, we should perform ablation testing, where we replace a CNN with a BiLSTM at each possible layer in our architecture.

Overall, we have successfully demonstrated that incorporating a pipelined convolutional neural network can be a feasible method of encoding context-aware representations of bodies of text. More analysis is required to see whether a convolutional approach can outclass the state-of-the-art algorithms in machine comprehension, but we believe that this project is a good start. Machine comprehension is a complex, open-ended task, we hope that this paper encourages others to pursue unique architectures.

Contributions

After over 70 hours of working by myself on this project (the last 50 hours were in the past four days), I feel very accomplished! ☺

References

[1] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100, 000+ questions for machine comprehension of text. *CoRR*, abs/1606.05250, 2016.

[2] Min Joon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. *CoRR*, abs/1611.01603, 2016.

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

[4] Caiming Xiong, Victor Zhong, and Richard Socher. Dynamic coattention networks for question answering. *CoRR*, abs/1611.01604, 2016.

[5] Shuohang Wang and Jing Jiang. Machine comprehension using match-lstm and answer pointer. *CoRR*, abs/1608.07905, 2016.

[6] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. Character-aware neural language models. *CoRR*, abs/1508.06615, 2015.

[7] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[8] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *CoRR*, abs/1404.2188, 2014.

[9] Wenpeng Yin, Hinrich Schütze, Bing Xiang, and Bowen Zhou. ABCNN: attention-based convolutional neural network for modeling sentence pairs. *CoRR*, abs/1512.05193, 2015.

[10] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. *CoRR*, abs/1602.03001, 2016.

[11] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[12] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *CoRR*, abs/1505.00387, 2015.

[13] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.