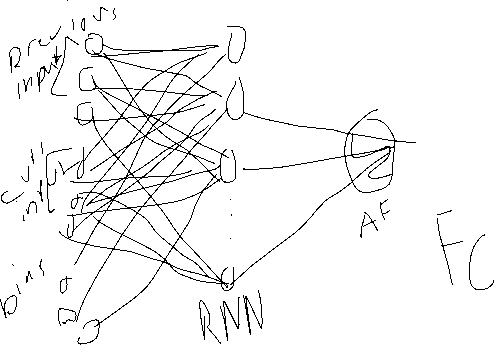
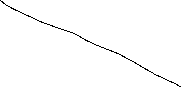
Part 1:

The dataset that I chose was the Metro Interstate Traffic Volume from the UCI Machine learning repository. This dataset showcases the hourly traffic volume of I-94 in Minneapolis, MN based on weather features and holidays from 2012-2018. This is a multivariate time series dataset. Furthermore, there are 48,204 rows and 9 columns. 3 continuous columns, traffic volume, clouds all, temperature, 1 date column, date time, and 5 categorical columns, weather main, weather description, snowfall, rainfall, and holiday. Lastly, the date column is a timestamp in a yyyy-mm-dd hh:mm:ss format where the date starts at 2012-10-02 09:00:00 and ends at 2018-09-30 23:00:00.

Link: <https://archive.ics.uci.edu/dataset/492/metro+interstate+traffic+volume>

Part 2:

I created an array where the data ranged from 0 to 25 and contained 2,200 data points in between those values. Next, I created a sin wave based on those values, and created a data frame where the first columns contained the datapoints that ranged between 0 to 25, and the other column where sin is performed on those data points. I split the data into a training and testing in a 80/20 split, and then scaled it using the min max technique. After that, I created model that consisted of a simple rnn layer with 20 neurons followed by a fully connected layer with one output where the sequence length was 70. The optimizer was adam and the loss function calculated was mean square error. I fitted the RNN model using a time series generator that contained that scaled training data with a batch size of 1 at 100 epochs. The loss stayed very close to zero for most of the epochs. Furthermore, I predicted the next datapoints using the model, and then plotted the results. Unfortunately, the predicted line and actual line were not the same, the actual line increased, decreased and then increased, while the predicted line remained constant between the ranges 0.91 and 0.96. This indicated that the model severely overfitted, some possible suggestions to improve the model would be adding a regularization technique to prevent overfitting. For example, I could use a dropout layer in between the Simple RNN layer and the dense layer.



Part 3:

Question 3.1: Vanishing gradient happens during backpropagation of a neural network model where the gradients keep getting smaller and smaller to the point where the model cannot learn anymore.

A graph of function and function

Description automatically generated

Techniques like switching the optimizer to relu or using gradient clipping can prevent vanishing gradients from happening. Exploding gradient happens during backpropagation of a neural network when the gradients remain large which prevents the model from converging to an optimal solution. Regularization and gradient clipping can prevent exploding gradients from happening.

A diagram of a graph and a diagram of a graph

Description automatically generated

Question 3.2: The limitations of a SimpleRNN is mainly the vanishing and exploding gradient issue. SimpleRNNs can’t remember long sequences because the gradients get smaller as the model backpropagates making the RNN model struggle to learn long-term sequences. On other hand, simpleRNNs may experience exploding gradients because the model is heavily biased towards those short-term sequences making the gradients extremely large and unstable.

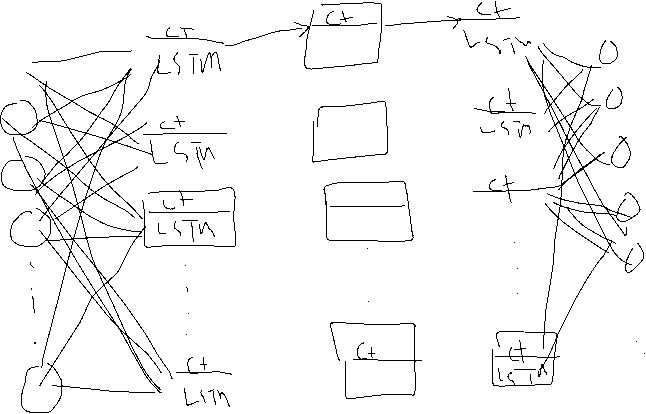
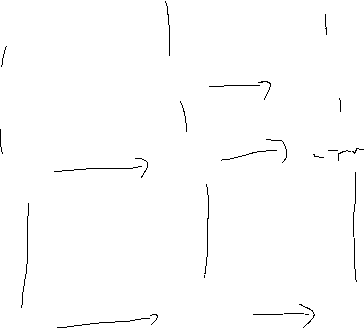
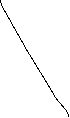
Question 3.3:

LSTM models are a powerful solution to both the vanishing and exploding gradient problem. This is because of the LSTM’s implementation of the cell state and the gates that monitor what goes into the cell state. Furthermore, the cell state allows the model to retain longer sequences and helps LSTM model capture long term dependencies. There are 3 types of gates that monitor what information goes into the cell state: the forget gate, which decides what information of the previous cell state should be removed, and what information should be kept. Then the input gate, which determines what information should be added to the current cell state. Then lastly, the output gate, which determines what information from the current cell state should be added to the current hidden state.

A diagram of a long short term memory

Description automatically generated

Part IV:



Summary of Core Parameters:

11 input features

Sequence length = 95

3 lstm layers 50 neurons each where activation function was relu

Dropout layer of 20% after the first and second lstm layer 1

1 fully connected layer with one output

Optimizer was adam

Loss = mse

Performance metric = rmse

Epoch = 10

Batch size = 10 (for training), 1( for testing)

Report:

For this LSTM model project I used column traffic volume to predict traffic volume of the next hour. Since, weather main column consisted of 11 classes there were one hot encoded into 11 columns. The data was split in a 90/10 split into testing and training, where training had 90% of the data and testing 10% of the data and last 95 data points training of the data included in them. When fitting the LSTM model, it took about an average of 2 minutes per epoch to train. Where the loss hovered around 0.01 majority of the time. . Then, I evaluated the model using the testing and achieved an rmse score of 719.27, indicating that there is some strong signs of overfitting. After that I retrained the model using all of the dataframe besides the last 95 datapoints, and then forecasted the next 95 datapoints with it. I obtained a rmse score of 1187.31.

Part V:

To improve the model I suggest adding complex elements along with more regularization. I am deciding to add the 1d convolution layer to introduce weight sharing element to the rnn which can help the model learn faster along with a maxpooling layer of 2x2 which can improve feature extraction. I believe that adding these two elements will significantly improve the model’s loss and generalization. I am also, removing a lstm layer and then adding more neurons to first and second layer. Then, I’ll increase the dropout by 10 percent for each dropout layer that comes after the lstm layer, and then use a dropout of 50% after the maxpooling layer. Lastly I’ll add an extra dense layer of 25 neurons which is then followed by another dropout layer, and then another dense layer with one output. I believe that this should not only be able to capture the complex behavior of this time series data, but also increase computational efficiency and generalization.

Summary of Core Parameters:

N\_features =1

Conv1d = filters 25, kernel\_size = 5x5 stride of 2 and relu

Maxpooling layer of 2x2

Dropout of 50%

LSTM 1 150 units and relu

Dropout of 30%

Fully connected layer of 25 neurons

Drop out of 25%

Fully connected layer with one output

Optimizer = adam and loss = mse

Performance metric = rmse

Batch size = 10 (training) 1 (for testing)

Report

When fitting the training data, the loss hovered around 0.023 with an average epoch of 35 seconds. Next, I evaluated the model with the testing data and obtained an rmse score of 973.42. After that I retrained the new model using all of the dataframe besides the last 95 datapoints, and then forecasted the next 95 datapoints with it. . When assessing the predicted 95 data points to the actual datapoints, I obtained a rmse score of 1108.33. When assessing the full dataset it model generalized worse than the testing. obtained a rmse score of 1108.33. It took about 35 seconds per epoch when fitting. The full dataset results ended up generalizing worse than the testing for this model.

Part VI:

When comparing the first model to the second model, there were some different outcomes. For example, the loss for the first model was significantly less than second model, the first model generalized better with the testing data than the second model. However, when it came to forecasting, the second model performed better than the first model, even though the second model still had a larger loss per epoch when training the full dataset. Also, the second model trained a minute and 30 seconds per epoch than faster the first model for both fitting the training data and the full dataset.

Part VII:

The purpose of this final project to is to apply my knowledge of Recurrent Neural Networks (RNN) to two different datasets. One will be based on Sine waves and the other will be based on a time series dataset that I retrieved from the internet that was not used in class. The two types of RNNs that I focused on for this final project was SimpleRNNs and LSTMs.

Before diving into what models I developed for each dataset, I want to discuss the core concepts behind SimpleRNNs and LSTMs. SimpleRNNs are a type of RNN that is great for short sequence data like small time series and sentences for sentimental classification. However, a limitation to SimpleRNNs is the fact that it struggles to account remember information form longer sequences, and this is mainly due to the vanishing and exploding gradients. For vanishing gradients, SimpleRNNs can’t remember long sequences because the gradients get smaller as the model backpropagates making the RNN model struggle to learn long-term sequences. On other hand, simpleRNNs may experience exploding gradients because the model is heavily biased towards those short-term sequences making the gradients extremely large and unstable. Some possible tools to reduce exploding and vanishing gradient is using regularization techniques like dropout and gradient clipping. Furthermore, LSTM models are a powerful solution to both the vanishing and exploding gradient problem. This is because of the LSTM’s implementation of the cell state and the gates that monitor what goes into the cell state. Furthermore, the cell state allows the model to retain longer sequences helps LSTM model capture long term dependencies. There are 3 types of gates that monitor what information goes into the cell state: the forget gate, which decides what information of the previous cell state should be removed, and what information should be kept. Then the input gate, which determines what information should be added to the current cell state. Then lastly, the output gate, which determines what information from the current cell state should be added to the current hidden state.

The dataset that I used for the SimpleRNN model was a Sinewave created using the numpy. I created an array where the data ranged from 0 to 25 and contained 2,200 data points in between those values. Next, I created a sin wave based on those values, and created a dataframe where the first columns contained the datapoints that ranged between 0 to 25, and the other column where sin is performed on those data points. I split the data into a training and testing in a 80/20 split, and then scaled it using the min max technique. After that, I created model that consisted of a SimpleRNN layer with 20 neurons followed by a fully connected layer with one output where the sequence length was 70. The optimizer for this RNN model was adam and the loss function that was calculated was mean square error. I fitted the RNN model using a timeseries generator that contained that scaled training data with a batch size of 1 at 100 epochs. The loss stayed very close to zero for most of the epochs. Furthermore, I predicted the next datapoints, and then plotted the results. Unfortunately, the predicted line and actual line were not the same, the actual line increased, decreased and then increased, while the predicted line remained constant between the ranges of .91 and .96. This indicates that the model severely overfitted. Some possible suggestions to improve the model would be adding a regularization technique to prevent overfitting. For example, I could use a dropout layer in between the Simple RNN layer and the dense layer.

For the time series dataset that I retrieved from the internet, I chose was the Metro Interstate Traffic Volume from the UCI Machine learning repository. This dataset showcases the hourly traffic volume of I-94 in Minneapolis, MN based on weather features and holidays from 2012-2018. This is a multivariate time series dataset. Furthermore there are 48,204 rows and 9 columns. 3 continuous columns, traffic volume, clouds all, temperature, 1 date column, date time, and 5 categorical columns, weather main, weather description, snowfall, rainfall, and holiday. Lastly, the date column is a timestamp in a yyyy-mm-dd hh:mm:ss format where the date starts at 2012-10-02 09:00:00 and ends at 2018-09-30 23:00:00. Because this dataset was a large time series dataset, I decided to use an LSTM model. I used column traffic column to predict traffic volume of the next hour indicating that this was LSTM model for univariate time series. The data was split in a 90/10 split, where training had 90% of the data and testing 10% of the data and last 95 data points of the training data included in them. For the architecture of the LSTM model I built, there was 1 input feature, a sequence length of 95, 3 lstm models that consisted 50 neurons each where the activation function was relu, a dropout layer of 20% after the first and second lstm layer followed by a full connected layer with one output. The optimizer used was adam and the loss was mean square error, and 10 epochs when fitting the mode. Lastly, the performance metric was root mean square error. When fitting the LSTM model with the training, it took about an average of 2 minutes per epoch to train. Where the loss hovered around 0.01 majority of the time. Then, I evaluated the model using the testing and achieved an rmse score of 719.27, indicating that there is some strong signs of overfitting. After that I retrained the model using all of the dataset besides the last 95 datapoints, and then forecasted the next 95 datapoints with it. I obtained a rmse score of 1187.31. It took about 2 minutes per epoch to train the model for forecasting. When comparing the full dataset to the testing, it generalized worse than the testing.

To improve the model I decided to add some complex elements along with more regularization. I decided to add a 1d convolution layer to introduce a weight sharing element to the rnn which can help the model learn faster along with a maxpooling layer of 2x2 which can improve feature extraction. I also added these two elements to significantly improve the model’s loss and generalization. I also, removed a lstm layer and then added more neurons to first and second layer. Then, I increased the dropout by 10 percent for each dropout layer that comes after the lstm layer, and then used a dropout of 50% after the maxpooling layer. Lastly I’ll add an extra dense layer of 25 neurons which is then followed by another dropout layer, and then another dense layer with one output. This should not only be able to capture the complex behavior of this time series data, but also increase computational efficiency and generalization.

When fitting the training data with the model, the loss hovered around 0.023 with an average epoch of 35 seconds. Next, I evaluated the new model with the testing data and obtained an rmse score of 973.42. After that, I retrained the new model using all of the dataset besides the last 95 datapoints, and then forecasted the next 95 datapoints with it. It took about 35 seconds per epoch when fitting it. When assessing the predicted 95 data points to the actual datapoints, I obtained a rmse score of 1108.33. The full dataset results ended up generalizing worse than the testing for this model as well.

When comparing the first model to the second model, there were some different outcomes. For example, the loss for the first model was significantly less than second model, the first model generalized better with the testing data than the second model. However, when it came to forecasting, the second model performed better than the first model, even though the second model still had a larger loss per epoch when training the full dataset. Also, the second model trained a minute and 30 seconds per epoch than faster the first model for both fitting the training data and the full dataset.

Overall, what I’ve learned from developing SimpleRNNs and LSTMs was that there tradeoffs depending on the task that you use them. For example, SimpleRNNs are not as computationally expensive and take way less time to train than LSTMs however, it does do well in remembering information from longer sequences. On the other hand, LSTMs do well for tasks that require an output that depends on longer sequences, however, LSTMS do take longer to train. Also, another thing that I learned was that adding a convolution layer to the LSTM can speed up the training significantly.

In conclusion, RNNs and its variants are great for sequence like data like time series and language, I used multiple instances where RNNs can be used effectively with the Sine Wave dataset and the time series dataset for traffic volume. In the future, I want to expand the traffic volume dataset problem by making it a multivariate time series rnn problem where I add columns such as weather main and weather description to see if understanding patterns behind those columns will increase performance of forecasting.