Detection of equivalent questions on Quora

Sergio Ramirez Martin, Dillon Pulliam, and Abhishek Bhargava
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
`{sergior, jpulliam, and abharga2}@andrew.cmu.edu`

Abstract

In this project we explore the usage of Convolutional Neural Networks (CNNs) for the detection of semantically equivalent questions, specifically on a Quora dataset hosted on Kaggle. First we build a basic random forest classification model to test and use as a baseline. We then train one CNN on the entire dataset, and three seperate CNNs on subsets of the dataset, before testing each of these models on our test set. Finally, we explore a variety of different methods to combine these three CNNs in various ways (max voting, weighted average vote, and sigmoid unit) to determine if the combination of smaller models into one more comprehensive model performs well.

1 Introduction

For the final project of *10-701: Introduction to Machine Learning*, we chose to work on the problem of identifying duplicate (semantically equivalent) questions posted on Quora, an online Q&A forum where users can post questions and exchange ideas by responding to other people's questions. This distribution of knowledge is restricted when there are different discussion forums that are trying to answer the same question. Instead, it would be more efficient for both seekers and providers of knowledge to have all semantically equivalent questions directed to one common discussion forum. As explained by Iyer et al. [10], knowledge seekers would find all the information related to their specific question in one place, and knowledge providers would reach a wider audience with their responses. As an example of semantically equivalent questions, consider the following two sentences taken from a Quora dataset:

Sentence 1: "What Game of Thrones villain would be the most likely to give you mercy?"
Sentence 2: "What Game of Thrones villain would you most like to be at the mercy of?"

Obviously, two semantically equivalent questions like these should be on the same discussion forum. However, detecting that these sentences are semantically equivalent is not as easy for a computer as it is for a human. Building a machine learning model to detect such equivalence between questions is a difficult task due to "the ambiguity and variability of linguistic expression", as stated by Gabrilovich and Markovitch [5]. But despite such difficulties, machine learning scientists have succeeded at building models that achieve high accuracies for specific datasets (more detail in section 2).

For the purpose of this project, we will focus on a specific Quora dataset hosted on the online machine learning forum, *Kaggle*. The dataset provided is in CSV file format and consists of over 400,000 datapoints of potentially "duplicate," or semantically equivalent, questions. More specifically, in this project we will focus on three types of models to detect semantic equivalence. First we will build a Random Forest classifier as our baseline. Second, we will develop a Convolutional Neural Network (CNN), similar to the one described in Bogdanova et al. [4], as our primary model. And third, we will design various ensembles of smaller versions of these CNNs as our exploratory method.

2 Background and Related Work

There has been quite a bit of work done in duplicate question detection. Many current state-of-the-art methods involve using Recurrent Neural Networks, such as Siamese gated recurrent unit (GRU) networks [8]. However, we wanted to investigate the slightly broader question of how plausible it is to use new tools in natural language processing. In particular, we wanted to look at whether Convolutional Neural Networks are robust enough to be a powerful tool in NLP, and also whether ensemble methods would be effective in solving the problem of detecting duplicate questions.

Bogdanova et al. [4] successfully applied Convolutional Neural Networks to the problem of detecting duplicate questions. They showed that the CNN approach is plausible, and even performs better than various support vector machine approaches on the Ask Ubuntu data set. Addair [3] tried a similar approach, using the GloVe algorithm to produce word embeddings. In general, CNNs are largely untested in the NLP space, which makes them so interesting to investigate.

We also wanted to look into ensemble methods to combine classifiers. The idea is that if multiple different models reach different local minima, each classifier might learn different features of the dataset. If we could combine this knowledge from different classifiers, we might be able to produce a much better classifier overall. Many researchers have experimented with this approach before, not directly for this application, but on various other applications. For example, people have investigated using various activation functions to feed into an ensembling method [6], which is something that we looked into. We ended up choosing to use cosine similarity and feeding that into a sigmoid unit to optimize the weights on each hypothesis. Another common method used to construct ensemble classifiers is using majority vote [12]. This simply means that given a test sample, we perform inference using each hypothesis, and whichever class appears the most frequently in the resulting classifications is the final classification of the ensemble classifier.

3 Dataset

In this project, we used a public Quora dataset hosted on *Kaggle*. This dataset contains $404,351$ pairs of potentially equivalent questions (36.9% labeled as duplicate). The .csv file containing this data can be downloaded from Que [2]. The data is organized by columns as follows:

- *Column 1*: Named "id", contains integers which are the identification numbers for each question pair.
- *Column 2*: Named "qid1", contains integers which are the identification numbers for each of the first questions in the question pairs.
- *Column 3*: Named "qid2", contains integers which are the identification numbers for each of the second questions in the question pairs.
- *Column 4*: Named "question1", contains a string representing the first question in each question pair.
- *Column 5*: Named "question2", contains a string representing the second question in each question pair.
- Column 6: Named "is_duplicate", contains a binary value for each question pair (1 if the question pair is a duplicate, 0 otherwise).

We first cleaned the data by removing every character that was not alphabetical or a space (including numbers). The reasoning behind this was that our dataset contained multiple "garbage words" made of numbers and special characters that did not add any semantics to the sentence. Thus we did not need to learn word embeddings for these "garbage words". Another transformation that we did to the data was making every word lowercase. This was done for consistency in our word embeddings. For example, the words "What" and "what" should have the same word embedding representation. Making every word lowercase before creating the word embeddings ensured this consistency. Lastly, we shuffled the entire dataset before training each model.

4 Methods

4.1 Pre-Training: Word Embeddings

The first thing we did was pre-train and learn the word embeddings for all words in our dataset. This was crucial as word embeddings are more informative than string representations for Natural Language Processing problems. Word embeddings, in particular *word2vec*, is a technique used to map strings of words into a vector space. This technique is state-of-the-art, and is used in nearly all of the recent paraphrase detection papers we read, such as Bogdanova et al. [4], Wang et al. [16], and He et al. [7].

Word2Vec uses a two-layer neural network to learn what words are correlated with (typically used near) other words. The words in the vocabulary are then represented as d-dimensional vectors, where d is a chosen (not learned) hyper-parameter that represents the number of neurons in the hidden layer of the neural network. We chose to set $d = 200$. The vector representation of a word is actually just the output of the hidden layer of this neural network. Words that are considered as having similar meanings are often also close together in the vector space. This technique was used to pre-train all our models so we could transform our data and better capture the context and semantics of words.

The specific version of *word2vec* we used comes from the *gensim* library Řehůřek and Sojka [15]. More specifically, we chose to use the CBOW method while learning a 200-dimensional representation for all words. This method works by using one-hot encoding on the entire vocabulary and setting all words around a target word to 1 at the input level and all words except the target word to 0 at the output level. This method then loops through all sentences for the entire vocabulary to learn the word embedding, in this case the weights corresponding to the hidden layer.

After learning the word embeddings for all tokens in our dataset, we then used the PCA decomposition library in *sklearn* to visualize them [14]. PCA analysis was used as it allowed us to transform a 200-dimensional vector (length of our word embeddings) down to a smaller dimensional space while at the same time retaining as much variance as possible in the data. In our case, we transformed our learned word embeddings to a 2-dimensional space that could easily be visualized using the *pyplot* library [9]. Below is a visual representation of our word embeddings for a select few words in our dataset. Notice specifically how words that are close in semantic meaning, such as "facebook", "whatsapp". and "quora", are also close in the vector space:

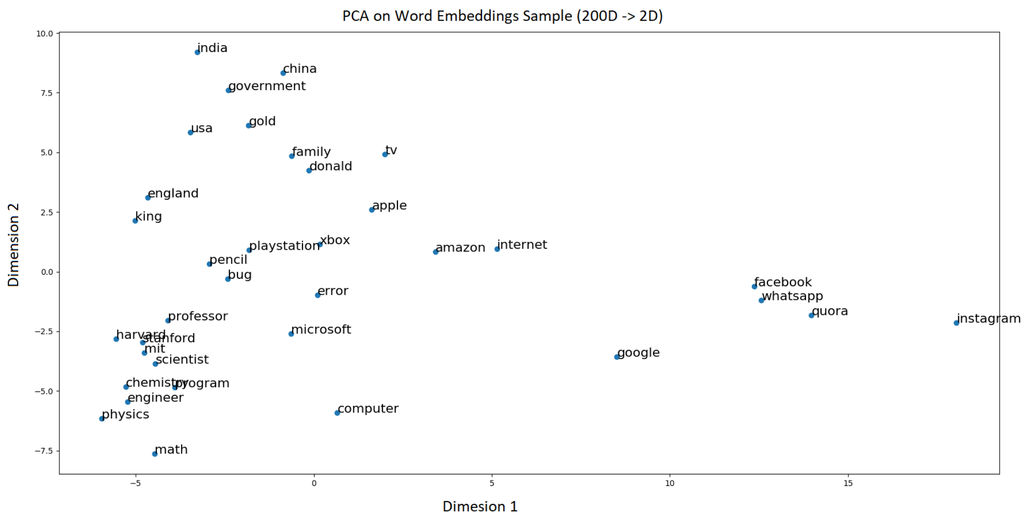

Figure 1: Learned Word Embeddings with PCA Dimensionality Reduction 200D -> 2D

4.2 Baseline Classifier: Random Forest

Before building our primary models of investigation (CNN and ensemble of CNNs), we first built a baseline Random Forest classifier to get a ballpark estimate for the type of accuracy we should seek to beat. The Random Forest model we developed followed the basic layout of the design proposed by [1]. As mentioned, the overall scheme of this model is extremely simplified. This model first splits our Quora dataset 80-20 in regard to training and test set percentages using the "train_test_split" function from *sklearn* Pedregosa et al. [14]. Using this built in feature allowed us to ensure there was no ambiguity whatsoever in the way our dataset was broken up.

After splitting the dataset, we then proceeded to train our random forest model. This was done by building two same length lists of *numpy* arrays [11] with each array containing all tokenized words in each sentence. After building these lists, we then calculated the number of common words as both a total and as a percentage for each sentence combination. For example, consider the following two sentences taken from our dataset:

Sentence 1: "On Facebook, how can I add mutual friends?"
Sentence 2: "Can I see an ordered list of the people with whom I share the most mutual friends on Facebook?"

In this case, we would calculate that there are five total words in common: "Facebook", "can", "I", "mutual", and "friends." We would also calculate that there are 37.037% words in common. This second number is computed by dividing the common word count by the average number of words in our two sentences. In essence, this transforms our two *numpy* arrays of tokenized words into a two-dimensional vector with element one storing the total number of words in common, and element two storing the percentage of common words.

After computing these two values for each sentence combination in our lists, we then created two separate random forest classifiers from the built in class developed by *sklearn* [14]. One forest was trained on the common word count value, while the other was trained on the percentage value. The specific parameters of the forests we learned are as follows: *trees in forest* = 100, *criterion* = gini score, *max depth of trees* = 1. The reasoning behind using depth of one for the trees was due to each tree only being supplied one value per data point (a common word count or a percentage). Therefore, allowing our trees to have greater depth would have been useless as there would be no other value to split on. Below is a diagram of the random forest model we used:

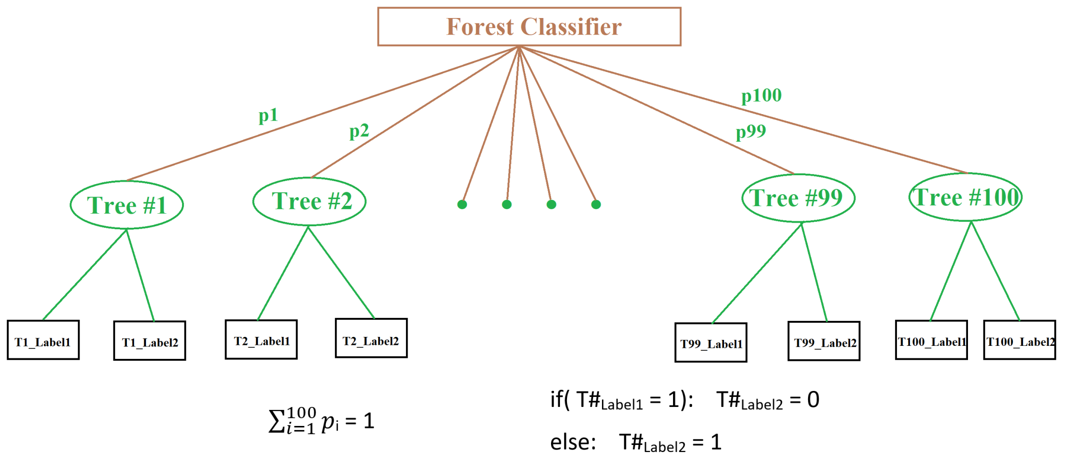

Figure 2: Random Forest Classifier

4.3 Main Model: Convolutional Neural Network (CNN)

The main model we developed tries to replicate the CNN approach provided by Bogdanova et al. [4]. As shown in Figure 3, this model takes as input the word embedding (a vector) for each word in the question pair (notice that each question is processed separately through the model until the output step). These embeddings are then concatenated into windows as follows:

$$x_i = [e_{i-\frac{k-1}{2}}, \ldots, e_i, \ldots, e_{i+\frac{k-1}{2}}]^T$$

where x_i is the window around the i^{th} word embedding, e_i is the i^{th} word embedding, and k is a chosen (not learned) hyper-parameter. In our case, we chose $k = 3$. Each window is of size $k \cdot d$, where d is the size of the word embedding vectors (200 in our case). Padding is used to complete the window around the first and last embedding. Padding should be a vector of the same size as the word embedding vectors. In our case, we chose vectors of zeros.

Once we have our windows, they are taken as input by a one-layer convolutional network. The operations that happens in this layer are as follows:

$$v_i = \sigma(Wx_i + b)$$

where v_i is the output of the convolution for x_i (the i^{th} window), W is a matrix of dimension $clu\text{X}d \cdot k$, and b is the bias vector of size clu, where clu is a hyper-parameter (300 in our case). Also, $\sigma(x) = \tanh(x)$. Both W and b are parameters to be learned and are the same for both questions.

The outputs of the convolutional layer are then added together and the $\tanh$ function is applied to the sum component-wise to obtain the vectors q_1 and q_2. These two vectors are the representation of each question after going through the model. Finally, we apply cosine similarity to these to vectors to obtain the output of the model as follows:

$$O_M = \frac{q_1 \cdot q_2}{|q_1| \cdot |q_2|}$$

If this output is close to 1, this means that the vectors q_1 and q_2 are close in the vector space and the sentences are closely related semantically. On the other hand, if the output is close to 0, the sentences are not semantically equivalent.

To train this model, we used 90% of the entire dataset. We also used stochastic gradient descent and the mean-squared error loss function to update our model:

$$M = \arg\min_{\hat{M}} \sum_{(x,y)\in D} \frac{1}{2}(y - O_M(x))$$

where M is the model, D is the trainig set, x is a question pair in the training set, y is the true label for x, and $O_M(x)$ is the model's output when given x as an input.

We used the backpropagation algorithm to update W and b. We implemented this model using the powerful capabilities of *pytorch*, a *python* library [13].

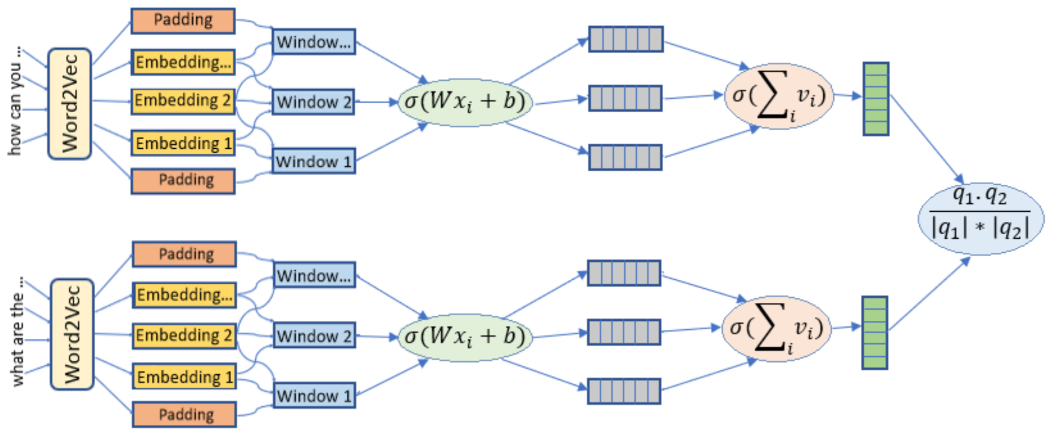

Figure 3: CNN model with cosine similarity.

4.4 Ensemble Model: Various Techniques

After developing our basic Convolutional Neural Network (CNN) and training it on the entire dataset, we then explored a variety of various ensemble techniques to try to improve our overall model accuracy. The first thing we did was split up the shuffled dataset 90-10 percent in regards to training and test sets. After this split, we then split our training set again into thirds. In other words, we had three training sets, each consisting of 30% of the entire dataset and one final test set consisting of 10% of all data points. Finally, our three separate training sets were again randomly split 90-10 percent in terms of training and test sets. These three separate training sets were then used to train three separate CNN models with each model taking the same form as in section 4.3.

After training each of our three individual CNNs, we then experimented with a variety of different methods to combine these learners. Our thinking was that if each learned to classify different types of questions as duplicate and non-duplicate, then by combining them we may perform better on the test set. In other words, we hoped that each individual CNN would learn a different local minimum.

The first and simplest ensemble technique we experimented with was a majority vote function. In this model we fed each CNN the question pair and them classify it as either duplicate or non-duplicate. We then took a majority vote of the classification prediction and used that as our ensemble prediction.

The second ensemble technique we implemented was an average function. In this model we again fed each CNN the question pair and had them classify it (note that the final layer of our CNN is cosine similarity, by classify we mean the value output from this). We then took an average of these classification predictions by adding them all together and dividing by three. This average output was then used to make the final ensemble output prediction:

$$\text{if } \left(\frac{1}{3}\sum_{i=1}^{3} CNNoutput_i > 0.5\right) \implies label = 1 \text{ ; else } \implies label = 0$$

The third and final ensemble technique we experimented with was a sigmoid unit as the final layer. For this model we used three inputs, each corresponding to one of our individual CNN outputs. We trained the weights of this final layer using the 90% training set that was also used to train each CNN. To train we randomly chose 50,000 data points from this subset and split that 90-10 percent in terms of training and test sets. These data points were used to learn the final weights of our sigmoid unit. Following is a diagram of our sigmoid ensemble CNN model (this same model applies for both majority vote and average by simply replacing the final sigmoid layer with these other functions):

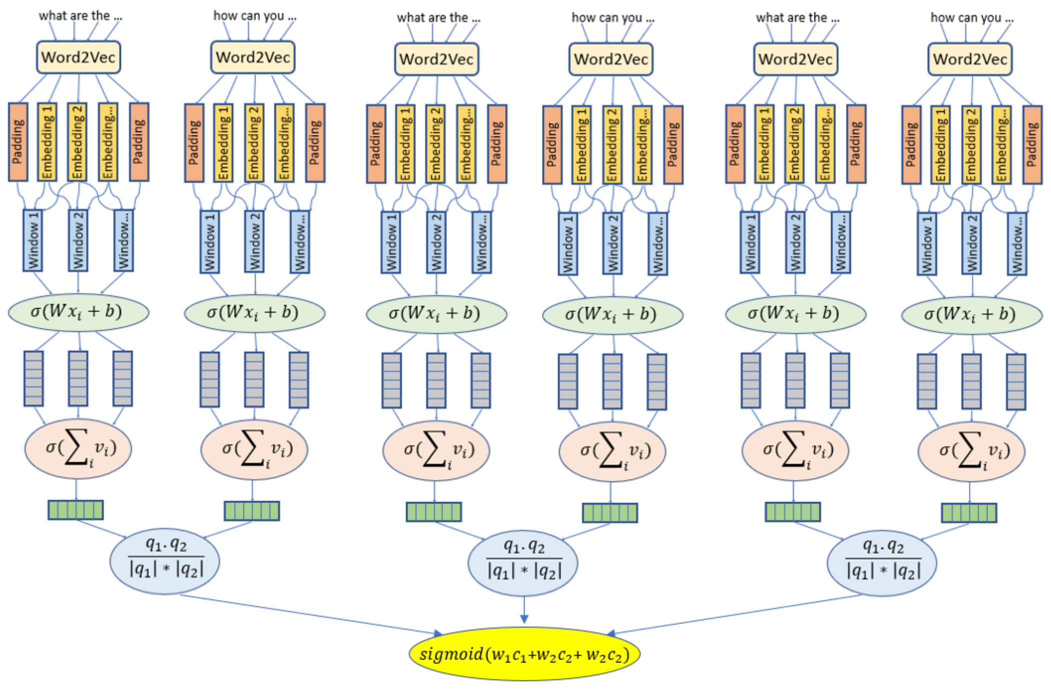

Figure 4: Sigmoid Ensemble of Three CNNs

5 Experiments and Results

5.1 Baseline Classifier: Random Forest

The first model we tested was our baseline Random Forest classifier. As mentioned earlier, this model was trained on 80% of the dataset and tested on the remaining 20%. We tested two Random Forests, one using total similar word count, and another using percentage similar word count. Our total similar word count model achieved an accuracy of 63.239% on the test set while our percentage similar word count model achieved 63.054%. Again, note that these two models were both extremely oversimplified with their purpose being to give us a low-level baseline. In fact, these two models essentially scored at the level you would expect simply by choosing the majority class. We believe this shows that a model can in fact be too simple, and that counting the number of similar words in two sentences is a poor estimate in regard to their semantic similarity.

5.2 Main Model: Convolutional Neural Network (CNN)

The second model we tested was our primary CNN. To test this model we split our dataset 90-10 percent in regard to training and test sets. Our model was then trained on over 360,000 data points, and evaluated on the test set after every epoch. After 14 epochs, the model achieved a test accuracy of 77.443%. As expected, this model outperformed the accuracy of the Random Forest classifier.

5.3 Ensemble Model: Various Techniques

For the final part of this project we evaluated the performance of our ensemble CNNs. We first trained each individual CNN on their specific subspace of the dataset and then evaluated them on a validation set. As described in section 4.4, each model was trained on approximately 123,000 data points which were split 90-10 percent in terms of training and validation sets. CNN1, CNN2, and CNN3 achieved validation accuracies of 75.295%, 75.789%, and 76.045% respectively.

After training these models on their respective training sets, we then evaluated each on our left out test set. CNN1 achieved 75.890% accuracy, CNN2 achieved 75.688% accuracy, and CNN3 achieved 75.665% accuracy. Again, this far exceeded the baseline accuracy for our Random Forest classifier, showing the potential for CNNs, typically used in the field of image recognition and computer vision, to be used in addressing NLP problems. And, as mentioned previously, these models perform nearly as well as our main CNN trained on the entire test set, adding credence to our belief that since our network is so small it doesn't need to see as many sample points to perform well.

For the final step, we explored a variety of different techniques to combine our three trained CNNs in various ways to form an ensemble. The first technique we explored was a majority vote function, this technique scored an accuracy of 76.002% on the test set. The second technique we implemented was an average vote, this technique scored a 76.066% accuracy on the test set. Lastly, we trained a final layer sigmoid unit. This model was trained on the same training set used to train our three individual CNNs and achieved 74.910% accuracy on the test.

5.4 Summary

Model	Testing Accuracy (%)	Training Samples
Random Forest (total similar word count)	63.239	323,480
Random Forest (percentage similar word count)	63.054	323,480
Main CNN	77.443	363,915
CNN1	75.890	109,174
CNN2	75.688	109,174
CNN3	75.665	109,174
Majority Vote	76.002	N/A
Average	76.066	N/A
Sigmoid Unit	74.910	45,000

Table 1: Testing accuracies for each model

Figure 5 shows the validation accuracies per epoch for our main CNN, each individual CNN (1, 2, and 3), and our sigmoid unit ensemble. We also show in Table 1 a summary of the test accuracies of the baseline Random Forest, our main CNN model, CNN1, CNN2, CNN3, majority vote ensemble, average value ensemble, and sigmoid ensemble. We suspect that the reason the ensemble methods did not perform much better than the individual CNNs is because the CNNs learned the same local minimum.

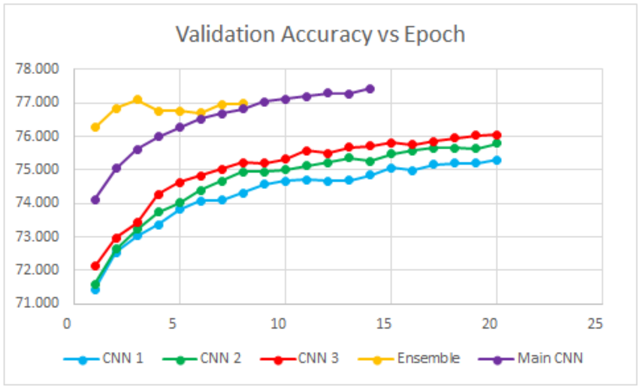

Figure 5: Validation accuracy for each model per epoch

6 Conclusions and Future Work

Convolutional Neural Networks are generally used exclusively for image analysis applications. Here, we were able to verify and show that CNNs can also be extremely useful in natural language processing. This suggests many new use cases for CNNs and also has implications in training time for NLP models. Many standard approaches to detecting duplicate questions involve Siamese networks and other large models that contain upwards of 500,000 parameters. Our CNN contains only about 180,000 parameters, making it faster to train.

Our results showed that the CNN approach can achieve good accuracies, although maybe not state-of-the-art with our current design. Future work may try to modify the structure of the CNN in regard to window size, word embedding size, multiple convolutional layers, etc. to see if this results in better performance. We were also able to show that the CNN approach performs significantly better than a Random Forest classifier. With respect to ensemble classifiers, we showed that at least for this particular problem, majority vote and weighted average of cosine similarities had almost the same performance, and performed better than any of the three CNNs (individually). The weight-optimized ensemble classifier with the extra sigmoid unit did not perform as well.

Future work should focus on applying Convolutional Neural Networks in other natural language processing questions. It would be interesting to see if CNNs can perform as well in machine translation, sentiment analysis, or named entity recognition. In addition, our ensemble classifier did not perform as well as we had hoped, most likely because the trained CNNs likely learned very similar weight matrices. This means any linear combination of the CNNs is unlikely to improve on any single CNN; it is more likely to simply add uncertainty. If the CNNs had learned different weight matrices, we believe the ensemble classifier would have been more successful. This suggests applications in time-varying semantic analysis. That is, future work could investigate how semantics of words and sentences vary over time. If we train three different CNNs, one on questions from the 18th century (although infeasible), one on questions from the 19th century, and one on questions from the 20th century, for example, they are likely to learn different weight matrices since the semantics of words is different in each period. Then, combining the CNNs using an ensemble classifier would likely lead to better results.

References

[1] Predictive modelling, a simple approach ! | kaggle. `https://www.kaggle.com/arathee2/predictive-modelling-a-simple-approach/notebook`. (Accessed on 12/12/2018).

[2] Question pairs dataset | kaggle. `https://www.kaggle.com/quora/question-pairs-dataset`. (Accessed on 12/12/2018).

[3] Travis Addair. Duplicate question pair detection with deep learning.

[4] Dasha Bogdanova, Cıcero Santos, Luciano Barbosa, and Bianca Zadrozny. Detecting Semantically Equivalent Questions in Online User Forums. *Proceedings of the 19th Conference on Computational Language Learning*, pages 123–131, 2015. URL `http://www.aclweb.org/anthology/K15-1013`.

[5] Evgeniy Gabrilovich and Shaul Markovitch. Computing Semantic Relatedness using Wikipedia-based Explicit Semantic Analysis. *Proceedings of The Twentieth International Joint Conference for Artificial Intelligence*, pages 1606–1611, 2007. URL `http://www.aaai.org/Papers/IJCAI/2007/IJCAI07-259.pdf`.

[6] Mark Harmon and Diego Klabjan. Activation ensembles for deep neural networks. *CoRR*, abs/1702.07790, 2017.

[7] Hua He, Kevin Gimpel, and Jimmy Lin. Multi-Perspective Sentence Similarity Modeling with Convolutional Neural Networks. *Proceedings of EMNLP 2015*, (September):1576–1586, 2015. doi: 10.1016/j.jvolgeores.2014.02.007. URL `http://www.anthology.aclweb.org/D/D15/D15-1181.pdf`.

[8] Y. Homma. Detecting duplicate questions with deep learning. 2017.

[9] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[10] Shankar Iyer, Nikhil Dandekar, and Kornél Csernai. First Quora Dataset Release: Question Pairs - Data @ Quora - Quora, 2017. URL `https://data.quora.com/First-Quora-Dataset-Release-Question-Pairs`.

[11] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`. [Online; accessed <today>].

[12] Carlos Orrite, Mario Rodríguez, Francisco Martínez, and Michael Fairhurst. Classifier ensemble generation for the majority vote rule. In José Ruiz-Shulcloper and Walter G. Kropatsch, editors, *Progress in Pattern Recognition, Image Analysis and Applications*, pages 340–347, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-85920-8.

[13] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[15] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[16] Zhiguo Wang, Wael Hamza, and Radu Florian. Bilateral multi-perspective matching for natural language sentences. *IJCAI International Joint Conference on Artificial Intelligence*, pages 4144–4150, 2017. ISSN 10450823. doi: 10.24963/ijcai.2017/579.