ECE 657 Assignment 3

Design Report

Group 49

Pawel Jaworski, Ted Themistokleous

Question #1 Convolutional Neural Network for Image Classification

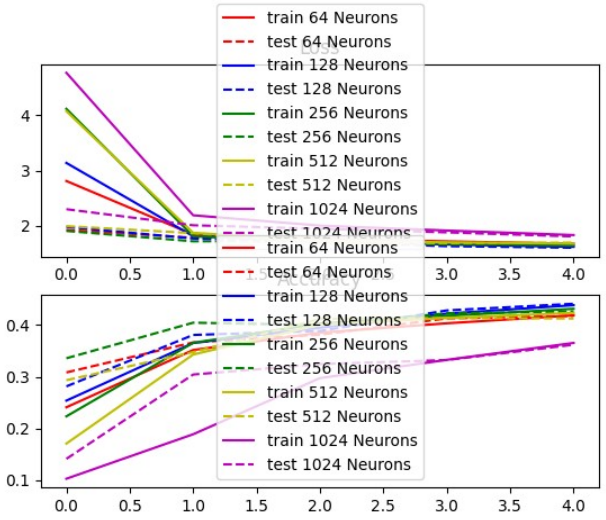
Dataset Preprocessing:

The data set was preprocessed the data using one hot encoding for each output target class, this allowed us to structure out output layer to the number of desired classes we wish to discriminate between in the data set. Further processing was done to normalize each colour channel so that each stream could be fed as inputs to the neural network structures created. If normalization is not done on the input RGB channels for each image, the training would be slowed down significantly. Without normalization training may even not come to a useful solution, due to how neural networks operate better on smaller weight values since gradient decent is used during the training process

Output Layer Selection:

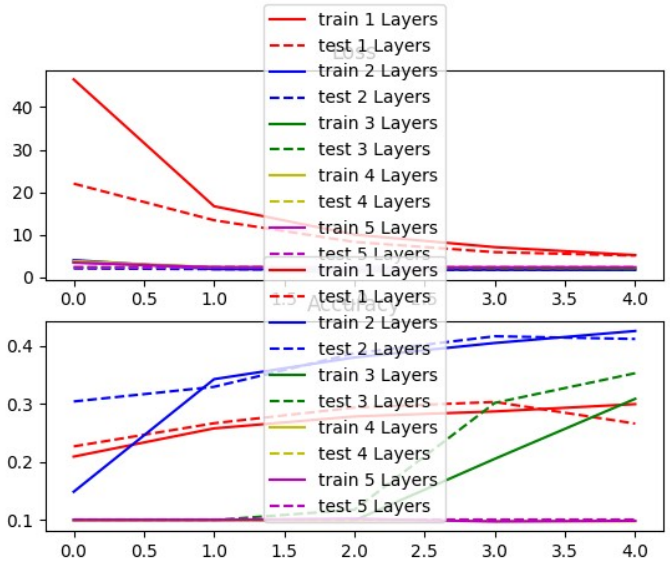
The output layer chosen was a soft-max for each output, which is a generalized form of logistic regression. This was chosen for its usefulness in classification problems, since once trained correctly will fire one node only for the desired class trained on. By combining a convolutional input layer with this choice of output layer, it allows for a way to effectively discriminate between each image. This leverage the one hot encoding output vector generated from initial data set pre-processing which we can use to verify our result in the test section.

MLP Training Varying Layers and Neuron counts:



By varying the number of neurons per layer for the MLP network, there was an observed difference in performance accuracy to a point. When varying the Neuron count from 64 to 128, the train and test accuracy increased only just over 0.42 . Once more than 128 neurons are used then the accuracy decreases. There is a change for all the networks at the initial epoch. For the case of 1024 neurons, the accuracy is the worst, resulting in an accuracy of around 0.35, less than the initial smallest neuron count of 64. This is consistent when using layers larger than 128 for the data, and implies that with more than 128 neurons there is over fitting occurring in the MLP model in the five epochs. This outcome consistent for both accuracy and loss of the network. 128 Neurons for train/test seem to have the best loss with larger values of neurons having worse loss performance (higher loss) than predecessor layer sizes This is consistent with accuracy.

When varying the amount of layers, the impact was not as obvious. When increasing from one to two layers the accuracy increases substantially for train and test sets. Adding more layers past two, begins to reduce accuracy such that for 5 layers, only 10% accuracy is observed. Things also converged slower with more layers. In fact, there is a delayed change in accuracy and loss with more layers. For 3 layers, the accuracy changed the most in the 2nd and 3rd epoch and continued to climb. This implies that with more epochs potentially 3 layers may prove to be better



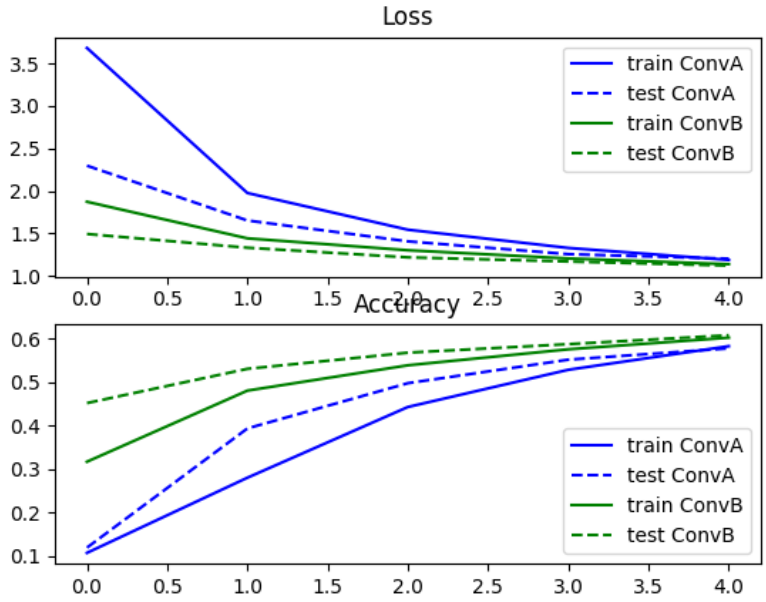
For the loss on the effect of layers is quite different. For having more than one layer, the loss converges to a value below 10. It still appears that two layers is optimal over 5 epochs for training with 5 layers being slightly worse in the 5 epoch window. This delayed effect is also seen which makes sense as more layers would mean more values needed to train to reach a stable result.

Train/test accuracy for MLP, and CNNs:

The MLP performed worse than the two CNNs with an accuracy of around 40% for test and val. The first CNN, CNN1 had an accuracy of around 55% for training and validation whereas CNN2 had an accuracy just over 60% for training and validation. The loss characteristics for both CNN1 and CNN2 are also similar. Losses followed a similar trend with CNN2 having the lowest loss, compared to CNN1, which had a better loss then the MLP implementation. The CNNs performed better as the convolution layers were able to effectively create a down sampled representation of the data for CNN1, and in CNN2 the additional max pooling layers helped improve accuracy combined with dropout. The dropout applied to the fully connected layer for CNN2 allowed better regularization of the training data such that each node learned specific features instead of all nodes learning attempting to learn the same features. CNN2 leverage max pooling to

The MLP performed worse since it takes in every pixel, and tries to learn the features of each pixel in the weights, and then output classification result. Since there is so much data needed this process doesn't converge learns some but not all general features like the CNNs generate when performing the convolution to work off less but more feature rich data.

Difference in CNN models:



CNN1 and CNN2 performed better compared to the MLP but CNN2 had around a 5% jump between CNN1. This was expected as a max pooling layer between each layer in CNN2 allowed for a better representation of important features to learn for the fully connected layer, opposed to running straight convolution layers into a fully connected. The CNN2 would be expected to converge faster to the final result but tapper off as seen after 5 epochs. CNN1 may converge to the same accuracy but with more epochs since it would need more epochs to converge to the same features that pooling extracts. CNN1 took about 3 minutes for epoch, whereas CNN2 took 20 seconds per epoch during training. In terms of training time, CNN1 took much longer than CNN2 as each feature is pooled into a smaller feature map thats used for the next layer since we have a ((512 – 64)/3 + 1) = 150x150 after the first pooling and approx 30x30 feature map after the final pooling layer. This would explain the increased speed up for the CNN2

Improvements to the network:

CNN2 could be improved by adding additional convolution layers and decrease the filter size per layer combined with average pooling instead to average each new map generated between layers. An example would be to run 64, 32, and then 16 over the image with varying strides. This would allow to extract features gradually averaging the most important features between generated maps instead of taking the maximum value. Another improvement would be to reduce the node count in the final fully connected layers by a factor of two until accuracy begins to decrease. By having two fully connected layers at the end on reduced feature maps, there may be a slight over-fitting to the filtered features of the original CNN2 since we're getting 64x64 image maps as the end result instead of a 512x512 image. Combining the above two improvements into on network should give a better accuracy than CNN2 while not adding too much additional training time.

Question #2 Recurrent Neural Network for Regression

Dataset Preprocessing:

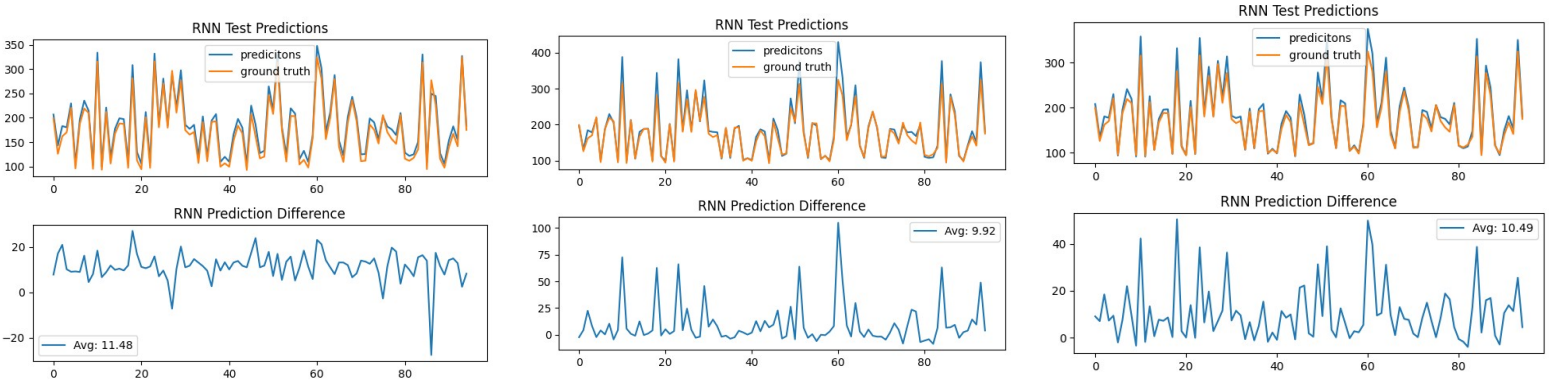
The input data was preprocessed using standardization, implemented by the scikit-learns minmaxscaler for the dataset. This allows the data to be captured around the average of the chosen small time horizon of stock price making the data more Gaussian for training features .

Design Methodology:

An initial design of using a vanilla long term short term (lstm) RNN architecture was used and then tested since stock data reflects a set of time series data. The initial system was chosen to use a relu activation function and the Adam optimizer in order to minimize the mean squared error (MSE).

Next batch normalization was tried to the vanilla model. Tuning resulting in adding additional parameters for beta and game initialization with an epsilon of 0.001. This resulted in better performance of the stock lstm with 50 neurons in the initial layer

Layers were then added to to the batch normalization lstm to explore if they had a positive effect on performance which showed varying gains. 2, 4, 8, 10 layers were tried for 1000 epochs before finding that the optimal amount of layers to be around 4 to 8. This resulted in picking 6 to be the best amount of layers for this problem after additional testing . The average error between the actual stock price and predicted price was used to quantify the best amount of layers, in which 6 had an average of 9.92. Seen below is the iterations for 4, 6 and 10.

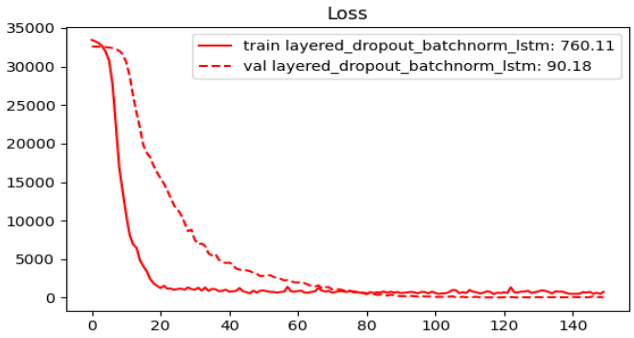


Outcome of trying different layer sizes between LSTM models for stock prediction. 6 layers performed the best (center), where 4 (left) and 10 layers(right) performed worse.

The amount of neurons per layers was then adjusted between 25 to 200 combined with and without dropout between (0.1 to 0. 5) for each set of varied neuron sizes. Dropout was added to test if removing connections during training improved performance and to avoided neurons from memorizing stock data instead of learning desired features.

Final Design and testing:

The outcome of this testing resulted in the final network configuration of batch normalized lstm with 0.2 dropout, 6 layers, and 150 neurons per layer. The loss function had a total loss of 90.19 on validation as seen in the figure below This was achieved with 150 epochs, a batch size of 32, and a train/validation split of 80%-20%. The final loss is shown below of the resulting network



Using more days of features:

If more days of features were added, this would theoretically improve accuracy as there would be more data for the network to train on and pickup additional temporal features up to a limit. With an infinity, or very large time horizon for stock data, this could aid in giving better results at the cost of potentially increasing node and layer counts to accommodate for the additional data to model the longer term time relationships between early and late points in the data set. This would increase training time, while also the amount of memory needed to represent the network.

Question #3 - Natural Language Processing on Imdb dataset

NLP was used to process the Imdb data set using a supervised learning approach to perform sentiment analysis on whether a film is good or bad based on the review text rating. The steps used to do this are as follows:

- Load and pre-process all the data (case, punctuation, html-tags, tokenize, stopwords, lemmatize, stemming)
- Use both train and test data to generate an word2vec model for word embedding
- Convert sentences to vector representation of our original sentences
- Embedded sentences are then padded and used to train a final classifier for sentiment

This design flow was used to come up with a unique vectorization model of the words in the dataset that carried context. Sentiment was one-hot encoded for good = [1 , 0] and bad = [0, 1] since the desired goal is to classify between good and bad reviews

Preprocessing

Preprocessing is done to "clean" the data of irregularities that would otherwise skew metrics when creating a word2vec representation of the entire corpus. The initial stage of putting everything to lower case removes, duplicate upper case characters that would otherwise appear to the vector model as a separate word thus reducing the vector space needed for our generated embedding model. The start of every sentence for example would thus result in an additional word and skew counts in the continuous bag of words or skip-o-gram methods used during vectorization

Next, tag removal, scans the document for unwanted html/unwanted tags. An example of this are like break characters
 that add tags as unwanted "words". This needs to be done before punctuation cleaning otherwise the tags internal value will be appending to the word before it erroneously. Examples seen were "badbr", "terriblbr" which skewed cosine distance metrics when finding the top 5 similar words to "horribl", with bad, terribl being the other "close" words in the calculation.

Punctuation is important as it strips sentences of values that would skew our tokenization on spaces between words. Commands, periods and other pause/end of sentences are remove to aid in splitting each review,

Tokenization is them performed to generate a list of vectors that splits each word based on the spaces between them. This allows the word2vec algorithm to quickly iterate over each word when taking frequency counts before coming up with an embedding model to use.

Stop words are removed from the list of words after, in order to remove words that provide no overall context to the data set. Words such as "the, and, an" show up many times in English but appear before many different nouns and verbs in sentence. Without this step, we would add additional words and imply context.

Lemmatization and stemming are done as a final step to get the "base" word between similar words that represent the same context. This is done to reduce whats called "inflectional forms" of

words. An example of this is move, moving, moved, which come from the base word “mov’. Stemming chops off the end of the word (ing, ed, e) and represents the word in a simpler form where as lemmatization makes use off the inflectional context or “tense” of the word before we cut off the ending with stemming to produce a base word response.

NTLK was used for stemming/lemmatization using the Porter stemmer produced the best results during testing. Additionally NLTK was used for stopword removal using the stopwords corpus from the library and tokenization leveraged the ntlk tokenize() functionality. The other filtering stages used base python functionality. The

Generating Word Vectors for Embedding

Word2Vec was used to perform a continuous bag of words (CBOW) over the data set to infer context for words before and after each input word. This model uses a distributed representation of the context based on the words around a given input word. The tools used here from the gensim 4.0.0 package implemented word2vec for a CBOW design architecture that was able to be tuned on the data. Both the training data, and test data was used to perform the bag of words to generate a vector representation of each contextually relevant word. For our model we chose This results in a vector representation of each word using an adequately sized feature vector. In our design we took the average review length and used that as the feature vector size for our entire corpus, and tuned the learning and min learn rates while using a 10 word minimum word frequency count before an item was used. The learning rate used was 0.28, with a feature vector size of 119 (average word count in all reviews after pre-processing) with a negative sampling rate of 20, and window of 5. This gave the best set of embeddings after a couple of iterations.

Embedding Sentences

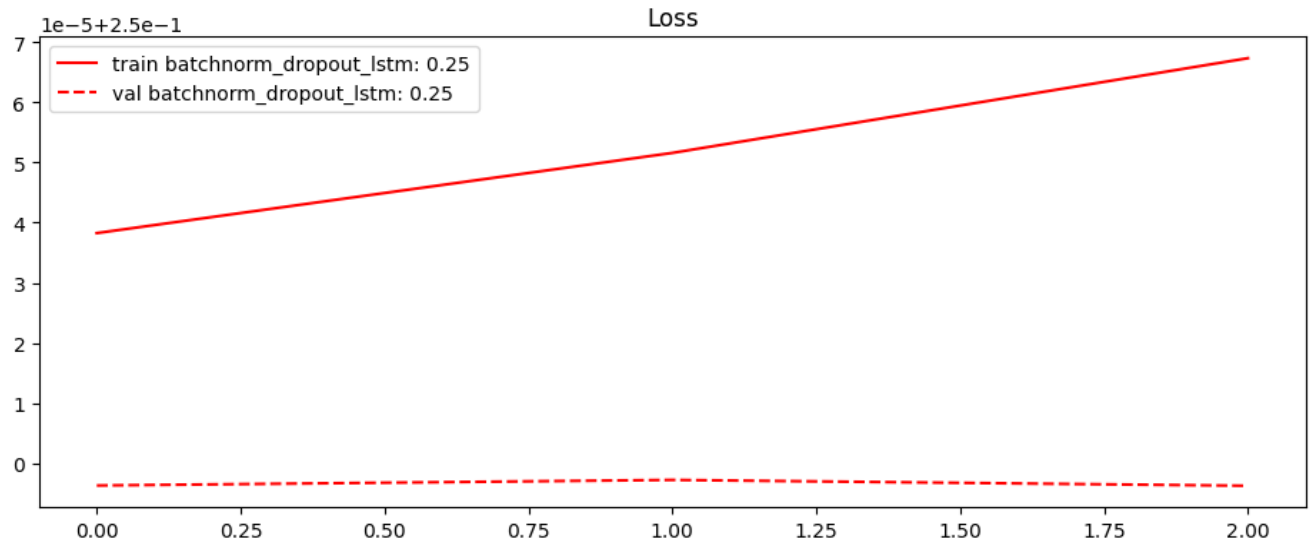
The stored sentences were then run over with the embedding model to generate a set of review vectors that then become training and test data sets prior to training the resulting end classifier.

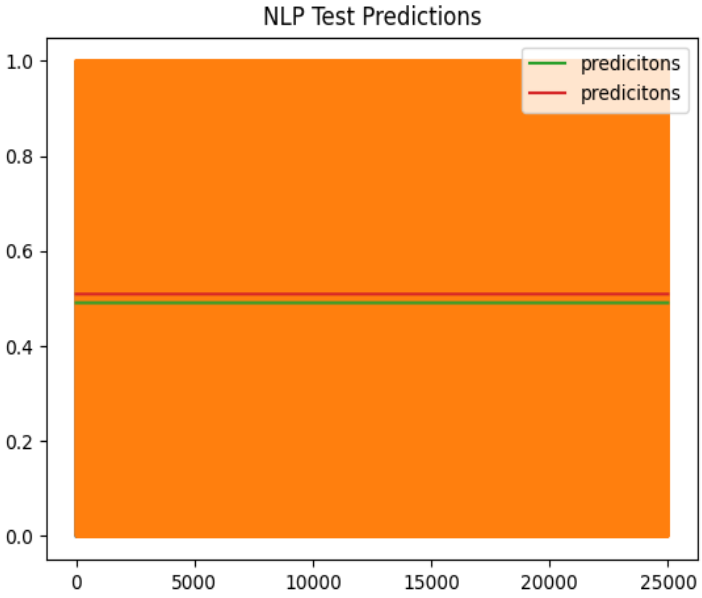
1D convolutional lstm layer for classification:

The final layer chosen for classification was a 54x5 relu activated 1D convolutional neural network with an 75 node lstm relu activated layer and a softmax output layer. After various methods of trial and error on the streamed word vectors, convolution layers were used for the large vector data sizes, whereas the lstm portion used to capture space based information between words in sentences. The final layer was chosen as the desired goal is to classify between good and bad sentiments from the output of the distance between vectorized words.

Final Result

The final resulting output resulting in a loss of 0.25 using this resulting data between train and test data below with the resultign predictiosn for the 25k test set.





This result seems slightly suboptimial as we're able to perform around half the accuracy of positive and negative sentiment with the reviews based on the word embeddings. This unfortunetly results in poorer performance from other state of the art solutions.

Possible improvements

Possible improvements would be to increase the size of the word vector feature space by using larger vectors and trying a different set of calssifiers (decision tree)