Nayan Arora (u3249907)

Soft Computing Class

Semester 2 2023

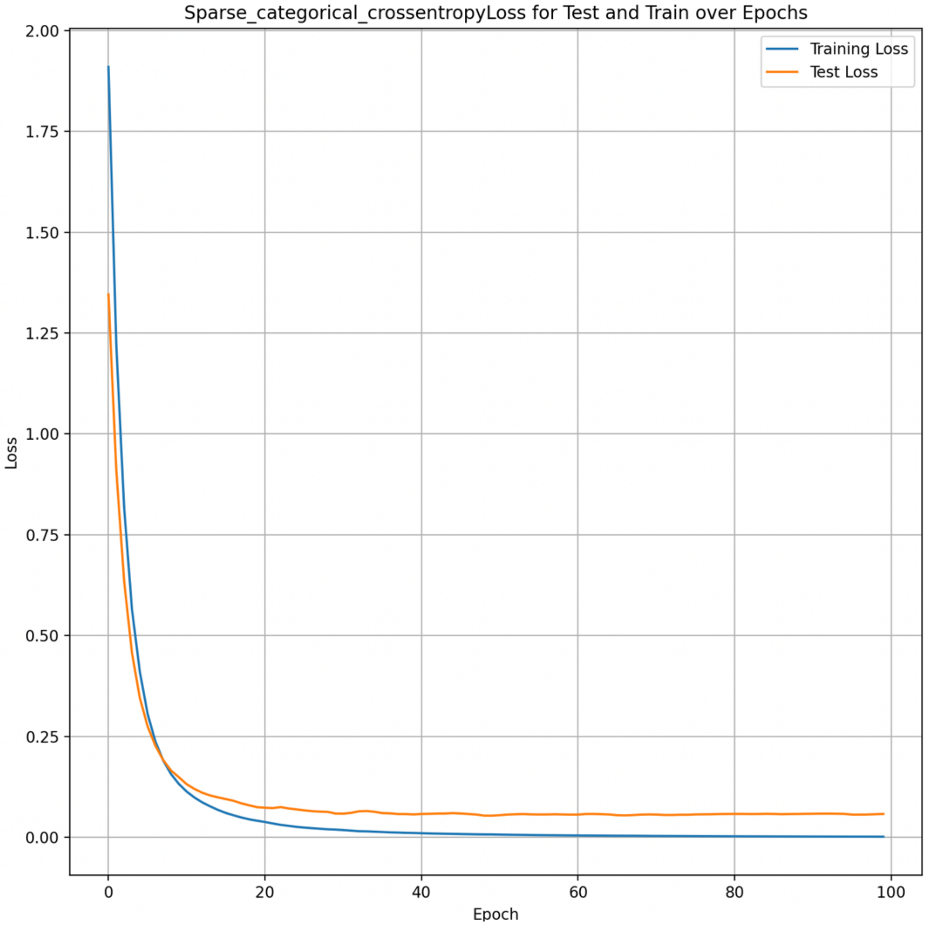
Assignment 1: Neural Networks (PG)

Task1 – Train a basic MLP

Details of MLP	Hidden Nodes: 1 hidden layer with 50 Epochs: 100 Hyperparameters: lr = 0.001 and batch_size = 50 Testing Accuracy: 97.27% Training Accuracy: 100%
Best estimate of test accuracy for a generalised solution.	For this we define a Sequential model with three layers: an input layer with 100 neurons, a hidden layer with 50 neurons, and an output layer with 6 neurons for multiclass classification problem as in this case to achieve an overall accuracy of 97%. Attached below are the screenshots of the test run. After the data pre-processing a keras model is defined as shown below, with activation relu and softmax because this is a multiclass classification problem with a total of 6 possible classes. The dataset given is used to train the mlp over 100 epochs to achieve the attached accuracy. Adam optimizer was used along with sparse_categoricalcrossentropy loss function for this task. The plot for test and training loss function for epoch is attached as well.

```
# define the keras model
model = Sequential()
model.add(Dense(100, input_dim=n_features, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(50, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(n_class, activation='softmax'))
```

```
Epoch 99/100 - Training Loss: 0.0019 - Test Loss: 0.0575 - Training Accuracy: 100.0 - Test Accuracy: 97.273
6/6 [==============================] - 0s 1ms/step - loss: 0.0019 - accuracy: 1.0000
Epoch 100/100 - Training Loss: 0.0019 - Test Loss: 0.0582 - Training Accuracy: 100.0 - Test Accuracy: 97.273
4/4 [==============================] - 0s 907us/step
Final Training Accuracy: 1.0
Final Test Accuracy: 0.9727272987365723
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        40
           1       0.88      0.94      0.91        16
           2       1.00      1.00      1.00        19
           3       0.93      0.87      0.90        15
           4       1.00      1.00      1.00        16
           5       1.00      1.00      1.00         4

    accuracy                           0.97       110
   macro avg       0.97      0.97      0.97       110
weighted avg       0.97      0.97      0.97       110
```



Task2 – Train a basic CNN and MLP

Dataset name	**Diabetes Binary**
Details of CNN	Epochs: 50 Hyperparameters: lr = 0.001 and batch_size = 30 Testing Accuracy: 72.374% Training Accuracy: 72.372%
Details on MLP	Epochs: 50 Hidden Nodes: 1 hidden layer with 40 neurons Hyperparameters: lr = 0.001and batch_size = 20 Testing Accuracy: 72.322% Training Accuracy: 72.67%
Best estimate of test accuracy for a generalised solution for CNN.	72% test accuracy is the generalised solution for this task with the CNN model attached below. It uses Conv1D input layer with input layer has the same number of features as the input features. Followed by a MaxPooling1D layer and a hidden Dense layer with 80 neurons which is fed into the Sigmoid activation function to give the binary result. Since this is a binary classification problem, we use binary cross entropy loss function.
Best estimate of test accuracy for a generalised solution for MLP.	The generalised solution for the MLP is very similar to the CNN in this case. An input layers with 40 neurons and the same number of features as the input was used. Followed by a hidden layer with relu activation and 20 neurons which when fed into the next and final output layer with sigmoid activation produces the binary result. Adam optimizer with a learning rate of 0.001 was used. In this task both the CNN and MLP had similar performance which tells us that it really depends on the type of problem and the kind of data to decide which model to use. CNNs are used for more complex problems and are thus better at learning and capturing trends in the data but in this binary classification problem both had similar performance.

CNN model:

```
# Define the CNN model
model = Sequential()
model.add(Conv1D(input_shape=(n_features, 1), padding = 'same', filters=64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(80, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# compile the keras model
learning_rate = 0.001
optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
```

MLP model:

```
# define the keras model
model = Sequential()
model.add(Dense(40, input_dim=n_features, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(20, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(1, activation='sigmoid'))

# compile the keras model
learning_rate = 0.001
optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
```

Result:

```
Epoch 47/50
2475/2475 - 3s - loss: 0.5423 - accuracy: 0.7255 - 3s/epoch - 1ms/step
Epoch 48/50
2475/2475 - 3s - loss: 0.5425 - accuracy: 0.7252 - 3s/epoch - 1ms/step
Epoch 49/50
2475/2475 - 3s - loss: 0.5422 - accuracy: 0.7258 - 3s/epoch - 1ms/step
Epoch 50/50
2475/2475 - 3s - loss: 0.5423 - accuracy: 0.7267 - 3s/epoch - 1ms/step
Training Accuracy: 72.67
Test Accuracy: 72.322
(base) nayanarora@Nayans-MacBook-Pro softComputing %
```

Second table is just for postgraduate students.

Dataset name	**Music Genre**
Details of CNN	Epochs: 50 Hyperparameters: batch_size = 40, lr = 0.001 Testing Accuracy: 31.587% Training Accuracy: 33.977%
Details on MLP	Epochs: 50 Hidden Nodes: 1 hidden layer with 40 neurons Hyperparameters: batch_size = 40, lr = 0.001 Testing Accuracy: 32.207% Training Accuracy: 33.657%
Best estimate of test accuracy for a generalised solution for CNN.	In this multiclass classification task, multiple different types of models were trained but a higher accuracy was not achieved. This could be because the data has outliers and needs a lot more visualization and pre-processing to remove any and all skews from it. An overall 31% accuracy was achieved by the CNN model attached below. It runs for 50 epochs and has two hidden layers with 100 and 40 neurons respectively. It is fed into the softmax activation function using the adam optimizer and sparse_categorical_crossentropy loss function. The accuracy achieved is generalised accuracy for predicting the music genre.
Best estimate of test accuracy for a generalised solution for MLP.	Similar results were seen in the MLP where an accuracy of 32% was achieved. A sequential MLP with a learning rate of 0.001 was used. 60 neurons for the input layer along with 1 hidden layer with 40 neurons was used to generate and classify into the 10 distinct music classes (n_classes). Working on this task has helped analyze and understand how CNNs and MLPs work and the similarities in their code structure. The major difference is how CNNs are a much more complex in their internal model structure than MLPs. Although both can be used for classification problems, it is upto the user or us to decide which model suits our needs the best. As seen in this task multiple different hyperparamter configurations produced the same bad result. So it can be

	interpreted that the complexity of the CNN does not directly relate to 'better' accuracy.

CNN

```
# Define the CNN model
model = Sequential()
model.add(Conv1D(input_shape=(n_features, 1), filters=64, kernel_size=5, padding = 'same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dense(40, activation='relu'))
model.add(Dense(n_class, activation='softmax'))

# Compile the CNN model
learning_rate = 0.001
optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
```

```
Epoch 50/50
875/875 - 1s - loss: 1.7615 - accuracy: 0.3398 - 990ms/epoch - 1ms/step
Training Accuracy: 33.977
Test Accuracy: 31.587
```

MLP

```
# define the keras model
model = Sequential()
model.add(Dense(60, input_dim=n_features, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(40, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(n_class, activation='softmax'))

# compile the keras model
learning_rate = 0.001
optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
```

```
Epoch 50/50
875/875 - 1s - loss: 1.7772 - accuracy: 0.3366 - 866ms/epoch - 990us/step
Training Accuracy: 33.657
Test Accuracy: 32.207
==============================
469/469 [==============================] - 1s 1ms/step
Overall Accuracy: 32.2067
(base) nayanarora@Nayans-MacBook-Pro softComputing %
```

Task3 – Build a pretrained model for Chest-Xray dataset

The pretrained models used for this task are VGG16 and Inception v3.

The execution took a day and a half to complete on my personal machine. Hence, an alternate less complex AlexNet and Binary AlexNet model was also trained as a faster substitute (screenshots attached).

Note: I am using a mac and cuda is not available to use on my machine, but it still has a usable gpu. A faster implementation of a Pytorch model is implemented on mac using pytorch-metal framework. It is referred and checked for as;

```
if torch.backends.mps.is_available():
    mps_device = torch.device("mps")
```

use device = 'cuda' – if cuda is available.

Pertained name	**VGG16**
Details of model	Epochs: 5 Hyperparameters: learning rate = 0.001, batch_size = 12 Testing Accuracy: 89% Training Accuracy: 91%
Best estimate of test accuracy for a generalised solution for pretrained model.	An overall generalised accuracy of 91% was achieved using the VGG16 pretrained model. Using the adam optimizer and a learning rate of 5 the network was downloaded and implemented using the pytorch library. By studying the VGG16 model in depth, it is understood that is consists of 13 convolutional layers and 3 filly connected layers. Because of the complexity of this model, It is generally used for more complex image classification tasks to avoid overfitting. Value for num_classes was 2 and thus the data fed to the VGG16 model was 1280 input neurons with 2 input features. It is important to note that there are many pretrained models available to suit many types of tasks. The choice of the model depends on the problem. This task helped understand and experience the ideologies of transfer learning first hand.

CNN

```python
class VGG16BNNet(nn.Module):
    def __init__(self):
        super(VGG16BNNet, self).__init__()
        self.model = models.mobilenet_v2(pretrained=True)

        # Freeze model weights
        for param in self.model.parameters():
            param.requires_grad = False

        # Modify the final fully connected layer for the number of classes
        self.model.classifier[1] = nn.Linear(1280, num_classes)

    def forward(self, x):
        return self.model(x)
```

```
mean and std before normalize:
Mean of the image: tensor([0.5137])
Std of the image: tensor([0.2239])
Mean and Std of normalized image:
Mean of the image: tensor([2.1706e-08])
Std of the image: tensor([1.])
Shape of the Image:  torch.Size([12, 3, 224, 224])
Shape of the label:  torch.Size([12])
tensor([1, 1, 1, 1, 1, 1])
['NORMAL', 'PNEUMONIA']
{'NORMAL': 0, 'PNEUMONIA': 1}
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be remove
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/inception_v3_google-0cc3c7bd.pth" to /root/.cache/torch/hub/checkpoints/inception_v3_google-0cc3c7bd.pth
100%|██████████| 104M/104M [00:00<00:00, 212MB/s]
Epoch : 1 VGG16 Train Loss : 0.418007 VGG16 Train Acc : 0.919739
Epoch : 1 VGG16 Valid Loss : 774.064041 VGG!6 Valid Acc : 344.807692
```

Alternate model – alexNET

```python
# Pre-trained AlexNet
class PretrainedAlexNet(nn.Module):
    def __init__(self, num_classes=1000):
        super(PretrainedAlexNet, self).__init__()
        self.model = models.alexnet(pretrained=True)
        # Modify the classifier to match your number of output classes
        self.model.classifier[6] = nn.Linear(4096, num_classes)

    def forward(self, x):
        return self.model(x)
```

Second Table is for PG students.

Pertained name 2	**InceptionV3**
Details of model	Epochs: 5 Hyperparameters: 0.001, batch_size = 12 Testing Accuracy: 92% Training Accuracy: 96%
Best estimate of test accuracy for a generalised solution for pretrained model.	A generalised accuracy of 92% was achieved by the InceptionV3 pretrained model. Similar hyperparameter tunings were used and the weights and layers were frozen to avoid unwanted changes. This model requires even more computational power to run than the VGG16. The inceptionV3 model uses multiple 'inception' layers along with Conv2d and and fully connected dense layers. These mixed inception module layers are more complex cnn layers. Considering it all together we have over 23 different types of layers from input to model output layer. This is one the most complex models but it publicly available for use through the pytorch library. The concept of transfer learning is understood through this task. More variations of the code with easier and less complex algorithms like bert and yolo can also be easily provided if needed.

<u>CNN</u>

```
class InceptionV3Net(nn.Module):
    def __init__(self):
        super(InceptionV3Net, self).__init__()
        self.model = models.squeezenet1_0(pretrained=True)

        # Freeze model weights
        for param in self.model.parameters():
            param.requires_grad = False

        # Modify the final fully connected layer for the number of classes
        self.model.classifier[1] = nn.Conv2d(512, num_classes, kernel_size=1)
        self.model.num_classes = num_classes

    def forward(self, x):
        return self.model(x)
```

Alternate - BinaryAlexNet

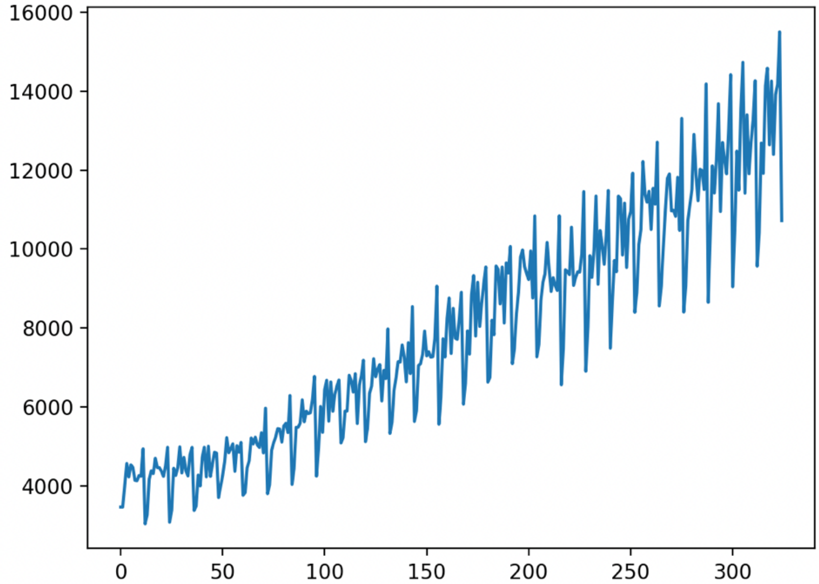
```
# Pre-trained B-AlexNet
class PretrainedBinaryAlexNet(nn.Module):
    def __init__(self, num_classes=2):
        super(PretrainedBinaryAlexNet, self).__init__()
        self.model = models.alexnet(pretrained=True)
        # Modify the classifier to match your number of output classes (2 in this case)
        self.model.classifier[6] = nn.Linear(4096, num_classes)

    def forward(self, x):
        return self.model(x)
```

Task4 – Build a LSTM to time series analysis.

Alcohol Sales

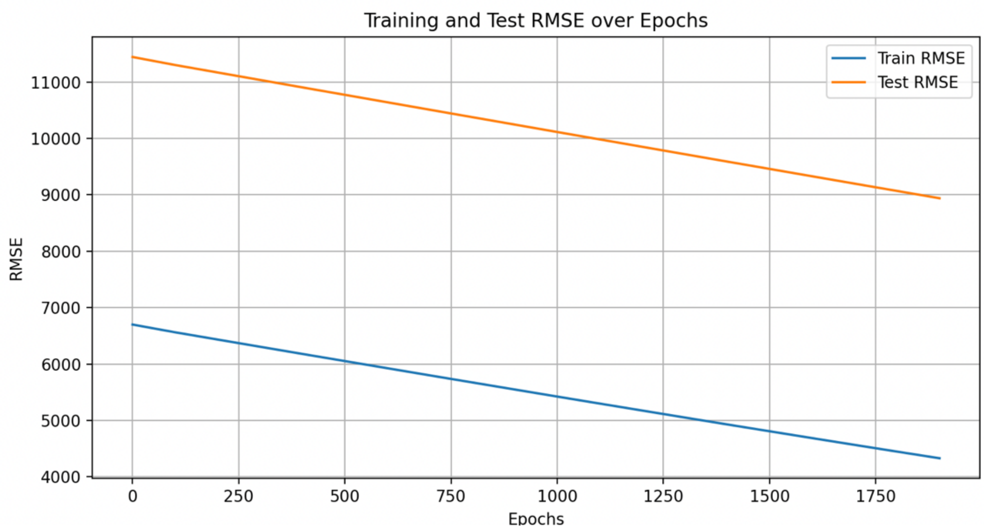
Details of LSTM model	Epochs: 2000 Hyperparameters: lookback = 3, lr = 0.001, batch_size = 10 Testing RMSE: 8939.4170 Training RMSE: 4329.6572
Best estimate of test RMSE for a generalised solution for pretrained model.	Generalised RMSE achieved for test is 8939.4170. It is important to note that the since the data given ranges from 4000 to 16000 the rmse achieved is not too high or 'bad'. It is appropriate and the RMSE loss over epochs is attached below to represent how over 2000 epochs both the test and train rmse gradually decreases which is a good sign. For training this regression model, the lstm model was used with 3 layers and 64 neurons each. A learning rate of 0.001 was used along with the MSELoss function to get the resultant RMSE values for this dataset. Attached below is a screenshot of the model. Since the dataset given was in chronological order, only the sales number column was used to train the model with a 70-30 data split without randomisation.

Given time series data for alcohol sales:




```python
class alcohol_lstm_model(nn.Module):
    def __init__(self):
        super().__init__()
        self.lstm = nn.LSTM(input_size=1, hidden_size=64, num_layers=3, batch_first=True)
        self.linear = nn.Linear(64, 1)
    def forward(self, x):
        x, _ = self.lstm(x)
        x = self.linear(x)
        return x

model = alcohol_lstm_model()
optimizer = optim.Adam(model.parameters(), lr = 0.001)
loss_fn = nn.MSELoss()
#loss_fn = nn.L1Loss()
loader = data.DataLoader(data.TensorDataset(X_train, y_train), shuffle=True, batch_size=10)
```

RMSE values over epochs:



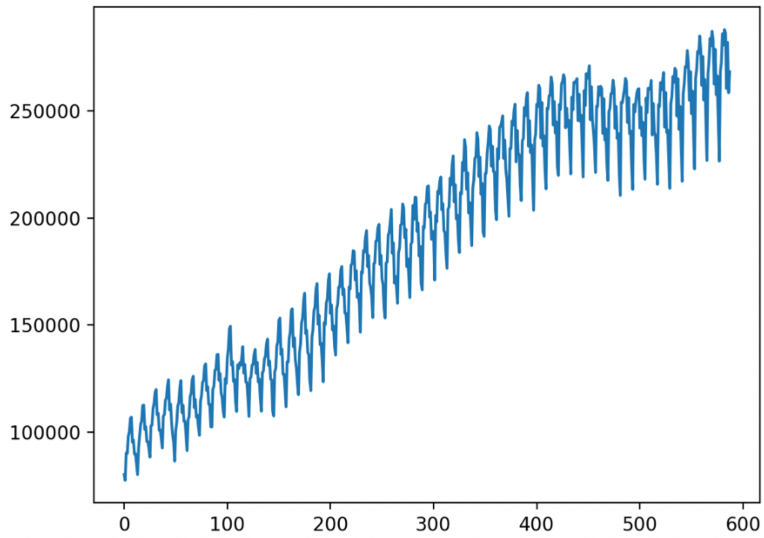
For PG students only:

Miles Travelled

Details of LSTM model	Epochs: 2000 Hyperparameters: lookback = 3, lr = 0.001, batch_size = 15 Testing RMSE: 2467700.75 Training RMSE: 163202.2188
Best estimate of test RMSE for a generalised solution for pretrained model.	For the miles travelled dataset a similar approach was used to build the LSTM model as used in the alcohol sales dataset. Two layers with 100 neurons were used in the linear LSTM model as attached below to achieve a generalised RMSE value of 2467700.75. MSEloss function was used to achieve these values. It was also noted that if instead of MSELoss function we use MAEloss function to calculate the absolute loss the rmse values achieved were around 149.6254. But by observing the data ranges the low values for remse seemed overfitting the data. Hence a generalised solution with low complexity model using a relatively low complexity dataset was achieved as shown in the screenshots below. The train and test RMSE values over 2000 epochs are shown in the attached screenshot below, which shows a gradual decrease over each epoch which basically means less error and better future predictions for this time series model.

Nayan Arora (u3249907)

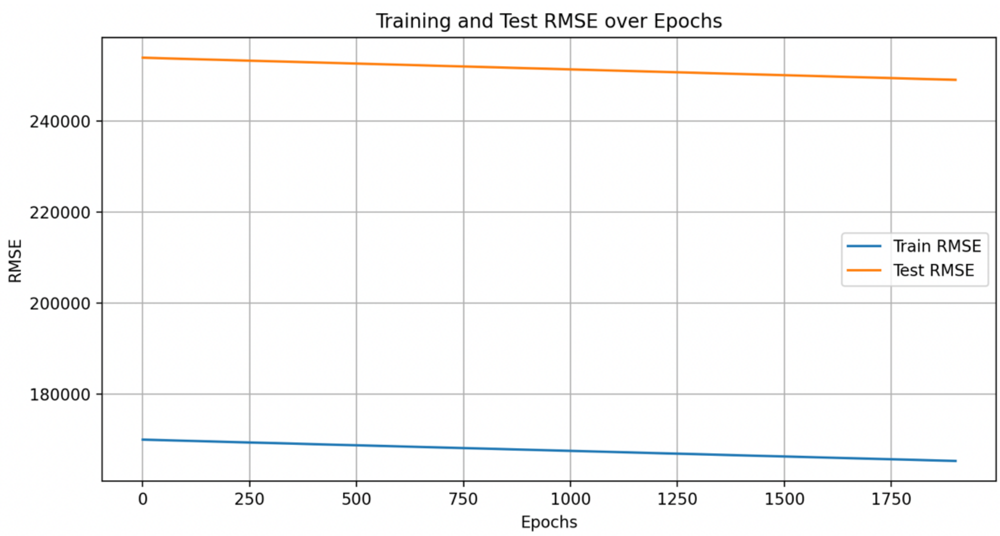
Given time series data for miles travelled:



LSTM model:

```python
class miles_lstm_model(nn.Module):
    def __init__(self):
        super().__init__()
        self.lstm = nn.LSTM(input_size=1, hidden_size=100, num_layers=2, batch_first=True)
        self.linear = nn.Linear(100, 1)
    def forward(self, x):
        x, _ = self.lstm(x)
        x = self.linear(x)
        return x

model = miles_lstm_model()
optimizer = optim.Adam(model.parameters(), lr = 0.001)
loss_fn = nn.MSELoss()
#loss_fn = nn.L1Loss()
loader = data.DataLoader(data.TensorDataset(X_train, y_train), shuffle=True, batch_size=15)
```

RMSE values over epochs:



Example output:

```
shape of the train dataset X and y

torch.Size([408, 3, 1]) torch.Size([408, 3, 1])
shape of the train dataset X and y

torch.Size([174, 3, 1]) torch.Size([174, 3, 1])
Epoch 0: train RMSE 170033.0781, test RMSE 253868.8594
Epoch 100: train RMSE 169666.9375, test RMSE 253489.0469
Epoch 200: train RMSE 169307.0156, test RMSE 253115.5781
Epoch 300: train RMSE 168947.2344, test RMSE 252742.2031
Epoch 400: train RMSE 168587.5156, test RMSE 252368.8438
Epoch 500: train RMSE 168227.8594, test RMSE 251995.4688
Epoch 600: train RMSE 167868.3750, test RMSE 251622.2031
Epoch 700: train RMSE 167508.8750, test RMSE 251248.8438
Epoch 800: train RMSE 167149.5625, test RMSE 250875.6094
Epoch 900: train RMSE 166790.3125, test RMSE 250502.3750
Epoch 1000: train RMSE 166431.1094, test RMSE 250129.1250
Epoch 1100: train RMSE 166072.0000, test RMSE 249755.9062
Epoch 1200: train RMSE 165712.9688, test RMSE 249382.7031
Epoch 1300: train RMSE 165354.0469, test RMSE 249009.5312
Epoch 1400: train RMSE 164995.2344, test RMSE 248636.3906
Epoch 1500: train RMSE 164636.5156, test RMSE 248263.2812
Epoch 1600: train RMSE 164277.8281, test RMSE 247890.1562
Epoch 1700: train RMSE 163919.2500, test RMSE 247517.0469
Epoch 1800: train RMSE 163560.6719, test RMSE 247143.8594
Epoch 1900: train RMSE 163202.2188, test RMSE 246770.7500
```