An Investigation into the Use of Pyramidal Convolution to Determine the Sentiments in IMDb Movie Reviews

Nikhil Friedman

Purdue Unviersity
friedm29@purdue.edu

Abstract

The field of machine learning has exploded in the past century due to the creation of the Neural Network, along with the continuously progressing ability for computers to facilitate their implementation. While a traditional Feed-Forward Neural Network (FFNN) is a powerful tool for nearly any classification or regression task it is given, there are improvements that can be made. One such improvement deals with how the features are handled before being passed through a more traditional Multi-Layer Perceptron (MLP) model. This improvement is referred to as a “Convolutional Neural Network” (CNN), due to the introduction of convolutions that are performed on the features before they are introduced to a more traditional densely linked network framework.

CNNs are notable for their abilities with image processing. More specifically, they are notable for their increased ability to recognize images of varying sizes, and their ability to retain special relations between objects in a given image. This is all facilitated through the “kernels” of a CNN, which represent the layers of convolutions being performed on the feature set before they are passed through the network. CNNs are more than capable for most image classification tasks, however, there are still potential improvements. This is where Pyramidal Convolution (PyConv) comes into play.

PyConv improves not only the efficiency of the training of the network, it also enables the input features to be more flexible and improves upon the spatial awareness of CNNs in some workloads. In this submission, I will document what exactly PyConv improves upon, and how I am implementing it in my own personal study.

Introduction

The concept of using Neural Networks to process large datasets with many features is not a new concept in the slightest. The first notion of the “Multilayer Perceptron” (MLP) was put forth by Frank Rosenblatt in 1958 . Rosenblatt’s model was not powerful, consisting only of an input layer, a hidden layer that could not adjust its own weights/biases, and an output layer. A decade later, in 1967, Shun’ichi Amari would create the first MLP with multiple hidden layers, each with weights being updated by Stochastic Gradient Descent (SDG) . This marked the first example of the modern “Feed Forward Neural Network” (FFNN), which is a model still used to this day for basic image recognition tasks. For example, the USPS famously created the MNIST dataset, which consisted of labeled images of handwritten digits from zero through nine. A FFNN was used to train a model off this dataset, which was capable of recognizing handwritten digits in zip codes on mail.

While FFNNs are more than a powerful tool, there were improvements to be made. The next big leap in the field of image recognition was the introduction of “Convolutional Neural Networks” (CNNs). These were introduced by Yann LeCunn and his research team in 1989 . A CNN consists of a traditional densely linked neural network being fed its input layer through layers of convolutions on the given input data. Much of the computation that happens after the layers of convolutions is identical to that which exists within a FFNN. While this may seem unnecessary, as adding more hidden layers could potentially allow a traditional FFNN to recognize the same patterns that the convolution layers could, the CNN architecture allows the network to retain spatial information to a better degree. This means that without adding much computational complexity in the training, the model can better identify images of different sizes, as well as the spatial relations between different objects in a given image. CNNs are still a powerful tool that is used to this day for image recognition tasks, however, there are of course improvements that can be made for specific workloads. On top of the potential for accuracy improvements, the nature of the convolutional layers in a CNN inherently causes an increase in its runtime complexity. This is because the convolutional layers in a CNN are fed one after another, meaning that the input of one layer must wait for the output of the previous layer for it to be computed. While modern processors are fast in single-cycle tasks, it is always better to be able to parallelize tasks, which is something that is not possible in this scenario. This is where the idea of “Pyramidal Convolution” (PyConv) comes into play.

Pyramidal Convolution is a far newer concept, and has been shown to improve flexibility, performance, and accuracy under certain workloads. The key difference between PyConv and traditional CNN is the way in which the kernels (or filters) are arranged within the network. Kernels merely represent a matrix for the convolutions to be performed on while the network is doing its forward-pass. In a traditional CNN, they are fed one after another, which causes the afore-

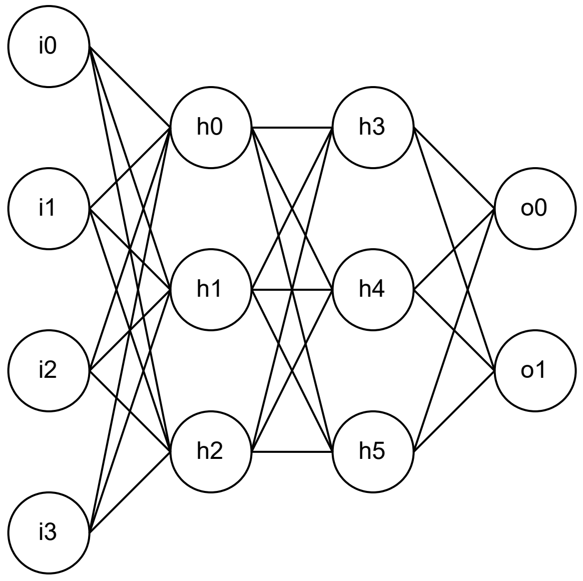


Figure 1: The above figure shows the structure for a traditional Feed-Forward Neural Network (FFNN). As can be seen above, the network consists of a layer of input neurons, two hidden layers, and a layer of output neurons. The layer of input neurons, or ”input layer” takes data directly from an entry in the dataset. This means that the number of input neurons directly corresponds to the number of features in the dataset. From there, the weights and biases are applied to the values of the input neurons, the values are passed through the hidden layers, and arrive at the output layer. The output layer is significant due to the presence of an activation function, which makes the classification result of the network more statistically significant. Essentially, the activation function (usually SoftMax) represents the values of the output neurons as a part of the entire magnitude of all of the output neurons combined. This result can be interpreted as the network’s ”prediction” for the classification of a given entry.

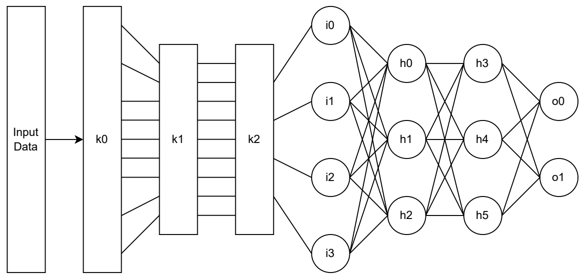


Figure 2: The above figure shows the structure for a traditional Convolutional Neural Network (CNN). As can be seen above, the network architecture is quite similar to that of a Feed-Forward Neural Network (FFNN), however, the treatment of the features before they reach the network is different in this case. A traditional FFNN, the input layer is driven directly by the features on a given entry in a dataset. In other words, the data that is being analyzed or used for training is directly fed through the network, with minimal pre-processing. A CNN takes a different approach by containing multiple kernel layers, each of which representing a differing amount of convolutions being run on the input data. These kernel layers are driven from one another, and eventually feed into the densely-connected network. Once the data reaches the network, the same process occurs as is seen in the FFNN implementation.

mentioned computational complexity issues. Additionally, this practice of passing the kernel layers into each other can decrease accuracy in some circumstances. PyConv instead has a set layer of kernels, each descending in size (to form a pyramid). Each of these kernels are fed independently from the input data, allowing them to be computed in parallel. The outputs of each of these kernels are then ran through a traditional neural network architecture, where a prediction can be made, or the model can be trained further. In summary, PyConv proposes a potential next step in the field of image classification, with significant improvements upon the traditional CNN architecture.

Performance Improvements

As previously mentioned, PyConv replaces the kernels of traditional CNNs with multiple kernels of varying sizes, all being driven in parallel from the input data. This not only decreases the runtime of training and running the model, since these computations can be performed in parallel now, but also provides extra hierarchical context for the network to process. By including multiple kernel sizes simultaneously, the network is being provided with extra information regarding the sizing of objects, different angular relationships, and plenty else that is not able to be specified.

Another performance improvement inherent to this style of network comes from the fact that the network following the convolutional kernels can often be downsized in a PyConv implementation. This is because traditional CNNs generally need extensively deep networks in order to pick up on the same hierarchical context that is provided by the Pyramidal kernel layers of PyConv. While this may not be true in all workloads, it has been proven to be generally true of PyConv implementations vs. traditional CNNs.

Lastly, the capacity to over-fit is far lesser in PyConv implementations rather than traditional CNNs. This is a result of the increased variety in the input layers for the networks, since more separate convolutions are being performed, and of the reduced complexity of the networks due to this variety. The feature treatment in PyConv implementations overall enables for more efficient computation during training, due to its high capacity for parallelization an its enabling of decreased density in networks. While traditional CNNs may outperform PyConv under certain workloads, the performance improvements of PyConv cannot be ignored.

Flexibility Improvements

The architecture of PyConv inherently adds many new ways for network architectures to be flexible. The number of separate parallel pyramidal kernels, as well as their sizes, can be

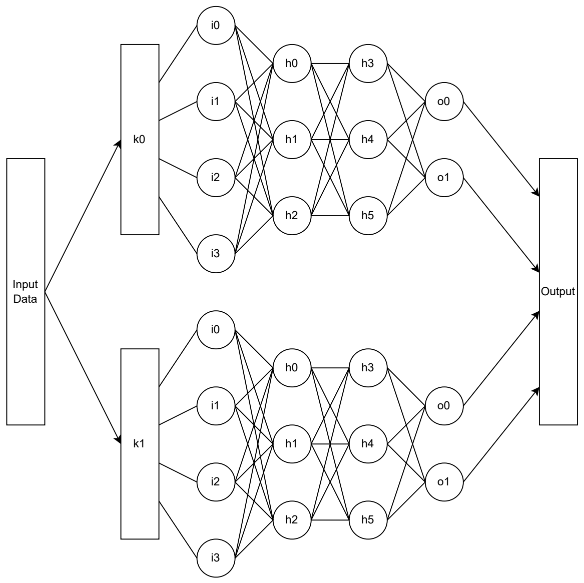


Figure 3: The above figure shows the structure of a network implementing Pyramidal Convolution (PyConv). As can be seen above, PyConv consists of the same pieces as a Convolutional Neural Network (CNN) - kernel layers and a densely-connected neural network. However, where PyConv differs is in the configuration of these elements. In a traditional CNN, the kernel layers are linked together, with each kernel layer receiving the previous layer's output as its input. In PyConv, however, each kernel layer receives input directly from the input data, and its output is directly passed into its own densely-linked network. As discussed below, this not only improves performance and flexibility, but also decreases training time due to the increased ability for parallelization.

altered to fit each specific image classification need. Since all of these extra kernels are added in parallel, there is theoretically no added computational cost to these modifications, making them extremely viable for actual workloads and implementations.

Lastly, the network architecture allows for different kernel sizes and implementations down the path of the network. This means that for larger networks, there can be additional kernels down the path of the network, allowing for even more accurate computation. To summarize, the PyConv architecture introduces a whole new variety of hyper-parameters, thus increasing the flexibility of the network implementation. While traditional CNNs are great, the addition of new hyper-parameters ensures that PyConv can be better tailored to specific workloads, which can improve accuracy and decrease training time.

Accuracy Improvements

As discussed previously, the main advantage of the CNN over traditional FFNNs is the additional spatial context given by the kernel layers. While CNNs are powerful for this type of image classification on their own, PyConv improves upon this advantage by adding even more separate kernels, of varying sizes. This not only allows for more efficient training, but it also provides additional context for the network to interpret. This has been shown to aid in the understanding of images of different sizes, along with different angular relationships and spatial relationships.

On top of this, the decreased training burden caused by the PyConv architecture allows for over-fitting to be less of a problem. The more simplistic network that the PyConv kernels allows for not only decreases computational burden, but also makes the network more accurate.

Related Work

In a paper titled ”Pyramidal Convolution: Rethinking Convolutional Neural Networks for Visual Recognition,” researchers Duta, Liu, Zhu, and Shao examined the use of the aformentioned PyConv networks for image recognition. They reasoned that the additional context provided by the PyConv architecture could be used in the field of image processing. This context, in their paper, was able to provide useful insights to the model regarding depth and angles in images, allowing for more precise image recognition to be conducted.

While the researchers Duta, Liu, Zhu, and Shao investigated the use of PyConv in the context of image recognition (more specifically, results with the ImageNet dataset), this document outlines an implementation of PyConv for text processing. This document will use a similar structure to the one used by the aforementioned researchers, though it will be modified to analyze IMDb movie reviews. The assumption here is that the same benefits realized in their study will present themselves in this study. It is hoped that the same context that provided spatial information to their model will provide contextual information for different tokens and words within sentences.

Problem Definition

IMDb movie reviews present a unique challenge for machine learning models due to their complex linguistic devices, such as metaphors, sarcasm, and overall intricate use of the English language. PyConv networks benefit from multiple layers and sizes of convolutions. In image recognition, this allows the network to gain a better understanding of the image from different scales and angles. In the context of text recognition, this same advantage allows the network to understand the different patterns and contextual information regarding the text at different sizes. This enhances the ability of the model to understand broader sentence structure, leading to better results in text analysis. This makes the choice of PyConv appealing for the problem of categorizing IMDb reviews, since the varying kernel sizes of the different convolution layers should give it an inherent advantage over traditional models when it comes to these outlined nuances.

In this implementation, the PyConv network architecture is applied to make predictions regarding the sentiment of movie reviews sourced from IMDb. IMDb holds a vast database of reviews of movies, both positive and negative, which made it an excellent source for training/testing data.

The reviews, as mentioned above, are incredibly nuanced in their use of the English language, and will therefore serve as an excellent benchmark for the effectiveness of the PyConv architecture in the field of text analysis.

Methodology

The PyConv architecture detailed in the paper ”Pyramidal Convolution: Rethinking Convolutional Neural Networks for Visual Recognition.” was used to create the PyConv architecture for the model used in this implementation. The data used for this project (IMDb Sentiment) was first pre-processed with using the Global Vectors for Word Representation (GloVe) project. These GloVe embeddings were saved to a file for efficiency, and then fed into a Long Short-Term Memory (LTSM) CNN, which made use of PyConv blocks in each of its convolutional layers. The PyConv blocks consisted of convolutional layers with differing kernel sizes, and acted as initial feature extractors for the model. Numerous additional enhancements for both training efficiency and accuracy were implemented within the model and training procedure to ensure usable results were generated.

GloVe Embeddings

GloVe embeddings are pre-trained word embeddings that aim to extract the semantic relationships between words. This relationship exists at many different levels, such as in-between words directly next to each other, and words spread out between numerous sentences. GloVe embeddings assign word representations to each token that are sensitive to the semantic similarities shared by certain words, allowing it to provide a more accurate representation of the associated values and meanings placed on words.

In this project, GloVe embeddings are used as the first layer in the model, to pre-process all of the data coming in. During pre-processing, each token contained in a given review is mapped to its calculated ”GloVe vector”, which represents the information contained in that token. Each of these vectors are then strung into a sequence of embeddings, which becomes the input to the model that is to be trained.

For this specific task of sentiment classification, the GloVe embeddings are crucial for extracting the nuance in the reviews sourced from IMDb. As previously mentioned, there are many literary elements at play in these reviews, so working from GloVe vectors rather than the physical words within the reviews takes much of the burden off of the neural network. In other words, the responsibility of determining the similarity between ”King” and ”Knight” is given to the GloVe embeddings, rather than to the neural network, allowing it to focus on the task of sentiment recognition more effectively.

PyConv Blocks

As discussed earlier in this paper, PyConv blocks offer tremendous value in the area of feature extraction, due to the multiple perspectives acquired by the numerous convolutional layers of varying sizes. Each of these kernels effectively has a different perspective on the data, allowing the network to create more accurate and generalizable results.

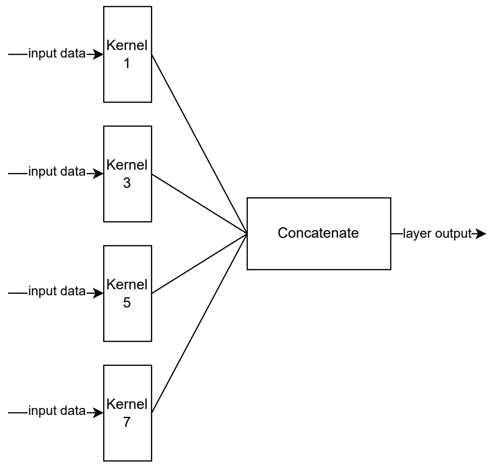


Figure 4: The above figure shows how the PyConv blocks are structured. Each PyConv blocks consists of 4 convolution kernels, each of which existing at a different kernel size. As mentioned previously, this is crucial for the feature extraction capabilities of the network. The more that can be extracted in this process, the less the fully connected layers need to do. The output of these kernels is concatenated, and sent to the next layer.

Within our implementation, each PyConv block consists of four parallel convolutional layers, specifically with kernel sizes of 1, 3, 5, and 7. Smaller kernels, such as 1 and 3, help the model understand patterns at a highly-localized level. These can be short idioms and other phrases that involve multiple tokens. Larger kernels, such as 5 and 7, can identify patterns in larger sets of information, such as phrases or sentences that span a large amount of tokens.

CNN-LTSM Hybrid Model

In this project, our PyConv blocks are used within the larger structure of our model, which is a hybrid CNN-LTSM model. This architecture aims to reap the benefits of both CNNs and LTSMs in order to properly analyze text.

The CNN component, equipped with the implemented PyConv blocks, serves as the main feature-extractor for the model. This CNN layer is fed the output from the GloVe-embedded text input, which allows it to perform with a higher accuracy, and consider the nuanced context associated with each of the tokens its fed. The output from this CNN layer represents the original information from the GloVe embeddings represented in a way that showcases the spatial and contextual relationships between the tokens in the input text. This output allows the following layers of the model to perform their analysis to a higher degree.

Taking the output from the CNN layer as its input, the LTSM layer. This layer is designed to process sequential data, which in this case, is the convoluted output from the series of token embeddings generated by GloVe. This layer is intended to recognize patterns in how these GloVe vectors

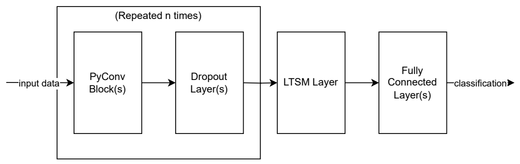


Figure 5: The above figure details how the PyConv blocks are fit into the overall network architecture. Each PyConv block is fed input data (or the data from the previous PyConv Layer), and is assigned its own dropout layer. The dropout layer is implemented to reduce the over-fitting of the model, by "dropping-out" certain neurons in a given epoch. The output from all of the sequential PyConv layers and their respective dropout layers is then fed into the LTSM layer, followed by the fully connected layers. The output of the last fully connected layer is the classification prediction of the model.

interact sequentially. The output of this layer is mapped directly to our classification output, which represents the predicted sentiment for the given movie review.

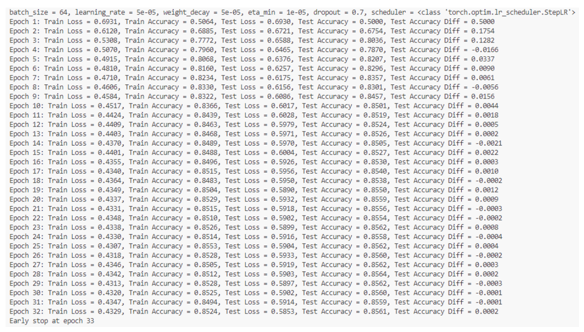
Training Strategy

The training strategy used for this project was designed to optimize results and reduce training time. In order to prevent over-fitting and keep our results generalizable, dropout regularization was added to the network architecture. In addition, a learning rate scheduler, as well as an optimizer were added to the main training loop. A reasonable batch size of 64 was decided upon to optimize both results and training time, and the training loop was aimed to hit 50 epochs, though early dropout was added to prevent stagnation in the results.

Dropout Regularization is achieved by adding "dropout" layers throughout the network architecture. These layers randomly ignore a fraction of neurons during the training process, forcing the model to not assign too much weight to any specific set of neurons. This generally creates more generalizable results, and avoids over-fitting of data during the training process.

A Learning Rate Scheduler adjusts the learning rate of the model as the model trains. As the model performs Gradient Descent on its large number of parameters, it will gradually get closer and closer to a local minima. In order to properly approach this minima without overshooting it, the learning rate will have to be gradually decreased as the number of epochs increases. While there are more sophisticated learning rate schedulers that will adjust the learning rate when stagnation is detected, this project made use of the StepLR scheduler. This scheduler is simple in design, simply decaying the learning rate by a set amount after a set amount of epochs.

This project made use of the AdamW optimizer. This optimizer is a modified version of the Adam optimizer, which is an optimization on the traditional Stochastic Gradient Descent (SGD) model. The AdamW optimizer includes an L2 regularizer, which effectively penalizes large weights within the model.



```
batch_size = 64, learning_rate = 5e-05, weight_decay = 5e-05, eta_min = 1e-05, dropout = 0.7, scheduler = <class 'torch.optim.lr_scheduler.StepLR'>
Epoch 1: Train Loss = 0.6931, Train Accuracy = 0.5064, Test Loss = 0.6930, Test Accuracy = 0.5000, Test Accuracy Diff = 0.5000
Epoch 2: Train Loss = 0.6120, Train Accuracy = 0.6885, Test Loss = 0.6721, Test Accuracy = 0.6754, Test Accuracy Diff = 0.1754
Epoch 3: Train Loss = 0.5308, Train Accuracy = 0.7772, Test Loss = 0.6588, Test Accuracy = 0.8036, Test Accuracy Diff = 0.1282
Epoch 4: Train Loss = 0.5070, Train Accuracy = 0.7960, Test Loss = 0.6465, Test Accuracy = 0.7870, Test Accuracy Diff = -0.0166
Epoch 5: Train Loss = 0.4915, Train Accuracy = 0.8068, Test Loss = 0.6376, Test Accuracy = 0.8207, Test Accuracy Diff = 0.0337
Epoch 6: Train Loss = 0.4810, Train Accuracy = 0.8160, Test Loss = 0.6257, Test Accuracy = 0.8296, Test Accuracy Diff = 0.0090
Epoch 7: Train Loss = 0.4710, Train Accuracy = 0.8234, Test Loss = 0.6175, Test Accuracy = 0.8357, Test Accuracy Diff = 0.0061
Epoch 8: Train Loss = 0.4606, Train Accuracy = 0.8330, Test Loss = 0.6156, Test Accuracy = 0.8301, Test Accuracy Diff = -0.0056
Epoch 9: Train Loss = 0.4584, Train Accuracy = 0.8322, Test Loss = 0.6086, Test Accuracy = 0.8457, Test Accuracy Diff = 0.0156
Epoch 10: Train Loss = 0.4517, Train Accuracy = 0.8366, Test Loss = 0.6017, Test Accuracy = 0.8501, Test Accuracy Diff = 0.0044
Epoch 11: Train Loss = 0.4424, Train Accuracy = 0.8439, Test Loss = 0.6028, Test Accuracy = 0.8519, Test Accuracy Diff = 0.0018
Epoch 12: Train Loss = 0.4409, Train Accuracy = 0.8463, Test Loss = 0.5979, Test Accuracy = 0.8524, Test Accuracy Diff = 0.0005
Epoch 13: Train Loss = 0.4403, Train Accuracy = 0.8468, Test Loss = 0.5971, Test Accuracy = 0.8526, Test Accuracy Diff = 0.0002
Epoch 14: Train Loss = 0.4370, Train Accuracy = 0.8489, Test Loss = 0.5970, Test Accuracy = 0.8505, Test Accuracy Diff = -0.0021
Epoch 15: Train Loss = 0.4401, Train Accuracy = 0.8488, Test Loss = 0.6004, Test Accuracy = 0.8527, Test Accuracy Diff = 0.0022
Epoch 16: Train Loss = 0.4355, Train Accuracy = 0.8496, Test Loss = 0.5926, Test Accuracy = 0.8530, Test Accuracy Diff = 0.0003
Epoch 17: Train Loss = 0.4340, Train Accuracy = 0.8515, Test Loss = 0.5956, Test Accuracy = 0.8540, Test Accuracy Diff = 0.0010
Epoch 18: Train Loss = 0.4364, Train Accuracy = 0.8483, Test Loss = 0.5950, Test Accuracy = 0.8538, Test Accuracy Diff = -0.0002
Epoch 19: Train Loss = 0.4349, Train Accuracy = 0.8504, Test Loss = 0.5890, Test Accuracy = 0.8550, Test Accuracy Diff = 0.0012
Epoch 20: Train Loss = 0.4337, Train Accuracy = 0.8529, Test Loss = 0.5932, Test Accuracy = 0.8559, Test Accuracy Diff = 0.0009
Epoch 21: Train Loss = 0.4331, Train Accuracy = 0.8515, Test Loss = 0.5918, Test Accuracy = 0.8556, Test Accuracy Diff = -0.0003
Epoch 22: Train Loss = 0.4348, Train Accuracy = 0.8510, Test Loss = 0.5902, Test Accuracy = 0.8554, Test Accuracy Diff = -0.0002
Epoch 23: Train Loss = 0.4338, Train Accuracy = 0.8526, Test Loss = 0.5899, Test Accuracy = 0.8562, Test Accuracy Diff = 0.0008
Epoch 24: Train Loss = 0.4330, Train Accuracy = 0.8514, Test Loss = 0.5916, Test Accuracy = 0.8558, Test Accuracy Diff = -0.0004
Epoch 25: Train Loss = 0.4307, Train Accuracy = 0.8553, Test Loss = 0.5904, Test Accuracy = 0.8562, Test Accuracy Diff = 0.0004
Epoch 26: Train Loss = 0.4318, Train Accuracy = 0.8528, Test Loss = 0.5933, Test Accuracy = 0.8560, Test Accuracy Diff = -0.0002
Epoch 27: Train Loss = 0.4346, Train Accuracy = 0.8505, Test Loss = 0.5919, Test Accuracy = 0.8562, Test Accuracy Diff = 0.0003
Epoch 28: Train Loss = 0.4342, Train Accuracy = 0.8512, Test Loss = 0.5903, Test Accuracy = 0.8564, Test Accuracy Diff = 0.0002
Epoch 29: Train Loss = 0.4313, Train Accuracy = 0.8528, Test Loss = 0.5897, Test Accuracy = 0.8562, Test Accuracy Diff = -0.0003
Epoch 30: Train Loss = 0.4320, Train Accuracy = 0.8525, Test Loss = 0.5902, Test Accuracy = 0.8560, Test Accuracy Diff = -0.0001
Epoch 31: Train Loss = 0.4347, Train Accuracy = 0.8494, Test Loss = 0.5914, Test Accuracy = 0.8559, Test Accuracy Diff = -0.0001
Epoch 32: Train Loss = 0.4329, Train Accuracy = 0.8524, Test Loss = 0.5853, Test Accuracy = 0.8561, Test Accuracy Diff = 0.0002
Early stop at epoch 33
```

Figure 6: The above figure shows the output from a successful run of the training algorithm for the model. This shows the model configured as specified in this document. As can be seen, the testing accuracy stabilizes around 85%, the model detects this, and initiates an early stop. This preserves the accuracy of the model, as it will prevent it from becoming over-fit.

Experimental Results

The model and hyper-parameters detailed in this paper was able to achieve an overall 85% testing accuracy on the IMDb sentiment dataset. As shown in Figure 7, this was the highest possible stable test accuracy that was able to be achieved with the model architecture described. And as shown in Figure 8, 85% was indeed the proper plateau for the model as described.

While these results are acceptable given the training time constraints, there are optimizations to be made. The model is limited by its depth and pre-processing strategy. While it is entirely possible to get good results with GloVe pre-processing, it is not tailor-made for this model. There are additionally more precise pre-processing algorithms available that could potentially increase performance. Due to the constraints with GPU resources and training time, the batch size and overall model depth both needed to be reduced. A larger model would theoretically be able to handle more patterns in the dataset, allowing for a convergence in testing accuracy at a higher percentage.

Some specific areas for improvement lie within complex semantic structures, such as sarcasm. The model, in many cases, is unable to deal with these cases, often giving them an incorrectly predicted positive result. Additionally, the model struggles with reviews of varying sizes. In adding additional testing data, it became apparent that since most of the IMDb reviews were at least a paragraph in length, the model had become specialized for that text size. When inputting shorter text (a couple sentences) or larger texts (a couple paragraphs), the model struggled to return accurate results. These issues can be resolved with more training data dealing with specific issues, such as sarcasm, and more training data with examples of shorter and longer reviews.

Despite having an ambitiously small network depth for a text-processing project of this nature, our model was able to deliver significant results. With a larger model, more thorough data, and expanded pre-processing models, a PyConv architecture could be very useful in the field of text analysis.

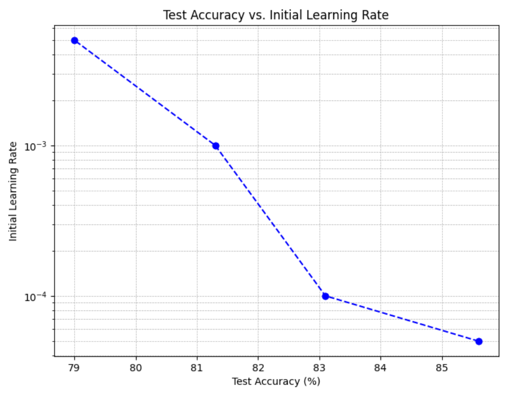


Figure 7: The above figure shows the initial learning rates compared with the achieved testing accuracy after the models had stabilized. While many of the higher initial learning rates achieved higher than 85% testing accuracy for certain epochs, they soon became over-fit and displayed erratic results. The only repeatable result was generated by an initial learning rate of 5e-5.

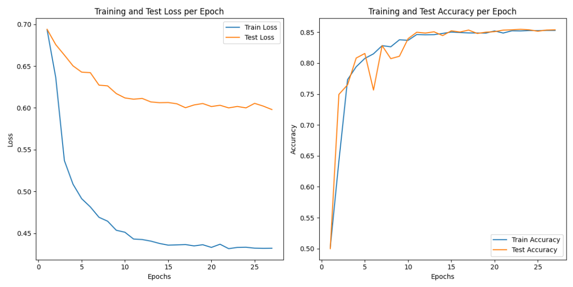


Figure 8: The above figure shows the training and testing loss and accuracies over the epochs for a succesful training run of the model. The training loss decreases at a much more significant rate (as to be expected) than the testing, however, both the testing and training accuracies move at around the same rate, converging at 0.85.

Conclusion

After designing the model architecture, and tweaking the hyper-parameters to effectively train it, the model was able to determine the sentiment of an IMDb review with an 85% accuracy. Without using more sophisticated models for preprocessing, such as BERT, RoBERTa, or DistilBERT, this result is in-line with models of similar complexity. In order to truly improve accuracy for this dataset, the network size would need to be increased so that it could store the patterns and generalizations required to fully interpret the IMDb reviews.

After experimenting with multiple different augmentations of the same network architecture, the results were most repeatable with 4 PyConv layers, a dropout layer, an LTSM layer, and 2 fully connected layers. The PyConv layers brought the size of the data from the dimension of the embedded vectors to 64. This output was then fed into the LTSM layer, followed by the two fully connected layers. In between each layer was a dropout layer, which had a 70% chance of ignoring the neurons within that layer.

Although some implementations of this architecture experimented with smaller batch sizes, such as 32 or even 16, smaller initial learning rates, such as 1e-5, and different learning rate schedulers, such as CosineAnnealingWarmRestarts, the most reliable results were found with the configuration documented in this paper. The smaller batch sizes in some cases produced testing accuracies as high as 90%, however, these results proved not repeatable when run multiple times back to back. Additionally, the CosineAnnealingWarmRestarts scheduler provided great initial results in the training process, however, as the epochs increased, the inherent oscillation introduced by the scheduler made results unstable, with testing accuracies increasing and decreasing sharply from epoch to epoch. Lower learning rates caused the model to never converge, since the initial steps took it nowhere near a global minima.

All in all, this implementation provides a strong case for the use of Pyramidal Convolution in the context of text analysis. The increased context and angles provided by the PyConv have proven to not only be useful in the application of spatial analysis in images, but also in the analysis of contextual relationships between words and sentences in text. The PyConv architecture in this model fed its fully connected layers valuable information extracted from the GloVe embeddings it was given. Without the use of this architecture, more of the burden of discovering these patterns would be placed on the fully connected layers, which would create an unnecessary need for a larger model. By offloading this task to any convolutional layer, in this case PyConv, the fully connected layers are able to perform their job to the fullest capacity, and produce the best possible results.

References

[1] Ionut Cosmin Duta, Li Liu, Fan Zhu, and Ling Shao. Pyramidal Convolution: Rethinking Convolutional Neural Networks for Visual Recognition. *arXiv preprint arXiv:2006.11538*, 2020.

[2] 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. URL: http://www.aclweb.org/anthology/D14-1162.

[3] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. URL: http://www.aclweb.org/anthology/P11-1015.

Links

Google Drive

This Google Drive folder contains my code implementation for this paper, as well as my video explaining it.

Google Drive Folder