Neural Networks For Sentence Classification

Deepti Suresh Aparna Ganesh Ryan Maxey

1. Introduction

Text classification can be used for a variety of tasks such as sentiment analysis, topic detection, intent identification, and much more. Automatic text classification applies machine learning, natural language processing, and other AI-guided techniques to automatically classify text in a faster, more cost-effective, and accurate manner [1]. Sentence classification is a type of text classification and often has many real life applications such as detecting hate speech, being used in AI-driven chatbots for assisting with mental health, being used in writing assistants such as Grammarly, along with others.

In our paper, we propose two recurrent neural network (RNN) models and a convolutional neural network (CNN) model to perform sentiment analysis to classify if a sentence is positive or negative. Our work will be using a convolutional neural network (CNN) to conduct sentiment analysis, reproducing the model from Convolutional Sentiment Analysis [2]. Additionally, we will be reproducing and testing a simple RNN [3] and an Advanced RNN model [4]. The dataset that we will be using is the IMDb dataset [5] which is an extensive set of movie reviews.

2. Background

Recurrent neural networks are a type of deep learning algorithm that use sequential or time series data. They have proven to be effective for temporal and ordinal tasks such as language translation, speech recognition, and natural language processing. Due to their utility in these scenarios, RNNs have become popular choices for voice assistants, such as Siri, and translating tools, such as Google Translate [6].

CNNs are a popular type of deep learning neural network that has primarily been used for image processing and computer vision tasks. Although RNNs are more associated with Natural Language Processing (NLP) tasks, including sentence classification, CNNs can also show promising results. Traditionally, with image processing, CNNs are used because they have the ability to extract features, however, this can also be applied to text in order to extract important information that will accurately classify the sentence's sentiment [7].

3. Approaches

For the recurrent neural network (RNN), a sequence of words will be used as input. The RNN will produce a hidden state for each word, processing them sequentially. To calculate the hidden state for a word, the RNN will take the current word, x_t, the hidden state produced for the previous word, h_{t-1}, and use the equation below.

$$h_t = RNN(x_t, h_{t-1})$$

Once the final hidden state is produced, it is passed to a linear layer which gives the predicted sentiment of the input sentence [3].

The Advanced RNN model features the most drastic changes from the simple RNN model. Firstly, we used packed padded sequences. This will make our RNN only process the non-padded elements of our sequence, and for any padded element the output will be a zero tensor. Next is the use of pre-trained word embeddings. Now, instead of having our word embeddings initialized randomly, they are initialized with these pre-trained vectors, "glove.6B.100d" [9]. We also utilized a different RNN architecture, LSTM (Long Short-Term Memory) which will help us overcome the vanishing gradient problem that standard RNNs have [10]. Next, we used a bidirectional RNN. As well as having an RNN processing the words in the sentence from the first to the last (a forward RNN), we have a second RNN processing the words in the sentence from the last to the first (a backward RNN). Additionally, we used a multi-layer RNN, where essentially the idea is that we add additional RNNs on top of the initial standard RNN, where each RNN added is another layer. And finally we utilized dropout and a different optimizer.

With Convolutional Neural Networks (CNN), each sentence is used as a sequence of word embeddings as input to the model. The input layer takes each word in a sentence and converts it into a higher-dimensional vector. This is fed into the convolutional layers which will apply a set of kernels to create a feature map which will indicate local patterns and relationships between words. Lastly, the fully connected layer will flatten the produced feature maps into fully connected layers which utilize weights to predict the final output. This layer is important for the layer to learn more complex relationships in order to classify the sentiment of the sentence. The output

will be a probability distribution of the likelihood of a sentence belonging to the different sentiment classes (positive or negative) [8].

Our goal is to compare these three approaches and investigate which neural network performs better through analysis of various performance metrics such as accuracy, precision, recall, and F1 score.

4. Results

4.1 Main Results

Model	Accuracy	Precision	Recall	F1-Score
Simple RNN (Base Model)	44.50%	0.4762	0.6854	0.5223
Advanced RNN	84.39%	0.7537	0.926	0.831
CNN	85.52%	0.8021	0.8585	0.8211

Table 1: Results for Simple RNN, Advanced RNN, and CNN Models

Here are our results. For our Simple RNN model, we were able to achieve an accuracy of 44.5%. For our Advanced RNN and CNN models, it was 84.39% and 85.52% respectively, which are significantly higher than the Simple RNN model. We can also see that we obtained much better values for Precision, Recall, and F1-Score for the Advanced RNN and CNN models, indicating that our enhancements were effective in improving the performance.

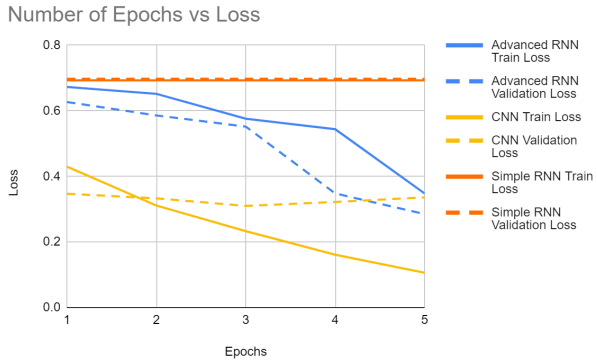

Figure 1: Training Loss and Validation Loss for Simple RNN, Advanced RNN, and CNN

So here, we plotted the training and validation loss per epoch for each model, where the dotted line in Figure 1 represents the validation loss and the solid line represents the training loss. There was very little change in training and validation loss across each epoch for the Simple RNN. This may indicate that the model might have converged to a suboptimal solution or have plateaued, resulting in a high bias or low variance. Conversely, the Advanced RNN model's training loss and validation loss decreased overtime indicating that the model is learning and improving its ability to make predictions on the data. This is a positive sign, as it suggests that the model is effectively capturing the patterns and relationships in the training data and is generalizing well to the validation data. However, for the CNN model, we can see that although the training and validation loss decreases over time, the validation loss does not decrease as much indicating that there might be a case of overfitting.

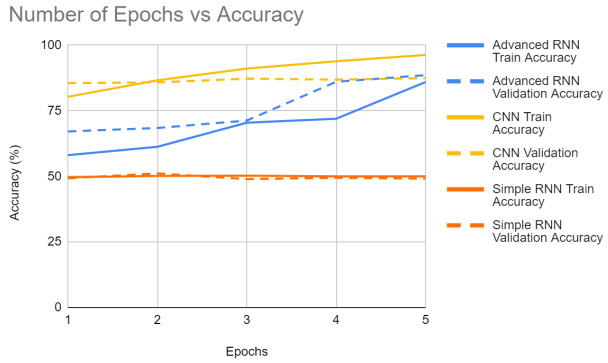

Figure 2: Training Accuracy and Validation Accuracy for Simple RNN, Advanced RNN, and CNN

We also plotted the training and validation accuracy per epoch for each model to understand the models' behavior and if it correlates with the training and validation losses we saw in the previous graph. We can see that there is very little change in the Simple RNN's training and validation accuracy. However, for the Advanced RNN model, there is a gradual increase in training and validation accuracy indicating that model is learning well and is able to fit the data properly. For the CNN model, we can see that it achieves the highest validation accuracy and train accuracy in comparison to the other models, however its validation accuracy does not increase as much over time. As mentioned before, this might indicate that our CNN model is slightly overfitting. However, since the CNN model also has the highest test accuracy out of all the models, this is not a definitive conclusion.

4.2 Exploratory Tasks

4.2.1 Train/Validation Dataset Split Ratio

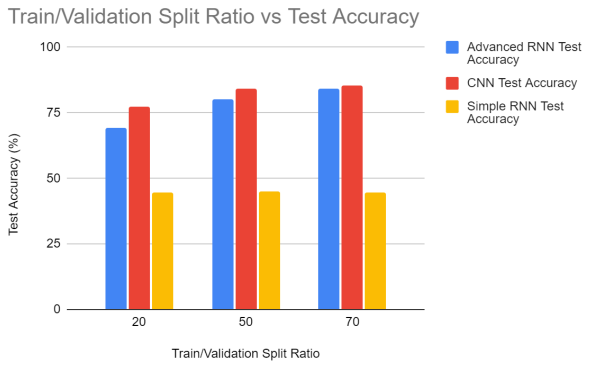

Figure 3: Test Accuracy for Various Train/Validation Splits

The first exploratory task involved training the model with three different ratios for splitting the training and validation sets. We used splits of 20, 50, and 70 percent. This number represents the percentage of the dataset used for training with the remaining data used for the validation set (this is after initially reserving a subset of the data for testing). We can see that the Simple RNN model with its limitations was mostly unaffected by the different splits. However, we can see an increase for both the Advanced RNN and CNN models as the split size increases. Intuitively, this makes sense because these models have shown the ability to iteratively improve so having less training data would prevent them from achieving higher test accuracies.

4.2.2 Hyperparameter Tuning: Batch Sizes

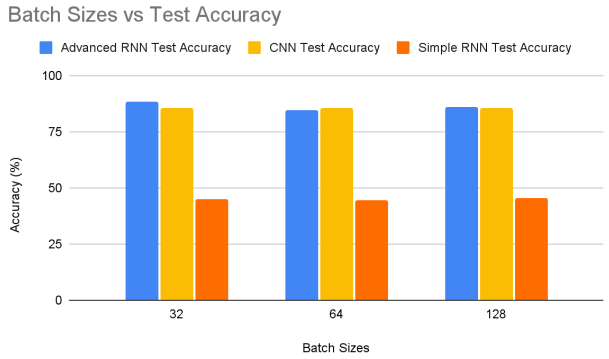

Figure 4: Test Accuracy for Various Batch Sizes

Another hyperparameter that was tested was the batch size for the number of sentences input into the model at a time. We tested the three models with batch sizes of 32, 64, and 128. Again, there is relatively little improvement in the test accuracy of the Simple RNN for the different batch sizes. There is also little change in the test accuracy for the Advance RNN and CNN models. However, we can observe that a batch size of 32 offered the best test accuracy for the Advanced RNN.

4.2.3 Hyperparameter Tuning: Optimizer (SGD vs Adam)

One of the hyperparameters we explored was altering the optimizers and observing how the performance of our models vary. Optimizers are used to update weights and biases of a model in such a way that it reduces the risk of errors or loss from predictions and improves the overall accuracy of predictions and classification made by the model.

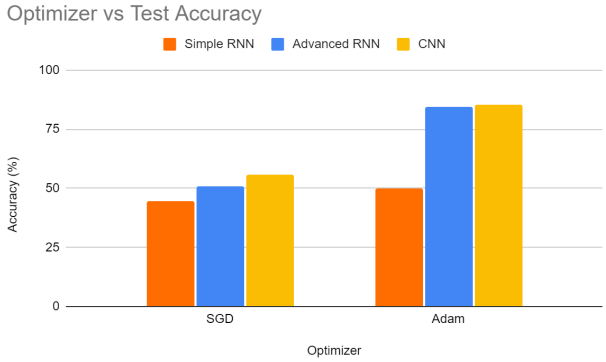

Figure X: Test Accuracy of models against SGD and Adam optimizer

Traditionally, Stochastic Gradient Descent (SGD) is a popular optimizer because it is more efficient than Gradient Descent as it is faster and more suitable for larger datasets. However, its disadvantage is that the gradient is not computed for the dataset as a whole, but only at a random point for each iteration. This causes higher variance, making it harder for the algorithm to converge. On the contrary, Adam tends to converge much faster. Moreover, instead of maintaining a single learning rate throughout training like in SGD, the Adam optimizer updates the learning rate for each network weight individually. Due to this reason, we see that the Simple RNN, Advanced RNN and CNN all have higher accuracies using Adam instead of the SGD optimizing algorithm.

4.2.4 Hyperparameter Tuning: Dropout Rate

Another hyperparameter that was investigated was the dropout rate for the Advanced RNN and CNN models. To define, dropout is a regularization method which is used to make the model more robust. This is done by dropping nodes at random which makes the training process noisy. Due to this, nodes within a layer have to take on more or less responsibility for the inputs and therefore the network aims to fix errors committed by previous layers.

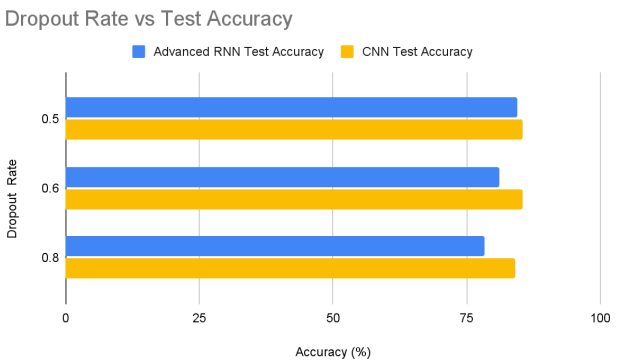

Figure Y: Test Accuracy of models with varying dropout rates

Some popular dropout rates for neural networks range from 0.5 to 0.8, hence, we explored values 0.6 and 0.8 to see how they compare to the default value of 0.5. Although insignificant to the eye in the figure above, the accuracy does drop as the dropout rates increase. This is because when the dropout rate is increased beyond a certain threshold, it causes the model to not be able to fit properly. A higher rate would result in higher variance and can degrade training. Additionally, since this is a form of regularization, if it is done too much, it can lower the capacity of the model to a point where poor predictions are yielded from the model.

6. Code Explanation and Running Instructions

For our project we decided to use Google Colab, a cloud-based development environment to run our code and perform all the necessary computations. For each model, we created a separate Colab Notebook to keep the code clean and segregated. We describe the code design and implementations in detail below for each model. The Google Colab notebooks are linked below.

- Simple RNN
- Advanced RNN
- CNN

6.1 Simple RNN Model

The simple RNN model starts by preparing the data and choosing a tokenizer for dividing the input sentences into tokens. We use TorchText to import the IMDb dataset and it automatically splits into two equal-sized subsets to be used for the training and testing sets. We follow this by further splitting the initial training set into training and validation sets. This is where we experiment with different training/validation splits. We also build a vocabulary by taking the top 25,000 words and creating one-hot vectors for them. Lastly we create a BucketIterator to iterate over the example sentences.

After preparing the data we build the model with an embedding layer, recurrent neural network, and linear layer. The embedding layer is used to turn a one-hot vector into a dense vector representation for input into the RNN. The linear layer takes the final hidden state produced by the RNN as input and uses a fully connected layer to produce the predicted sentiment.

After creating the model, we move on to training. By default we use SGD for the optimizer, and Binary Cross Entropy with Logits for the loss function. Here we also experiment with using Adam as the optimizer. Next, we train the model by iterating over all the training data, and this is done for five epochs. During the validation stage, if the model's performance is the best we have seen, then we save the parameters and later use them for testing where we obtain our final testing loss and accuracy.

6.2 Advanced RNN Model

We first begin preparing the data for the Advanced RNN Model. We use TorchText to import the IMDb dataset and it automatically splits into two subsets to be used for the training and testing sets. Since we're using packed padded sequences, we will make the RNN model process the non-padded elements of our sequence, and for any padded element, the output will be a zero tensor. We also utilize pre-trained word embeddings to avoid having our word embeddings initialized randomly. We used "glove.6B.100d" vectors [9], where glove is the algorithm used to calculate the vectors. 6B indicates that these vectors were pretrained on 6 billion tokens and 100d indicates that these vectors were 100-dimensional. We ensure that all the packed padded sequence tensors are sorted by their lengths by setting `sort_within_batch = True` in the iterator.

For implementing the model, since padding tokens are irrelevant to determining the sentiment of a sentence, the embedding for the pad token will remain at what it is initialized to (we initialize it to all zeros later). We use LSTM instead of the standard RNN and since the final hidden state of our LSTM [11] has both a forward and a backward component, which will be concatenated together, the size of the input to the nn.Linear layer is twice that of the hidden dimension size. Implementing bidirectionality and adding additional layers are done by passing values for the num_layers and bidirectional arguments for the RNN/LSTM. Dropout is implemented by initializing an nn.Dropout layer and using it within the

forward method after each layer we want to apply dropout to. The LSTM has a dropout argument which adds dropout on the connections between hidden states in one layer to hidden states in the next layer.

Before we pass our embeddings to the RNN, we pack them allowing our RNN to only process the non-padded elements of our sequence. The RNN will then return a packed sequence as well as the hidden and cell states which are both tensors. We ensure that the lengths argument of packed_padded_sequence is a CPU tensor. We then unpack the output sequence to transform it from a packed sequence to a tensor. As we want the final layer forward and backward hidden states, we get the top two hidden layers from the first dimension and concatenate them together before passing them to the linear layer, after applying dropout.

Next we create an instance of our RNN class with our new parameters and arguments for the number of layers, bidirectionality, and dropout probability, copy the pre-trained word embeddings into the embedding layer of our model, and then replace the initial weights of the embedding layer with the pre-trained embeddings. We then initialize both <unk> and <pad> token to all zeros to explicity tell model that they are irrelevant.

Now it's time to train the model. We define the optimizer (Adam) and criterion (BCEWithLogitsLoss) and place the model on a CPU if available. We train the model for five epochs and call methods such as train and evaluate and calculate the accuracy, precision, recall, and f1-score per epoch. We finally test the trained model on the test set to obtain our new and improved accuracy. Now our model is capable of classifying the sentiment of any sentence related to movie reviews.

6.3 CNN Model

The CNN model also begins with the preliminary step of preparing the data for this type of model. The hurdle with implementing a CNN is visualizing how it can be used with text. It involves converting words into word embeddings. A filter will then be used that is [*n* x *embedding_dimesion*], this means it will cover *n* sequential words. The model will use different filter sizes such as 3, 4 and 5 to investigate the occurrence of certain patterns across all these filters to find relevant information for sentiment analysis. Each element will have a weight associated with it and the output filter will be a real number that indicates the weighted sum of all elements covered by the filter. As the filter moves down, the output of weighted sums are calculated. Finally, max pooling is applied on these outputs in which the maximum value over the dimension is considered. This maximum value is an indicator of the most important feature that determines the sentiment of the sentence.

To delve into implementation details, in order to make our model more generic to the number of filters it can take, all convolutional layers are placed in a nn.ModuleList. Then, for varying filter sizes, the appropriate layers will be created and passed into the *forward* method in which they are iterated through to compute the output and fed through the max pooling before being passed into dropout layers.

Once the model has been built, we train and test the model similar to the Advanced RNN.

7. Lessons Learned

There were valuable lessons that we can take away from this project. In order to achieve optimal performance with Neural Networks for the goal of Sentence Classification, first and foremost it is important to ensure that we have adequate computing resources to support our models. Additionally, we should set the runtime type to "GPU" or "TPU" when using Google Colab to run the code to ensure efficient computation. We also learned that utilizing the CPU can result in extremely slow processing times, thus, necessary precaution must be taken to use a more powerful computing source. Overall, our experimentation with different models revealed some noteworthy findings. We learned that with slight modifications, Recurrent Neural Networks (RNNs) can be a powerful tool for the task of Sentiment Analysis. Additionally, Convolutional Neural Networks (CNNs) proved to be well-suited for the task of sentiment analysis as well. These insights are certainly valuable for future research pertaining to Neural Networks for Sentence Classification, as they provide guidance on which models to consider and how we can optimize them according to the tasks.

8. Future Work

Future work for the project could include supporting more complex classification of movie reviews. Rather than the binary classification of "positive" and "negative", there could be additional labels for reviews with sentiment in between these two. This could be done by choosing a different dataset such as Amazon Product Data that includes reviews with ratings from one to five along with feedback of whether or not a review is helpful. The Stanford Sentiment Treebank is another dataset that could be explored. Lastly, there could also be further

testing of hyperparameters such as the embedding dimension and hidden dimension.

References

[1] MonkeyLearn Inc. “Text Classification: What It Is and Why It Matters.” MonkeyLearn, monkeylearn.com/text-classification/.

[2] Trevett, B (2021) Convolutional Sentiment Analysis https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/4%20-%20Convolutional%20Sentiment%20Analysis.ipynb

[3] Trevett, B (2021) Simple Sentiment Analysis https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/1%20-%20Simple%20Sentiment%20Analysis.ipynb

[4] Trevett, B (2021) Updated Sentiment Analysis https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/2%20-%20Upgraded%20Sentiment%20Analysis.ipynb

[5] Maas, Andrew L, et al. “Learning Word Vectors for Sentiment Analysis.” The 49th Annual Meeting of the Association for Computational Linguistics Human Language Technologies: Held at the Portland Marriott Downtown Waterfront in Portland, Oregon, USA, June 19-24, 2011, ACL?, 2011.

[6] “What are recurrent neural networks?,” IBM. [Online]. Available: https://www.ibm.com/topics/recurrent-neural-networks. [Accessed: 26-Mar-2023].

[7] Shreya Ghelani, “Text Classification—RNN’s or CNN’s?,” Medium, Jun. 02, 2019. https://towardsdatascience.com/text-classification-rnns-or-cnn-s-98c86a0dd361

[8] “Introduction to Convolutional Neural Networks CNNs,” aigents.co. https://aigents.co/data-science-blog/publication/introduction-to-convolutional-neural-networks-cnns

[9] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation.

[10]Wikimedia Foundation. (2023, April 26). Vanishing gradient problem. Wikipedia. Retrieved May 4, 2023, from https://en.wikipedia.org/wiki/Vanishing_gradient_problem

[11] Olah, C. (n.d.). Understanding LSTM networks. Understanding LSTM Networks -- colah's blog. Retrieved May 4, 2023, from https://colah.github.io/posts/2015-08-Understanding-LSTMs/