**Chapter 7**

**Time Series Models**

Introduction:

In Traditional Machine learning use case, data set is typically is a collection of features and target variables. Model uses features to learn and predict the target variable. For example, to predict the house price, here the features could be number of bedrooms, number of baths, square footage and target variable is the price of the house. One thing, if we observe in such a use case is all the records in the dataset are treated equally when predicting target variables and also order of the data doesn’t matter much.

Whereas in time series prediction, order of the data plays important role to capture some the features like trends and seasons. Time series datasets are typically datasets, where time measure is involved. For examples, Climate dataset where temperature is being recorded on hourly basis.

Time series datasets can have one or more features, If there is single feature it’s called as Univariant time series and if there are multiple features it’s called as multi variant time series.

Time series forecasting:

Time series forecasting is form of building a regression model to predict the desired outcome, by using the historical time series data. In simple terms, time series forecasting uses, historical data to train the model and predict for the values.

Stock price forecasting is one of the common use case for the time series forecasting.

Why Deep learning:

Deep learning in time series forecasting overcomes the disadvantages of tradition machine learning, such as:

* Identifying complex patterns, trends and seasons.
* Forecasting for long terms.
* Handling missing values.

Also, There are various options and techniques available in deep learning to perform time series forecasting like:

* Recurrent Neural Network (RNN)
* Long Short Term Memory (LSTM)
* Gated Recurrent Unit (GRU)
* Encoder Decoder model

Section -2 Chapter -7 Heading -2

Getting started with Time Series models

Before we build our pytorch lightening time series forecasting models, it is important to understand some of the important techniques are used.

Windowing/Sequences:

In time series forecasting, it is important to give the model the complete information possible at a given time and the past data in time series plays an important role to make the future predictions. This is the technique, where we reshape our time series dataset into fixed windows, that will give model the most complete information to achieve accurate predictions.

Let’s try to understand windowing with below examples.

A picture containing text, receipt

Description automatically generated

Let us consider univariant timeseries dataset with nine rows as shown in the above diagram, which has data column represented in format <mm/dd/YY> and with a column called target. Let’s apply windowing on the above datset with window size being 3.

Table

Description automatically generated

After applying windowing with window size as 3, we do see there are six rows of data being generated and the features for our window 1 will be the collection of all the features of 1/1/20, 1/2/20 and 1/3/20 and the target variable will the target value of 1/4/20.

For example if we are predicting temperature for a day by using humidity recorded, then for the window 1, features will be the collection of humidity recored for the days 1/1/20, 1/2/20 and 1/3/20 and the the target variable will the temperature value of 1/4/20.

Windowing/Sequencing is one of the most commonly used technique in time series forecasting, which helps us to provide complete information to our model to predict better forecasting.

In the next two sections of the chapter we shall go through the real-world examples for time series forecasting, explained in detail. Which typically follows below steps:

1. Load and apply feature engineering on Dataset
2. Build Model
3. Train Model
4. Perform Time series forecasting

At step 1, is where we perform process the data and this is the step where windowing technique is being applied.

Now, since so far we have understood about what and how time series forecasting works. In the next two chapters we shall use RNN and LSTM model to perform forecasting using pytorch lightening.

Section -2 Chapter -7- Heading -3

Time Series using RNN Model

Forecasting Daily Climate in Delhi

Sequence models in deep learning are one of the most commonly and widely used techniques to solve time series use cases. In this chapter we will solve one of the time series use case using RNN model using pytorch ligthening.

In this chapter we’re going to cover the following main topics:

* About the dataset and use case
* Loading Data
* Feature Engineering
* Creating Custom Dataset
* RNN model using Pytorch Lightening
* Train the model
* Measuring Training Loss
* Load the model
* Prediction on test dataset

Technical requirements (H1 – Section)

In this chapter we will be primarily using below python modules mentioned with their versions:

pytorch-lightning (version: 1.1.2)

seaborn (version: 0.11.0)

sklearn (version: 0.22.0)

numpy (version: 1.19.4)

torch (version: 1.7.0)

pandas (version: 1.1.5)

matplotlib (version: 3.2.2)

Working examples for this chapter can we found at this github link (paste here)

About the dataset and the use case

In this chapter we will be using publicly available dataset to build our first RNN model using pytorch lightening to perform forecasting, we shall also discuss all the steps and process involved in detail.

Dataset:

Dataset we are going to use in this chapter is called “Daily Climate time series data”, which has daily climate data recording for one of the city called Delhi in India, stating form 2013 to 2017. This dataset is available at kaggle.com and can be downloaded at this URL <https://www.kaggle.com/sumanthvrao/daily-climate-time-series-data?select=DailyDelhiClimateTrain.csv>.

This dataset has two different files, like traditional machine learning approaches we shall use one file for training our RNN model and the other file to perform forecasting:

*DailyDelhiClimateTrain.csv –* This file has five columns and 1462 rows of data, starting for 2013-01-01 to 2017-01-01. We will be using this file for training for RNN model.

Here is the screenshot of top 5 rows of our train dataset.

Table

Description automatically generated

*DailyDelhiClimateTest.csv –* This file has same five columns as above file, but with less number of records starting from 2017-01-01 to 2017-04-24. We shall be using this file to forecast and compare results with actual data and the forecasting results.

Here is the screenshot of top 5 rows of our test dataset.

Table

Description automatically generated

So far, we have talked about what is dataset and the files. Now, let’s try to understand what this dataset represents and about the columns in the above files correspond to.

Columns:

*date:* This is the date column represented in format <YYYY-MM-DD>, which represents each day, where temperature recording has been recorded.

*meantemp:* This column is of type float, representing mean temperature averaged out from multiple of 3-hour interval for the entire day.

*humidity:* This column is of type float, representing humidity value of the day. Units are grams of water vapor per cubic meter volume of air.

*windspeed:* This column is of type float, representing wind speed measured in kilometer per hour.

*meanpressure:* This column is of type float, representing mean pressure recorded for the entire day and in units of atm.

Use case:

Our goal here is to make use above dataset, build RNN model using pytorch lightening and predict mean temperature. That is we will be using past values of *humidity, wind\_speed, mean pressure and meantemp*  as our inputs for our model and predict future *meantemp* value.

Loading Data

We shall use python panda’s module to load our csv files, All the files mentioned above are saved under a directory called *sample\_data.* Here we shall use pandas *read\_csv*  method to load the data into pandas data frame.

Load Train dataset:

Let’s read our training dataset (*DailyDelhiClimateTrain.csv*) into pandas data frame object called *climate\_train* as shown in the below code block.

1. climate\_train = pd.read\_csv("./sample\_data/DailyDelhiClimateTrain.csv")
2. climate\_train.head(5)

 Now, quickly check total number of rows and columns available inside the *climate\_train* data frame.

1. print("Total number of row in train dataset:", climate\_train.shape[0])
2. print("Total number of columns in train dataset:", climate\_train.shape[1])

output:

Total number of row in train dataset: 1462

Total number of columns in train dataset: 5

Load Test dataset:

Let’s read our test dataset (*DailyDelhiClimateTest.csv*) into panda’s data frame object called *climate\_test* as shown in the below code block.

1. climate\_test = pd.read\_csv("./sample\_data/DailyDelhiClimateTest.csv")
2. climate\_test.head(5)

Now, quickly check total number of rows and columns available inside the *climate\_test* data frame.

1. print("Total number of row in test dataset:", climate\_test.shape[0])
2. print("Total number of columns in test dataset:", climate\_test.shape[1])

output:

Total number of row in test dataset: 114

Total number of columns in test dataset: 5

In this section, we have successfully loaded our both the files located under sub directory *sample\_data* into panda’s data frame and also made sure have correct number of rows and columns in the files. In the upcoming section we shall perform some operations on our data frames.

Feature Engineering

It's always good to perform normalization on our data set to produce good results and also it helps our model run faster.

Here we shall perform scaling on our data frame columns, that is we shall use robust scaler from sklearn preprocessing module and apply scaling on the both features and target columns.

In the below code block we are primarily performing these steps:

Create scalers:

* Create five different scaler objects for our both features and target columns.

Create transformers:

* Creating transformer object for all our features and target columns, by calling fit method on our *climate\_train* data frame. At this step our transformer for all the five columns for both features and target columns are ready.

Apply Scalers

* Finally, applying scaling on both test and train data frame and overwriting the columns values with new scaled values. After, applying this step we have our both test and train data frame with scaled values.

1. #create scalers
3. #target column
4. mean\_temp\_scaler = RobustScaler()
5. #features column
6. humidity\_scaler = RobustScaler()
7. wind\_speed\_scaler = RobustScaler()
8. mean\_pressure\_scaler = RobustScaler()
10. #Create transformers
11. mean\_temp\_transformer = mean\_temp\_scaler.fit(climate\_train[['meantemp']])
12. humidity\_scaler\_transformer = humidity\_scaler.fit(climate\_train[['humidity']])
13. wind\_speed\_scaler\_transformer = wind\_speed\_scaler.fit(climate\_train[['wind\_speed']])
14. mean\_pressure\_scaler\_transformer = mean\_pressure\_scaler.fit(climate\_train[['meanpressure']])
16. #Apply Scalers
18. #apply scaling on train dataset
19. climate\_train["meantemp"] = mean\_temp\_transformer.transform(climate\_train[['meantemp']])
20. climate\_train["humidity"] = mean\_temp\_transformer.transform(climate\_train[['humidity']])
21. climate\_train["wind\_speed"] = mean\_temp\_transformer.transform(climate\_train[['wind\_speed']])
22. climate\_train["meanpressure"] = mean\_temp\_transformer.transform(climate\_train[['meanpressure']])
23. #apply scalling on test dataset
24. climate\_test["meantemp"] = mean\_temp\_transformer.transform(climate\_test[['meantemp']])
25. climate\_test["humidity"] = mean\_temp\_transformer.transform(climate\_test[['humidity']])
26. climate\_test["wind\_speed"] = mean\_temp\_transformer.transform(climate\_test[['wind\_speed']])
27. climate\_test["meanpressure"] = mean\_temp\_transformer.transform(climate\_test[['meanpressure']])

Our both test and the train data frames are now ready with scaled values in it, entire data in our data frames are in sorted order by default and we may not need date column. Let’s delete date column from both of data frames by running below code block.

1. del climate\_train["date"]
2. del climate\_test["date"]

 Before we get into the next section of this chapter, let’s print top 5 rows of both train and test data frames, to make sure values are in scaled format.

1. climate\_train.head(5)

 output:

Table

Description automatically generated

1. climate\_test.head(5)

Table

Description automatically generated

Creating Custom Dataset

It’s always a good practice to access data via our own custom dataset, In this section we shall walkthrough the process of creating custom dataset using pytorch Dataset module, this module is available in pytorch utils data (torch.utils.data.Dataset

).

Let’s call our dataset as *BikeSharingDataset*, which is being inherited from torch.utils.data.Dataset class.

Here is the detail explanation of methods and the process involved in *BikeSharingDataset* in detail. Also, complete code is available at the end of this section and also available in github.

\_\_init\_\_():

**params:**

*train-* Takes a boolen as input, if true train dataset is being processed and returned

*validate-* Takes a boolen as input, if true validate dataset is being processed and returned

*test-* Takes a boolen as input, if true test dataset is being processed and returned

*window\_size-* Size of the window or sequence to be fed into our LSTM model. Default value for our window size is 240, here 240 window size represents 10 days of data. That is for each day we have 24 recoding and for 10 days we have 240 records of data.

\_\_init\_\_ is the method where we are loading our dataset, process and making it ready to be consumed. \_\_init\_\_ method is divided into four steps, let’s go through each step-in detail.

STEP1: Load the data inside class

1. #STEP1: Load the data
2. self.climate\_test = climate\_test
3. self.climate\_train = climate\_train

At this step, we are initializing our *climate\_test* and *climate\_train* data frames within our *ClimateDataset* class, so that we can use this data frames inside our class methods.

STEP2: Creating Features

1. #STEP2: Creating Features
2. if train: #process train dataset
3. features = self.climate\_train
4. target = self.climate\_train.meantemp
5. else: #process test dataset
6. features = self.climate\_test
7. target = self.climate\_test.meantemp

One of the two parameters of our *ClimateDataset* is train and test, both takes Boolean values as input. In this code block we are creating two variables, one is the features which consists of dataframe with our features column that is data frame with columns *humidity, wind\_speed, meanpressure and meantemp.* And second is the data frame with single column *meantemp.*

If the train parameter is *True,* features and target variable are setup with *climate\_train* dataset and if test parameter is True, features and target variable are loaded from *climate\_test* data frame.

STEP3: Create windows/sequencing

1. #STEP3: Create windows/sequencing
2. self.x, self.y = [], []
3. for i in range(len(features) - window\_size):
4. v = features.iloc[i:(i + window\_size)].values
5. self.x.append(v)
6. self.y.append(target.iloc[i + window\_size])

Since we are dealing with timeseries use case it is important for us perform windowing/sequencing on our dataset for our model to perform better. At this step, we are converting our dataset into windows of default size of 7, that is we are training RNN with the seven days of historical data to make prediction of the next day. This way in general sequential models perform well in time series predictions.

In the above code python class variable *x,* will be having sequence of past 7 days of features (*humidity, wind\_speed, meanpressure and meantemp*) and class variable y will be having the target variable(*meantemp)* for the next consecutive day.

STEP4: Calculate length of dataset

Here we are calculating total number of records for our data after performing windowing, which will later be used for the \_\_len\_\_(self) method.

To summarize our \_\_init\_\_() method, which takes boolean parameter as input and depending on the flag that is being set, It either picks up train or test dataset. Then features and the target is being extracted from the dataset, performs windowing on the features and target data with default window size as 7. After windowing, this sequence data is being stored in class variables called *x* and *y*.

Complete code block for \_\_init\_\_() method:

1. def \_\_init\_\_(self, train=False, test=False, window\_size=7):
3. #STEP1: Load the data inside class
4. self.climate\_test = climate\_test
5. self.climate\_train = climate\_train
7. #STEP2: Creating Features
8. if train: #process train dataset
9. features = self.climate\_train
10. target = self.climate\_train.meantemp
11. else: #process test dataset
12. features = self.climate\_test
13. target = self.climate\_test.meantemp
15. #STEP3: Create windows/sequencing
16. self.x, self.y = [], []
17. for i in range(len(features) - window\_size):
18. v = features.iloc[i:(i + window\_size)].values
19. self.x.append(v)
20. self.y.append(target.iloc[i + window\_size])
22. #STEP4: Calculate length of dataset
23. self.num\_sample = len(self.x)

\_\_len\_\_(self):

1. def \_\_len\_\_(self):
2. #returns the total number of records for climate data set
3. return self.num\_sample

This function returns the class variable *num\_sample,* which was initialized in \_\_init\_\_() method which has total count of records in our dataset.

\_\_getitem\_\_(self, index):

1. def \_\_getitem\_\_(self, index):
2. x = self.x[index].astype(np.float32)
3. y = self.y[index].astype(np.float32)
4. return x, y

In our \_\_init\_\_() method after performing windowing on our dataset, All our dataset set is being stored in two class variables x and y. This method takes index as a parameter and returns the record for both our features and target variable, that is returns value at index from both class variable x and y.

Here is the complete code snippet for the *ClimateDataset* class:

1. class ClimateDataset(torch.utils.data.Dataset):
2. def \_\_init\_\_(self, train=False, test=False, window\_size=7):
4. #STEP1: Load the data inside class
5. self.climate\_test = climate\_test
6. self.climate\_train = climate\_train
8. #STEP2: Creating Features
9. if train: #process train dataset
10. features = self.climate\_train
11. target = self.climate\_train.meantemp
12. else: #process test dataset
13. features = self.climate\_test
14. target = self.climate\_test.meantemp
16. #STEP3: Create windows/sequencing
17. self.x, self.y = [], []
18. for i in range(len(features) - window\_size):
19. v = features.iloc[i:(i + window\_size)].values
20. self.x.append(v)
21. self.y.append(target.iloc[i + window\_size])
23. #STEP4: Calculate length of dataset
24. self.num\_sample = len(self.x)
26. def \_\_getitem\_\_(self, index):
27. x = self.x[index].astype(np.float32)
28. y = self.y[index].astype(np.float32)
29. return x, y
31. def \_\_len\_\_(self):
32. #returns the total number of records for climate data set
33. return self.num\_sample

Before we end this section, let’s quickly test our *ClimateDataset,* one easy way to test is to use for loop and after single iteration stop the loop by calling break statement. In the below code, we are creating our *ClimateDataset* for the train data, looping it over single iteration and print size of the features, content of features and targets.

1. climate = ClimateDataset(train=True)
3. #let's loop it over single iteration and print the shape and also data
4. for i, (features,targets) in enumerate(climate):
5. print("Size of the features",features.shape)
6. print("Printing features:\n", features)
7. print("Printing targets:\n", targets)
8. break

output:

Text

Description automatically generated

Our features size muti dimensional with seven rows and four columns. We have seven rows of historical data because our default window size is seven and four column because we have four unique features. Finally, there is a single future target value.

RNN model using Pytorch Lightening

So far, we have loaded the dataset into pandas, performed feature engineering, created custom dataset to access data. Now, we are ready with our data which is processed and converted into sequence/windowing which is ready to be fed into our model. Let’s start writing our single layer RNN model.

To build our RNN model using pytorch Lightening , let’s divide the process into below steps:

1. Create Model
2. Setup Optimizer
3. Setup Data
4. Configure Traning Loop
5. Create Model

Let’s create a class called as RNN which is inherited from Lightening Module class from pytorch lightening.

Takes the following input or parameters in the constructor:

**input\_size**: Number of features that are being used, in our case total number of features our climate dataset is 4 and also default value for this param is 4.

**hidden\_dim**: This is total number of hidden RNN's, that is replicas of RNN needed. By default this value is being set to 10, that is we have 10 hidden layers.

**n\_layers**: This param is used to stack up the RNN's over other, for this chapter we shall use single layer neural network, but can be easily changed with this parameter.

**output\_size**: Number of output's expected from the model, since we are predicting mean temparature which is a regression problem therefore output size is 1.

\_\_init\_\_():

1. def \_\_init\_\_(self, input\_size=4, output\_size=1, hidden\_dim=10, n\_layers=1, window\_size=7):
2. """
3. input\_size: Number of features in the input
4. hidden\_dim: number of hidden layers
5. n\_layers: number of RNN to stack over each other
6. output\_size: number of items to be outputted
7. """
8. super(RNN, self).\_\_init\_\_()
10. self.hidden\_dim = hidden\_dim
11. self.n\_layers = n\_layers
13. self.rnn = nn.RNN(input\_size, hidden\_dim, n\_layers, batch\_first=True)
14. self.fc = nn.Linear(hidden\_dim \* window\_size, output\_size)
16. self.loss = nn.MSELoss()

In our \_\_init\_\_ method we are primarily doing three things:

* Initialize RNN model from torch nn module, here we are creating our RNN model with the input size, hiddern dimensions and number of layers that are being passed during initializing the model. Also, setting up batch\_first parameter as True since here RNN layer is our first layer.
* Creating Linear layer again grom torch nn module, where input size total number of dimensions times the window size and second parameter is the size of the output. In our case, we are predicting meantemp, output size is 1.
* Initializing loss function, Here we are using MSE loss function to calculate losses.

get\_hidden():

1. def get\_hidden(self, batch\_size):
2. hidden = torch.zeros(self.n\_layers, batch\_size, self.hidden\_dim)
3. return hidden

 This method takes batch size as input and returns three dimensional tensor initialized with zero’s on number of layers, batch size and the hidden dimensions of our RNN model.

Forward():

1. def forward(self, x):
2. batch\_size = x.size(0)
3. hidden = self.get\_hidden(batch\_size)
4. out, hidden = self.rnn(x, hidden)
5. out =out.reshape(out.shape[0], -1)
6. out = self.fc(out)
7. return out

In forward method, we are connecting all the layers by configuring inputs and outputs.

* First, we are picking up the batch size with the help of the size method.
* Initializing the hidden layer, by calling our utility function called as *get\_hidden* which returns us the multi-dimensional tensor filled with zero’s.
* Once the initial hidden layer is ready, input *x* along with hidden layer is being sent to our RNN model, which returns us output and the hidden layer.
* Before passing the output to our fully connected layer, output data is being flattened to single dimension and then sent as an input to the fully connected network.
* Finally, output from the fully connected network is being returned from the forward method.

1. Setup Optimizer

When we are using pytorch lightening to build our model, it is needed to set up our optimizer inside our model class, this can be done by overwriting method called *configure\_optimizers()*

1. def configure\_optimizers(self):
2. params = self.parameters()
3. # optimizer = optim.Adam(params=params, lr = 0.01)
4. optimizer = optim.RMSprop(params=params, lr = 0.001)
5. return optimizer

 In this method we are primarily doing two operations:

* Get the model parameters, since this method is written inside our model class parameters for the model can be accessed calling method *self.parameters()*.
* Parameters for the model, once collected are passed to RMSprop optimizer, which can be accessed from the torch optim module. Along with model parameters, in above code we have setup learning rate as 0.001
* Once the parameters and the learning rate is setup, RMSprop optimizer is returned.

Important Note

For this use case we have RMSprop as our optimizers and with learning rate as 0.001. This can always be tried with different optimizers and try it out with different learning rates for the better performance of the model.

1. Setup Data

To train our RNN model on our train dataset, let’s setup our train data loader. One of doing this is my overwriting method from pytorch lightening Lightning Module called *train\_dataloader.*

1. def train\_dataloader(self):
2. climate\_train = ClimateDataset(train=True)
3. train\_dataloader = torch.utils.data.DataLoader(climate\_train, batch\_size=20)
4. return train\_dataloader

In this method, we are creating a train Dataset from the ClimateDataset class which we have created in the initial sections of this chapter. Here, we are making use of pytorch DataLoader class to create the our train data loader.

Data Loader is being created from out *ClimateDataset* and with batch size of 20.

Important Note

- When creating Dataset from our custom dataset class which is *ClimateDataset,* train flag is set as *True* which means our dataset reurns train dataset.

- In the code we have fixed the batch size as 20, depending on your hardware requirement and the performance this batch size can be increased or decreased as per the need.

1. Configure Traning Loop
2. def training\_step(self, batch, batch\_idx):
3. features, targets = batch
4. output = self(features)
5. output = output.view(-1)
6. loss = self.loss(output, targets)
7. return {"loss": loss, 'log': {'train\_loss': loss}}

This is one of the important method, this is where input batches of data is being accesed and model training is performed.

Method training\_step takes two params

*batch\_idx*: Index of the batch

*batch*: This is data batch from the data loader retuned from the method *train\_dataloader.*

Below are the steps we are performing inside training\_step method:

* The batch parameter is of type tupple, which has both features and targets.
* We are getting the output of the model, by passing features to the *self* as you can see in the above code.
* When the output is being received from the model, same output is being converted into single dimensional array with the help of *view* pytorch method. Then, loss is being calculated.
* The *training\_step* method is expected to return loss function, we are returning loss function within dictionary.

Important Note

- The *training\_step* method returns dictionary with two keys, one is the loss and the second key is the log. Important thing here is anything we pass inside the log will be logged by the pytorch lightening, at this step we are logging our loss as train\_loss. We shall soon see how can we access and plot the train\_loss

Train the model

1. seed\_everything(10)
2. trainer = pl.Trainer(max\_epochs=30, progress\_bar\_refresh\_rate=20)
3. model = RNN()
4. trainer.fit(model)

  output:

Graphical user interface, text, application, email

Description automatically generated

At this step, our RNN model is build and ready to be trained. In the above code we have created our Trainer object from pytorch lightening with maximum epochs as 30 and refreshing progress bar for every 20 units. Once our pytorch lightening Trainer object is ready, in next steps we have created our RNN model object and started training.

Important Note

- Here we have applied *seed\_everything(10)* so that we can generate same results everything, easy to debug. But, this is optional and can be removed.

- Number of epochs are chosen to be 30, that is we are training our data for 30 iterations. This can we increased or decreased depending on the model performance.

Measuring Training Loss

In the above steps at training step, we have logged train\_loss. Now let’s use tensorboard to plot our training loss and monitor how the train loss though out each epoch. Let’s start tensor board by running below code:

1. %load\_ext tensorboard
2. %tensorboard --logdir lightning\_logs/

Chart, line chart

Description automatically generated

Here we can clearly see train loss is been going down for every iteration, it may further go down if we increase our epochs size, but for this use case we shall stick with epoch size of 30.

Load the model

Before we predict on the test data and compare results, which we are going to do in next section of this chapter. We need to first load the model, this can be done from the below steps:

Listing the files at pytorch lightening default path (*lightening\_logs*)

Graphical user interface, text, application, email

Description automatically generated

Once, we get to know the model file name, we can make use of below code which uses *load\_from\_checkpoint* method to load the model and change the mode to *eval.* Now the model is being loaded from file and ready to perform predictions.

1. #load the model
2. PATH = 'lightning\_logs/version\_0/checkpoints/epoch=29-step=2189.ckpt'
3. trained\_climate\_RNN = model.load\_from\_checkpoint(PATH)
4. trained\_climate\_RNN.eval()

 output:

Text, letter

Description automatically generated

Prediction on test dataset

This is the final step in this section, where we use model to make prediction on the test data set we created and plot the chart actual values Vs predicted values.

Steps involved in below code to perform prediction on test data set are:

* First create ClimateDataset object by passing *test* parameter as *True* and use dataset to create a loader with batch size of 20.
* Iterating over the data loader, collecting the features and these features are being used to make predictions over our the trained model.
* All the predicted and the actual target values are being stored in local variable called *predicted\_result* and *actual\_result.*

1. #Initialize Dataset
2. climate\_test = ClimateDataset(test=True)
3. train\_dataloader = torch.utils.data.DataLoader(climate\_test, batch\_size=20)
4. predicted\_result, actual\_result = [], []
5. #let's loop it over single iteration and print the shape and also data
6. for i, (features,targets) in enumerate(train\_dataloader):
7. result = trained\_climate\_RNN(features)
8. predicted\_result.extend(result.view(-1).tolist())
9. actual\_result.extend(targets.view(-1).tolist())

 Before we plot the chart, let’s use our *mean\_temp\_transformer* which was created in feature engineering phase to perform inverse transform on to the dataset.

1. actual\_predicted\_df = pd.DataFrame(data={"actual":actual\_result, "predicted": predicted\_result})
2. inverse\_transformed\_values = mean\_temp\_transformer.inverse\_transform(actual\_predicted\_df)
3. actual\_predicted\_df["actual"] = inverse\_transformed\_values[:,[0]]
4. actual\_predicted\_df["predicted"] = inverse\_transformed\_values[:,[1]]
5. actual\_predicted\_df

 output:

Table

Description automatically generated

Now let’s plot the line chart for actual Vs predicted values.

1. plt.plot(actual\_result,'b')
2. plt.plot(predicted\_result,'r')
3. plt.show()

Here we are plotting the values for *meantemp* actual values Vs predicted values

The blue line represent the actual mean temperature values and the red line represent predicted values from our RNN model.

Chart, line chart

Description automatically generated

Section -2 Chapter -7 Heading -4

Time Series using LSTM

Predicting London Bike Sharing Sales

In this chapter we shall build our LSTM model using pytorch lightening to predict London bike sharing sales.

In this chapter we’re going to cover the following main topics:

* About the dataset and use case
* Load and Split dataset
* Feature Engineering
* Creating Custom Dataset
* LSTM model using Pytorch Lightening
* Find Learning rate and Train the model
* Measuring Training Loss
* Load the model
* Prediction on test dataset

Technical requirements (H1 – Section)

In this chapter we will be primarily using below python modules mentioned with their versions:

pytorch-lightning (version: 1.1.2)

seaborn (version: 0.11.0)

sklearn (version: 0.22.0)

numpy (version: 1.19.4)

torch (version: 1.7.0)

pandas (version: 1.1.5)

matplotlib (version: 3.2.2)

Working examples for this chapter can we found at this github link (paste here)

About the dataset and the use case

In this chapter we will be using publicly available dataset to build our first RNN model using pytorch lightening to perform forecasting, we shall also discuss all the steps and process involved in detail.

Dataset:

Dataset we are going to use in this chapter is called “London bike sharing dataset”, which has Historical data for bike sharing in London- Powered by TfL Open Data, It has hourly recording of bike sharing sales happened stating form 2015-01-04 to 2017-01-03. This dataset is available at kaggle.com and can be downloaded at this URL <https://www.kaggle.com/hmavrodiev/london-bike-sharing-dataset> .

File name: *London\_merged.csv*

*London\_merged.csv –* This file has ten columns and 17414 rows of data, starting for 2015-01-04 to 2017-01-03.

Here is the screenshot of top 5 rows of our train dataset.

Table

Description automatically generated

Columns:

*timestamp:* Timestamp recording for the bike share occurred, in format <YYYY-MM-DD hh:mm:ss>

*cnt:* Total number of bike sharing.

*t1:* Actual temperature recorded at the time of bike sharing.

*t2:* Feel’s like temperature recorded at the time of bike sharing.

*hum:* Humidity in atmosphere in the percentage form.

*windspeed:* Wind speed recorded recorded at the time of bike sharing.

*weathercode:* Integer variable representing weather code, there are around 7 unique weather codes available in the data.

*isHoliday:* Column representing, if the given day is a holiday or not. 1- represents holiday and 0- represents as not a holiday.

*season:*This column represents categories of seasons. 0- represents spring, 1- represents fall and 2- represents winter.

*weather\_code:* This column represents how the weather condition

1 - Clear ; mostly clear but have some values with haze/fog/patches of fog/ fog in vicinity

2 - scattered clouds / few clouds

3 - Broken clouds

4 - Cloudy

7 - Rain/ light Rain shower/ Light rain

10 - rain with thunderstorm

26 - snowfall

94 - Freezing Fog

Use case:

Our goal here is to make use above dataset, build multi-layer stacked LSTM model and forecast the bike sharing sales. Also, in this use case we shall focus on some of the pytorch lightening techniques for handling validation dataset inside the model and also use one of the pytorch lightening automated technique to identify learning rate value.

Load and split dataset

We are using python panda’s module to load our csv files, All the files mentioned above are saved under a directory called *sample\_data.* Here we shall use pandas *read\_csv*  method to load the data into pandas data frame.

Load dataset

We know that our dataset has column name *timestamp,* which is of type timestamp. During the process of reading the dataset, let’s set the index column as timestamp and also pass the same column name in the *parse\_dates* parameter as shown in the below code block.

bike\_sharing\_df = pd.read\_csv("./sample\_data/london\_merged.csv", parse\_dates=['timestamp'], index\_col="timestamp")

bike\_sharing\_df.head(5)

output:

Table

Description automatically generated

Total number of rows and columns in our dataset are:

print("Total number of row in dataset:", bike\_sharing\_df.shape[0])

print("Total number of columns in dataset:", bike\_sharing\_df.shape[1])

output:

Text

Description automatically generated

Split dataset

Let’s spit the entire dataset into three different parts, that is train, validate and test dataset.

Train dataset

For the train dataset let’s consider data from range *2015-01-04* to *2016-07-31,* Below code snippet divides the dataset within our required date range. This train dataset is used to train our model.

bike\_sharing\_train = bike\_sharing\_df.loc[:datetime.datetime(year=2016,month=7,day=31,hour=23)]

print("Total number of row in train dataset:", bike\_sharing\_train.shape[0])

print("Train dataset start date :",bike\_sharing\_train.index.min())

print("Train dataset end date:",bike\_sharing\_train.index.max())

output:

Text

Description automatically generated with medium confidence

Validate dataset

For the validate dataset let’s consider data from range from the point where train dataset ends, that is from *2016-08-01* to *2016-10-31,* Below code snippet divides the dataset within our required date range. This validate dataset is used as validation dataset during the model training process.

bike\_sharing\_val = bike\_sharing\_df.loc[datetime.datetime(year=2016,month=8,day=1,hour=0):datetime.datetime(year=2016,month=10,day=31,hour=23)]

print("Total number of row in validate dataset:", bike\_sharing\_val.shape[0])

print("Validate dataset start date :",bike\_sharing\_val.index.min())

print("Validate dataset end date:",bike\_sharing\_val.index.max())

output:

Text

Description automatically generated

Test dataset

For the test dataset let’s consider data from range from the point where train dataset ends, that is from *2016-11-01* to *2017-01-03,* Below code snippet divides the dataset within our required date range. This test dataset is used as forecast our results after model being trained.

bike\_sharing\_test = bike\_sharing\_df.loc[datetime.datetime(year=2016,month=11,day=1,hour=0):]

print("Total number of row in train test:", bike\_sharing\_test.shape[0])

print("Test dataset start date :",bike\_sharing\_test.index.min())#2017-01-03 23:00:00

print("Test dataset end date:",bike\_sharing\_test.index.max())

output:

Text

Description automatically generated

So far, we have read the data and split the into three different pandas data frames:

*bike\_sharing\_train –* This has train dataset, which is used to train our model.

*bike\_sharing\_val* – This has validate dataset, which is used to validate our model after every epoch.

*bike\_sharing\_test* – This has test dataset, when our model is trained and ready. We shall use this data to perform forecasting and compare results.

Feature Engineering

It's always good to perform normalization on our data set to produce good results and also it helps our model run faster. There are different normalizations technique available, if you have noticed we have used Robust scaler in our previous chapter, In this chapter we shall use min max scaling technique.

Here we shall we shall use min max scaler from sklearn preprocessing module and apply scaling on all the columns of our bike sharing dataset, that is on train, validate and test data frames.

In the below code block we are primarily performing these steps:

Create scalers:

* Since we have five unique columns (*t1, t2, hum, wind\_speed, cnt*)*,* We are creating five different min max scaler for each of the columns.

Create transformers:

* In this step, once our scalers are ready for each of the column for our bikesharing dataset. Let’s use our train dataset to create transformers, this can be done by invoking *fit* method for five individual scalers. Here we with the help of *fit* method and on train dataset we have created five different transformers for each column (*t1, t2, hum, wind\_speed, cnt*) for our bike sharing train dataset.

Apply Scaling:

* This is the final step, where we are applying scaling on our train, validate and test data frame. With overwriting the columns values with new scaled values. After this step we have our three dataframes train, validate and test with scaled or with normalized data.

#create scalers

t1\_scaler = MinMaxScaler()

t2\_scaler = MinMaxScaler()

humidity\_scaler = MinMaxScaler()

wind\_speed\_scaler = MinMaxScaler()

count\_scaler = MinMaxScaler()

#Create transformers

t1\_scaler\_transformer = t1\_scaler.fit(bike\_sharing\_train[['t1']])

t2\_scaler\_transformer = t2\_scaler.fit(bike\_sharing\_train[['t2']])

humidity\_scaler\_transformer = humidity\_scaler.fit(bike\_sharing\_train[['hum']])

wind\_speed\_scaler\_transformer = wind\_speed\_scaler.fit(bike\_sharing\_train[['wind\_speed']])

count\_scaler\_transformer = count\_scaler.fit(bike\_sharing\_train[['cnt']])

#Apply scaling

#Apply scaling to train dataset

bike\_sharing\_train["t1"] = t1\_scaler\_transformer.transform(bike\_sharing\_train[['t1']])

bike\_sharing\_train["t2"] = t2\_scaler\_transformer.transform(bike\_sharing\_train[['t2']])

bike\_sharing\_train["hum"] = humidity\_scaler\_transformer.transform(bike\_sharing\_train[['hum']])

bike\_sharing\_train["wind\_speed"] = wind\_speed\_scaler\_transformer.transform(bike\_sharing\_train[['wind\_speed']])

bike\_sharing\_train["cnt"] = count\_scaler\_transformer.transform(bike\_sharing\_train[['cnt']])

#Apply scaling to validate dataset

bike\_sharing\_val["t1"] = t1\_scaler\_transformer.transform(bike\_sharing\_val[['t1']])

bike\_sharing\_val["t2"] = t2\_scaler\_transformer.transform(bike\_sharing\_val[['t2']])

bike\_sharing\_val["hum"] = humidity\_scaler\_transformer.transform(bike\_sharing\_val[['hum']])

bike\_sharing\_val["wind\_speed"] = wind\_speed\_scaler\_transformer.transform(bike\_sharing\_val[['wind\_speed']])

bike\_sharing\_val["cnt"] = count\_scaler\_transformer.transform(bike\_sharing\_val[['cnt']])

#Apply scaling to validate dataset

bike\_sharing\_test["t1"] = t1\_scaler\_transformer.transform(bike\_sharing\_test[['t1']])

bike\_sharing\_test["t2"] = t2\_scaler\_transformer.transform(bike\_sharing\_test[['t2']])

bike\_sharing\_test["hum"] = humidity\_scaler\_transformer.transform(bike\_sharing\_test[['hum']])

bike\_sharing\_test["wind\_speed"] = wind\_speed\_scaler\_transformer.transform(bike\_sharing\_test[['wind\_speed']])

bike\_sharing\_test["cnt"] = count\_scaler\_transformer.transform(bike\_sharing\_test[['cnt']])

Creating Custom Dataset

In this section of chapter we shall create Dataset for our Bike sharing data, single dataset for all of our train, validate and test datasets. In later part of this section, we will be using this dataset to create dataloader. Let’s walkthrough the process of creating custom dataset using pytorch Dataset module, we shall use module available in pytorch utils data (torch.utils.data.Dataset

).

Let’s create a class with name *ClimateDataset*, which is being inherited from torch.utils.data.Dataset class. Let’s go through each method within our *ClimateDataset* in detail.

\_\_init\_\_():

**params:**

*train-* Takes a boolen as input, if true train dataset is being processed and returned

*test-* Takes a boolen as input, if true test dataset is being processed and returned

*window\_size-* size of the window or sequence to be fed into our RNN model.

\_\_init\_\_ is the method where we are loading our dataset, process and making it ready to be consumed.

For our better understanding let’s divide the code block into four different steps and below we are going through each steps involved inside init method.

STEP1: Load the data inside class

#STEP1: Load the data

self.bike\_sharing\_train = bike\_sharing\_train

self.bike\_sharing\_val = bike\_sharing\_val

self.bike\_sharing\_test = bike\_sharing\_test

In the above section of this code, at the end of feature engineering section we were ready with three different pandas data frames for each train, validate and test data frames. Here to use to data frames inside our *BikeSharingDataset* class, in above code we are making copies of those three data frames.

STEP2: Creating Features

#STEP2: Creating Features

if train: #process train dataset, if FlightsDataset is initialized with train flag

features = self.bike\_sharing\_train

target = self.bike\_sharing\_train.cnt

elif validate: #process test dataset, if FlightsDataset is initialized with test flag

features = self.bike\_sharing\_val

target = self.bike\_sharing\_val.cnt

else:

features = self.bike\_sharing\_test

target = self.bike\_sharing\_test.cnt

In this code block we are creating two variables, one is the features which consists of dataframe with our features column that is data frame with columns *t1, t2, hum, wind\_speed* and *cnt.* And second is the data frame with single column *cnt.*

If the train parameter is *True,* features and target variable are setup for our train dataset that is with *bike\_sharing\_train* dataset, if validate parameter is True, features and target variable are setup for our validate dataset that is with *bike\_sharing\_val* data frame and similarly if test parameter is True, features and target variable are setup for our test dataset that is with *bike\_sharing\_test* data frame.

STEP3: Create windows/sequencing

#STEP3: Create windows/sequencing

self.x, self.y = [], []

for i in range(len(features) - window\_size):

v = features.iloc[i:(i + window\_size)].values

self.x.append(v)

self.y.append(target.iloc[i + window\_size])

This step is similar to same step and has same code block as previous chapter, Where windowing/sequencing on our dataset for our model to perform better. At this step, we are converting our dataset into windows of default size of 240 that is data worth of 10 days, We will be training our LSTM model with the tem days of historical data to make prediction of the next day.

In the above code we have two python class variable *x,* will be having sequence of past ten days of features (*t1, t2, hum, wind\_spee* and *cnt*) and class variable y will be having the target variable(*cnt)* for the next consecutive day.

STEP4: Calculate length of dataset

Here we are calculating total number of records for our data after performing windowing, which will later be used for the \_\_len\_\_(self) method.

To summarize our \_\_init\_\_() method, which takes three boolean parameter as input and depending on the flag that is being set, It either picks up train or validate or test dataset. Then features and the target is being extracted from the dataset, performs windowing on the features and target data with default window size as 240. After windowing, this sequence data is being stored in class variables called *x* and *y,* Which represent features and target variables.

Complete code block for \_\_init\_\_() method:

def \_\_init\_\_(self, train=False, validate=False, test=False, window\_size=240):

#STEP1: Load the data

self.bike\_sharing\_train = bike\_sharing\_train

self.bike\_sharing\_val = bike\_sharing\_val

self.bike\_sharing\_test = bike\_sharing\_test

#STEP2: Creating Features

if train: #process train dataset

features = self.bike\_sharing\_train

target = self.bike\_sharing\_train.cnt

elif validate: #process validate dataset

features = self.bike\_sharing\_val

target = self.bike\_sharing\_val.cnt

else: #process test dataset

features = self.bike\_sharing\_test

target = self.bike\_sharing\_test.cnt

#STEP3: Create windows/sequencing

self.x, self.y = [], []

for i in range(len(features) - window\_size):

v = features.iloc[i:(i + window\_size)].values

self.x.append(v)

self.y.append(target.iloc[i + window\_size])

#STEP4: Calculate length of dataset

self.num\_sample = len(self.x)

\_\_len\_\_(self):

def \_\_len\_\_(self):

#returns the total number of records for data set

return self.num\_sample

This function returns the class variable *num\_sample* variable which has total number of rows in the dataset*,* this is calculated and initialized in \_\_init\_\_() .

\_\_getitem\_\_(self, index):

def \_\_getitem\_\_(self, index):

x = self.x[index].astype(np.float32)

y = self.y[index].astype(np.float32)

return x, y

In our \_\_init\_\_() method after performing windowing on our dataset, All our dataset set is being stored in two class variables x and y. This method takes index as a parameter and returns the record for both our features and target variable, that is returns value at index from both class variable x and y.

Here is the complete code snippet for the *ClimateDataset* class:

class BikeSharingDataset(torch.utils.data.Dataset):

def \_\_init\_\_(self, train=False, validate=False, test=False, window\_size=240):

#STEP1: Load the data

self.bike\_sharing\_train = bike\_sharing\_train

self.bike\_sharing\_val = bike\_sharing\_val

self.bike\_sharing\_test = bike\_sharing\_test

#STEP2: Creating Features

if train: #process train dataset

features = self.bike\_sharing\_train

target = self.bike\_sharing\_train.cnt

elif validate: #process validate dataset

features = self.bike\_sharing\_val

target = self.bike\_sharing\_val.cnt

else: #process test dataset

features = self.bike\_sharing\_test

target = self.bike\_sharing\_test.cnt

#STEP3: Create windows/sequencing

self.x, self.y = [], []

for i in range(len(features) - window\_size):

v = features.iloc[i:(i + window\_size)].values

self.x.append(v)

self.y.append(target.iloc[i + window\_size])

#STEP4: Calculate length of dataset

self.num\_sample = len(self.x)

def \_\_getitem\_\_(self, index):

x = self.x[index].astype(np.float32)

y = self.y[index].astype(np.float32)

return x, y

def \_\_len\_\_(self):

#returns the total number of records for data set

return self.num\_sample

Before we end this section, let’s quickly test our *BikeSharingDataset* so that we are understanding things better*,* one easy way to test is to use for loop and after single iteration stop the loop by calling break statement. In the below code, we are creating our *BikeSharingDataset* for the train data, looping it over single iteration and print size of the features, content of features and targets.

bike\_sharing = BikeSharingDataset(train=True)

#let's loop it over single iteration and print the shape and also data

for i, (features,targets) in enumerate(bike\_sharing):

print("Size of the features",features.shape)

print("Printing features:\n", features)

print("Printing targets:\n", targets)

break

output:

Table

Description automatically generated

Our features size muti dimensional with 240 rows and nine columns. That is we have 240 rows of historical data because our default window size is 240 and nine column because we have four unique features. Finally, there is a single future target value.

Important Note

If you overwrite the window\_size param with new value, then one may not produce same output that is size of features as (240,9).

LSTM model using Pytorch Lightening

So far, we have loaded the dataset into pandas, performed feature engineering, created custom dataset to access data. Now, we are ready with our data which is processed and converted into sequence/windowing and ready to be fed into our model.

Let’s start building our Deep Learning model, Here let’s build multi-layer doubly stacked LSTML model to forecast London bike sharing sales.

To build our LSTM model using pytorch Lightening, let’s divide the process into below steps:

1. Create Model
2. Setup Optimizer
3. Setup Data
4. Configure Traning Loop
5. Configure Validation Loop
6. Create Model

Let’s create a class called as LSTM which is inherited from Lightening Module class from pytorch lightening.

Takes the following input or parameters in the constructor:

**input\_size**: Number of features that are being used, in our case total number of features our London bike sharing dataset are 9 and also default value for this param is 9.

**hidden\_dim**: This is total number of hidden LSTM’s, that is replicas of LSTM needed. By default this value is being set to 10, that is we have 10 hidden layers.

**n\_layers**: This param is used to stack up the LSTML's over each other, for this chapter we shall use two layer neural network, and the default value for this parameter is 2.

**output\_size**: Number of output's expected from the model, since we are forecasting *cnt* which is a regression problem therefore output size is 1.

**Window\_size**: Size of window, data to be divided into. Default size is 240, that is we are considering 10 days’ worth of data.

\_\_init\_\_():

def \_\_init\_\_(self, input\_size=9, output\_size=1, hidden\_dim=10, n\_layers=2, window\_size=240):

"""

input\_size: Number of features in the input

hidden\_dim: number of hidden layers

n\_layers: number of RNN to stack over each other

output\_size: number of items to be outputted

"""

super(LSTM, self).\_\_init\_\_()

self.hidden\_dim = hidden\_dim

self.n\_layers = n\_layers

self.lstm = nn.LSTM(input\_size, hidden\_dim, n\_layers, bidirectional=False, batch\_first=True)

self.fc = nn.Linear(hidden\_dim \* window\_size, output\_size)

self.loss = nn.MSELoss()

self.learning\_rate = 0.001

In our \_\_init\_\_ method we are primarily doing three things:

* Initialize LSTM model using torch nn module and build LSTM model with the input size passed, total required hidden dimensions and with number of layers that are being passed during initializing the model. Also, setting up batch\_first parameter as True since here LSTM layer is our first layer.
* Creating Linear layer with torch nn module, where input size total number of dimensions times the window size and second parameter are the size of the output. In our case, we are predicting column *cnt*, therefore our output size is 1.
* Initializing loss function, we are using MSE loss function to calculate losses.
* Also setting up learning rate as 0.001, however in the later part of this section we shall use pytorch lightening automated way to find the learning rate.

Important Note

During the process of automated process for identifying learning rate, by deafult pytorch lighening looks for the variables with name *learning\_rate* or *lr* inside our init method. It’s always a good practice to set learning with variable names as *learning\_rate* or *lr.*

get\_hidden():

def get\_hidden(self, batch\_size):

# hidden = torch.zeros(self.n\_layers, batch\_size, self.hidden\_dim)

hidden\_state = torch.zeros(self.n\_layers, batch\_size, self.hidden\_dim)

cell\_state = torch.zeros(self.n\_layers, batch\_size, self.hidden\_dim)

hidden = (hidden\_state, cell\_state)

return hidden

This method takes batch size as input and returns tupple with containing two tensors for hidden state and cell state for our LSTM hidden layer and these two tensor are initialized with zero’s.

Forward():

def forward(self, x):

batch\_size = x.size(0)

hidden = self.get\_hidden(batch\_size)

out, hidden = self.lstm(x, hidden)

out =out.reshape(out.shape[0], -1)

out = self.fc(out)

return out

In forward method, we are connecting all the layers by configuring inputs and outputs.

* First, we are picking up the batch size with the help of the size method.
* Initializing the hidden layer, by calling our utility function called as *get\_hidden* which returns us the multi-dimensional tensor filled with zero’s.
* Once the initial hidden layer is ready, input *x* along with hidden layer is being sent to our LSTM model, which returns us output and the hidden layer.
* Before passing the output to our fully connected layer, output data is being flattened to single dimension and then sent as an input to the fully connected network.
* Finally, output from the fully connected network that is Linear layer is being returned from the forward method.

1. Setup Optimizer

When we are using pytorch lightening to build our model, it is needed to set up our optimizer inside our model class, this can be done by overwriting method called *configure\_optimizers()*

def configure\_optimizers(self):

params = self.parameters()

optimizer = optim.Adam(params=params, lr = self.learning\_rate)

return optimizer

 In this method we are primarily doing two operations:

* Get the model parameters, since this method is written inside our model class parameters for the model can be accessed calling method *self.parameters()*.
* Parameters for the model, once collected are passed to Adam optimizer, which can be accessed from the torch optim module. Along with model parameters, in above code we have setup learning rate as 0.001
* Once the parameters and the learning rate is setup, Adam optimizer is returned.

Important Note

For this use case we have Adam as our optimizers and with learning rate as 0.001. This can always be tried with different optimizers and try it out with different learning rates for the better performance of the model.

In the later part of this chapter we will be making use of pytroch lightning automated technique to determine learning rate.

1. Setup Data

For our LSTM model we are making use of train data to train the model and use validate data set for validation. Pytorch lightening need data loaders to access the data, this can be achieved by over writing *train\_dataloader* and *val\_dataloader* method, which we are going to cover in the below code block.

To train our LSTM model on our train dataset, let’s setup our train data loader by overwriting method of *pl.LightningModule* called *train\_dataloader.*

def train\_dataloader(self):

climate\_train = BikeSharingDataset(train=True)

train\_dataloader = torch.utils.data.DataLoader(climate\_train, batch\_size=50)

return train\_dataloader

In this method, we are creating a train Dataset from the BikeSharingDataset class which we have created in the initial sections of this chapter. Here, we are making use of *torch.utils.data.DataLoader* class to create the our train data loader.

*The train\_dataloader* method returns dataloader with batch size if 50 with containing train dataset.

Similar to above above step, To train our LSTM model on our validate dataset, let’s setup our validate data loader by overwriting method of *pl.LightningModule* called *val\_dataloader.*

def val\_dataloader(self):

bike\_sharing\_val = BikeSharingDataset(validate=True)

val\_dataloader = torch.utils.data.DataLoader(bike\_sharing\_val, batch\_size=50)

return val\_dataloader

In this method, we are creating a validate Dataset from the *BikeSharingDataset* class which we have created in the initial sections of this chapter. Here, we are making use of *torch.utils.data.DataLoader* class to create the our train data loader.

*The val\_dataloader* method returns dataloader with batch size if 50 with containing validate dataset.

Important Note

- When creating Dataset from our BikeSharingDataset class*,* for the train\_dataloader method we have set *train*  flag as true and for val\_dataloader method we have set val flag as true.

- In the code we have fixed the batch size as 50, depending on one’s hardware requirement and the performance this batch size can be increased or decreased as per the need.

1. Configure Training Loop

def training\_step(self, train\_batch, batch\_idx):

features, targets = train\_batch

output = self(features)

output = output.view(-1)

loss = self.loss(output, targets)

self.log('train\_loss', loss, prog\_bar=True)

return {"loss": loss}

The *training\_step* is the method where we receive train batches of data and this is the code block where we model is being trained and loss is being calculated.

Method *training\_step* takes two inputs

*batch\_idx*: Index of the batch

*batch*: This is data batch from the data loader retuned from the method *train\_dataloader.*

Below are the step by step explanation *training\_step* method:

* The batch parameter is of type tupple, from which we are collecting features which is first element of tuple and targets which is the second element of the tuple.
* To get the output of our LSTM model, one way to get this is with the help of *self.* To the self we are passing the features, which takes in the fetures as input and passes it to our LSTM model and returns the output.
* Upon output is being received from the model, same output is being converted into single dimensional array with the help of *view* pytorch method. Then, loss is being calculated.
* Before the return statement, It’s a good idea to log the training loss later on it will be easily plotted in our tensorboard. This is achieved by calling *log* method with first input as name of the log here let’s call it as *train\_log*, second input is the log itself and with setting up flag *prog\_bar* as *True.* By setting *prog\_bar* as True, *train\_loss* value will be show on progress bar during the Trainer process.
* The *training\_step* method is expected to return loss function, we are returning loss function within dictionary inside key called as *loss*.

Important Note

- Any data that is being logged can will be logged inside pytorch lightening logging directory and can also we shown inside tensorboard.

1. Configure Validation Loop

def validation\_step(self, val\_batch, batch\_idx):

features, targets = val\_batch

output = self(features)

output = output.view(-1)

loss = self.loss(output, targets)

self.log('val\_loss', loss, prog\_bar=True)

The *validation\_step* is the method where we receive train batches of data and this is the code block where we model is being validated on validation dataset and validate loss is being logged.

Method *validation\_step* takes two inputs

*batch\_idx*: Index of the batch

*val\_batch*: This is validation data batch from the data loader retuned from the method *val\_dataloader.*

Below are the step by step explanation *validation\_step* method:

* The batch parameter is of type tupple, from which we are collecting features which is first element of tuple and targets which is the second element of the tuple.
* To get the output of our LSTM model, one way to get this is with the help of *self.* To the self we are passing the features, which takes in the fetures as input and passes it to our LSTM model and returns the output.
* Upon output is being received from the model, same output is being converted into single dimensional array with the help of *view* pytorch method. Then, validationloss is being calculated.
* Before the return statement, It’s a good idea to log the training loss later on it will be easily plotted in our tensorboard. This is achieved by calling *log* method with first input as name of the log here let’s call it as *val\_log*, second input is the log itself and with setting up flag *prog\_bar* as *True.* By setting *val\_log* as True, *val\_log* value will be show on progress bar during the Trainer process.
* Important Note
* Any data that is being logged can will be logged inside pytorch lightening logging directory and can also we shown inside tensorboard.

Find Learning rate and Train the model

seed\_everything(10)

model = LSTM()

trainer = pl.Trainer( max\_epochs=20, progress\_bar\_refresh\_rate=25)

# Run learning rate finder

lr\_finder = trainer.tuner.lr\_find(model, min\_lr=1e-04, max\_lr=1, num\_training=30)

# Pick point based on plot, or get suggestion

new\_lr = lr\_finder.suggestion()

print("Suggested Learning Rate is :", new\_lr)

# update hparams of the model

model.hparams.lr = new\_lr

output:

Graphical user interface, text

Description automatically generated

In the above code, First we have created LSTM model object which is now ready to be trained. Trainer object is created with maximum epochs as 20 and refreshing progress bar for every 25 units.

From the trainer object we are calling method called *lr\_find,* This is the method which helps us to identify the optimal learning rate. Input’s passed to the *lr\_find* method are model, minimum learning rate as 0.0001, maximum learning rate as 1 and the last input is the total number of training. Optimal learning rate suggested by *lr\_find* can be accesed by calling a new method called *suggestion()* which returns optimal learning rate. In the final step, we are overwriting our learning rate hyper parameter withnew learning rate.

One easiest way to check the learning rate is as in the below snippet:

print("model learning rate:",model.hparams)

output:

A picture containing website

Description automatically generated

Now since we have identified the optimal learning rate, it’s time to train our model.

trainer.fit(model)

output:

A picture containing graphical user interface

Description automatically generated

Important Note

- Here we have applied *seed\_everything(10)* so that we can generate same results everything, easy to debug. But, this is optional and can be removed.

- Number of epochs are chosen to be 30, that is we are training our data for 30 iterations. This can we increased or decreased depending on the model performance.

- When we pass auto\_lr\_find input as True, pytorch lighthening searches for the variable with name learning\_rate or lr within our LSTM class. Make sure you have create variable with name learning\_rate or lr within LSTM class.

- Learning rate is one of the important hyperparamter and it’s not easy to determine best learning rate. Pytorch lightening, helps us to determine the optimal learning rate but it may not be the best learning rate. But, it’s a good way to start with in identifying the best learning rate.

Measuring Training Loss

In the above steps at training step, we have logged train\_loss. Now let’s use tensorboard to plot our training loss and monitor how the train loss though out each epoch. Let’s start tensor board by running below code:

%load\_ext tensorboard

%tensorboard --logdir lightning\_logs/

Output:

Graphical user interface, chart, application

Description automatically generated

The two values *train\_loss* and *val\_loss,* Which we have logged in our LSTM model can now be accessed from our tensorboard.

Here are the two screen shots for the train and validate losses, these charts helps us to monitor our model performance.

Train loss:

Graphical user interface, chart

Description automatically generated

Validation loss:

Chart, line chart

Description automatically generated

Load the model

Before we predict on the test data and compare results, which we are going to do in next section of this chapter. We need to first load the model, this can be done from the below steps:

Listing the files at pytorch lightening default path (*lightening\_logs*)

<Add screen shot here>

Once, we get to know the model file name, we can make use of below code which uses *load\_from\_checkpoint* method to load the model and change the mode to *eval.* Now the model is being loaded from file and ready to perform predictions.

PATH = 'lightning\_logs/version\_1/checkpoints/epoch=3-step=810.ckpt'

trained\_bike\_share\_LSTM = model.load\_from\_checkpoint(PATH)

trained\_bike\_share\_LSTM.eval()

 output:

Text, letter

Description automatically generated

Prediction on test dataset

This is the final step in this section, where we use model to make prediction on the test data set we created and plot the chart actual values Vs predicted values.

Steps involved in below code to perform prediction on test data set are:

* First create BikeSharingDataset object by passing *test* parameter as *True* and use dataset to create a loader with batch size of 20.
* Iterating over the data loader, collecting the features and these features are being used to make predictions over our the trained model.
* All the predicted and the actual target values are being stored in local variable called *predicted\_result* and *actual\_result.*

#Initialize Dataset

bike\_sharing\_test\_dataset = BikeSharingDataset(test=True)

bike\_sharing\_test\_dataloader = torch.utils.data.DataLoader(bike\_sharing\_test\_dataset, batch\_size=20)

predicted\_result, actual\_result = [], []

#let's loop it over single iteration and print the shape and also data

for i, (features,targets) in enumerate(bike\_sharing\_test\_dataloader):

result = trained\_bike\_share\_LSTM(features)

predicted\_result.extend(result.view(-1).tolist())

actual\_result.extend(targets.view(-1).tolist())

Before we plot the chart, let’s use our *mean\_temp\_transformer* which was created in feature engineering phase to perform inverse transform on to the dataset.

actual\_predicted\_df = pd.DataFrame(data={"actual":actual\_result, "predicted": predicted\_result})

inverse\_transformed\_values = count\_scaler\_transformer.inverse\_transform(actual\_predicted\_df)

actual\_predicted\_df["actual"] = inverse\_transformed\_values[:,[0]]

actual\_predicted\_df["predicted"] = inverse\_transformed\_values[:,[1]]

actual\_predicted\_df

 output:

Table

Description automatically generated

Now let’s plot the line chart for actual Vs predicted values.

plt.plot(actual\_predicted\_df["actual"],'b')

plt.plot(actual\_predicted\_df["predicted"],'r')

plt.show()

Here we are plotting the values for Bike sharing sales forecast actual values Vs predicted values.

The blue line represent the actual mean temperature values and the red line represent predicted values from our RNN model.

Chart, bar chart

Description automatically generated