3

Linear Regression with Tensorflow

In the last chapter we covered Tensorflow at a high level, starting with what TensorFlow is, next we learnt how to install TensorFlow after which we examined the TensorFlow ecosystem and data representation in Tensorflow. We implemented our first model looking at the success time to student success ratio scenario in our hello world in TensorFlow case study, finally we concluded chapter 2 by learning how to debug and resolving error messages in Tensorflow.

In this chapter we will start by learning the intuition behind simple and multiple linear regression at a high level, unpacking the key parts of the model building in TensorFlow and this will resolve all the burning questions left in chapter two. Yes, in our hello world case study we could predict the outcomes using our student success prediction model, but we need to know what does what? How can we evaluate our model? How do we improve our model? What if the data we have isn’t in numeric form but categorical form, how will we handle it? To resolve these questions, we get hands on with a case study in which we will take up the role of a deep Learning Engineer in a fictional company, the task is to build a salary prediction model which will be used by the HR team to determine the salary of 7 new employees. Here we will be loading the dataset, after which we will preprocess the data, then build models with TensorFlow after which we would evaluate our model’s prediction and examine ways to improve the overall performance of our model to optimally predict the salary for the new hires based on a set of attributes. Let us jump in.

At the end of this chapter, you should be able to.

* Understand what regression is
* Perform data preprocessing
* Build, compile and train a regression model with TensorFlow
* Perform model evaluation on regression models
* Perform predictions using trained regression models
* Save and load regression models

Linear Regression with TensorFlow

In chapter one we looked at different types of ML algorithms, there we talked about supervised machine learning, and we drilled down into regression and classification problems. In this chapter we will focus our search light on regression modelling while in the next chapter we will look into classification modelling. What is regression? If you forgot, here is your second chance. Regression is a type of supervised machine learning task where the label is numeric value. The goal in regression modelling is to determine the relationship between independent variable/variables and a target variable which is numeric value. In the last chapter we created a model based on a linear equation(2x+5=y). This model is a regression model in which the input value (x) was the number of hours of study and the output value (y), the student’s grade as shown in figure 3.1.

Chart, line chart

Description automatically generated

**Figure 3.1–: A plot showing the number of hours of study versus the test score of students.**

With increase in the number of hours of study there was also a corresponding increase in the test score, this case study encapsulates what simple linear regression is all about as we have only one input which determines the output or the student’s test score. In linear regression modelling we use our model to find the best values of the feature weight (m) and the bias (b) to effectively model the relationship between x and y as shown in figure 2.2.

Text, letter

Description automatically generated

**Figure 3.2–: Equation for Simple Linear Regression**

In our hello world case study, our equation was 2x+5 = y, if we input a value of 30 we get a grade of 65, this is how we want the model to think, using TensorFlow we achieved this result. Now in this chapter we have a new dataset with more than one attribute playing a role in determining the output of the model (y). When we have multiple features that impact on the numeric value of the target output, this is a case of multiple linear regression. In this case we have an equation where y is the result of the summation of the product of feature weights and input variables and the bias term as shown in figure 3.3

A picture containing text, whiteboard

Description automatically generated

**Figure 3.2–: Equation for Multiple Linear Regression**

Now we have a high level understand of the theoretical basis of simple linear and multiple linear regression. How do we evaluate our model’s performance as not every case we will come across will be as simple as 2X+y =5. We need to know what performance metrics to look at for when implementing regression models as this will be useful for our task, in your exams and in your career as a data professional now and in future. So let us discuss evaluation metrics for regression models next. Ready?

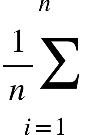
Evaluating Regression Models

From our hello world example, we tried to predict a student test score when the student has spent 43 hours studying during the term. Our study model arrived at 91 marks, when we calculated the test score by hand, we arrived at 92 marks. Yes, we are close but not completely correct, when we subtract the difference between our model’s prediction and the ground truth, we get a residual of -1. The non-negative value of the residual which is 1 is called the absolute error.

**Absolute Error** = |Ypred – Ytrue|

Where Ypred = Predicted value and Ytrue = Ground truth.

The mean absolute error(MAE) of a model is the average of all absolute errors of the data points under consideration, MAE measures the average of the residuals. And we can represent this by the equation

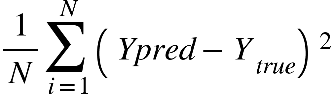
MAE = |Ypred – Ytrue|

Where n = the number of data points under consideration

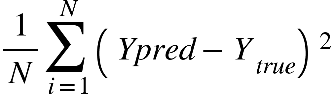
* = summation of the absolute errors of all the observations

open vertical bar Y subscript p r e d end subscript space minus Y subscript t r u e end subscript close vertical bar= Absolute value

If the MAE=0 it means Ypred=Ytrue, hence the model is 100 percent accurate and on the flip side if MAE= ∞, this means the model is completely off, probably not worth calling it a model. The large the error the large the value of the MAE, for performance evaluation, the aim we aim for low values of MAE. Another important evaluation metrics is the Mean squared error (MSE). MSE on the other hand squares the residuals, thus removes any negative values in the residuals. MSE unlike MAE penalizes larger errors since it raises in a quadratic fashion.

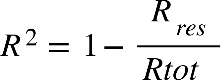
MSE= 

Like MAE when the there is no residuals we have a perfect model, so the lower the MSE value the better the performance of the model. Unlike MAE where errors large or small have a proportional impact, MSE penalizes larger errors comparison to smaller errors and MSE has a higher order of unit since we square the residual in this instant. Another useful metric in regression modelling is the root mean square (RMSE). Like the name it is the square root of the mean square error as shown in the equation.

RMSE =square root of M S E end root =

The larger the difference between the MAE and RMSE the greater the variance is between the residuals of the data points under consideration.

Lastly, let us look at Coefficient of determination (R Squared), R2 measures how well the dependent variable is explained by the independent variables in a regression modelling task. We can calculate R2 with this equation.



Where Rres is the sum of square of residuals and Rtot is the total sum of squares. The closer the value of R2 is to 1 the more accurate the model and the closer the R2 value of a model is to 0 the worse the model is. We looked at MAE, MSE, RMSE and R2, the good part is we will not be solving anything by hand, rather we will use the scikit learn library which has all these functions in built to evaluate our model. How about that as a relief?

We have breezed quickly through the theory at a high level, looking at our case study in chapter 2 which was a case of simple linear regression as we had only one input and we have also talked about multiple linear regression where we have more than one input. We have looked at some important regression metrics, now we are confident we can tackle a more complex regression problem. So, let us examine multiple linear regression with a case study which we will use to explain all the moving part required for model building in TensorFlow as well as understand how to evaluate our model, store and load models as well as use them to make predictions on new data. Let’s proceed to our case study.

Salary Prediction with TensorFlow

In this case study you will assume the role of a new Machine Learning Engineer at ABCD limited, a rapidly growing startup with over 200 employees. Now the company wants to hire 7 new employees and the human resource (HR) department have had a hard time coming up with the ideal salary based on varying qualifications, experience, roles, and training of each of the new hires. Your job is to work with the HR unit to determine the optimal salary for each of these new employees. Luckily, we went through the machine learning lifecycle in chapter one, we built our hello world case study in chapter two, and we have already covered some key evaluation metrics required for regression modelling. So, you are well equipped theoretically to carry out the task. Now, you sat down with HR to discuss all the key attributes which was used in deciding how much the company pays, thankful the HR manager in your company had ensured the company did not allow any disparity in salary based on gender or race. Now you have had a productive discussion with the HR manager, and you have a fuller understanding of the task and the requirements. Your defined your task as a supervised learning task(regression), also the HR unit allowed you to download employ records and their corresponding salaries for this task. Now you have the dataset, let us proceed with the task in our notebook.

Load Data

Open the notebook called “Linear\_Regression\_with\_TensorFlow.ipynb”. We will start by import all the necessary libraries for this project.

# import tensorflow

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import Sequential

from tensorflow.keras.layers import Dense

print(tf.\_\_version\_\_)

We run this code block, if everything goes well, we get to see the version of TensorFlow we are using.

2.8.0

Next, we import some additional libraries which would help us simplify our workflow.

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import MinMaxScaler

We now run this cell, and everything should work perfectly. Numpy is a scientific computing library in python that we use to perform mathematical operations on arrays, while pandas is a built in python library for data analysis and manipulation. Matplotlib and seaborn are used to visualize data and we will use sklearn for data preprocessing and splitting our data. We will apply these libraries in this case study, and you will get to understand what they do and also be able to apply them in your exam and beyond. Now we proceed to loading the dataset which we got from HR for this project.

#Loaing from the course github account

df=pd.read\_csv('https://raw.githubusercontent.com/oluwole-packt/datasets/main/salary\_dataset.csv')

df.head()

We use pandas to create a data frame which holds the record in a tabular manner and we use df.head() to print the first 5 entries in the data.

Graphical user interface

Description automatically generated

We now have a sense of what data was collected and the number of columns. We proceed by exploring the data to see what we can learn and how we can effectively develop a solution to meet the business objective. Let us proceed by looking at data pre-processing next.

Exploratory Data Analysis

It is always good practice to visualize your data, this gives you a sense of what the data looks like. Let us see what we can learn from visualizing the features in the data.

sns.set(style="darkgrid")

tdc =sns.scatterplot(x ='Experience', y ='Salary', data = df,

               hue ='Role')

tdc.legend(loc='center left', bbox\_to\_anchor=(1.0, 0.5), ncol=1)

We use seaborn to plot “Experience” against “Salary” and use hue to segment the Role column. When we run the code, we get the plot in figure 3.3.

Chart

Description automatically generated

**Figure 3.3–: Experience Vs Salary**

From the plot, we can see those with more experience are largely seniors and of course paid higher than those with less experience as we will naturally expect. However, we can see some employees are also well paid and they hold mid-level roles even those they just have one year experience. So, we cannot say the experience column alone determines how much an employee will earn. Let us look at another set of features with respect to the salary column. This time we take the Qualification, Salary and Cert Columns.

tdc= sns.scatterplot(x ='Qualification', y ='Salary', data = df,

               hue ='Cert')

tdc.legend(loc='center left', bbox\_to\_anchor=(1.0, 0.5), ncol=1)

When we run the code, we will the plot in figure 3.4.

Chart

Description automatically generated with medium confidence

**Figure 3.3–: Qualification Vs Salary**

From figure 3.4 we can see that again those with the same Qualification earn a varying around of salary. Phd holders have the highest salaries and a phd holder with additional certifications earns the most, same applies to Bsc and Msc holders. In our notebook we explored a few more plots, look through and see what you can learn. When you are done, let us proceed with our pre-processing task.

Data Preprocessing

From the dataframe we can immediately see that they are some irrelevant columns, and they hold personal identifiable information of employees, so we have them removed and we also inform HR about this.

#drop irrelevant columns

df =df.drop(columns =['Name', 'Phone\_Number','Date\_Of\_Birth'])

df.head()

We use the drop function in pandas to drop the name, phone number and date of birth columns. We now display the dataframe again using df.head() to show the first 5 rows of the data.

Table

Description automatically generated

We have successfully removed the irrelevant columns, so we can now proceed by checking for missing values in our dataset using the isnull function in pandas.

#check the data for any missing values

df.isnull().sum()

When we run this code block, we can see that there are no missing values in university and Salary columns. However, we have missing values for Role, Cert and Experience columns.

Experience 2

Qualification 1

University 0

Role 3

Cert 2

Salary 0

dtype: int64

There are a number of ways of handling missing values, from simply asking HR to fix the omissions INto simple imputation or replacement using mean, median or mode. In this case study we will drop the rows will missing values since it’s a small subset of our data. However, if you want to learn more about how to handle missing values, check out Hands-On Data Preprocessing in Python: Learn how to effectively prepare data for successful data analytics by Roy Jafari, it is an excellent book covering data preprocessing.

#drop the null values

df=df.dropna()

We use the dropna function to drop all the missing values in the dataset and we save the new dataset in df. Next we check to ensure they are no more missing values using the isnull function.

#check for null values

df.isnull().sum()

When we run the code, let’s see if there are any missing values.

Experience 0

Qualification 0

University 0

Role 0

Cert 0

Salary 0

dtype: int64

After running the code, we see there are no missing values anymore in our dataset. Our model requires us to pass in numeric values for it to be able to model our data and predict the target variable. So let us look at the data types next.

df.dtypes

When we run the code we get a display showing the different columns and their data types.

Experience float64

Qualification object

University object

Role object

Cert object

Salary int64

dtype: object

From the output we can see experience and salary are numeric values since they are float and int respectively. While qualification, university and cert are categorical values. This means we cannot train our model yet, we have found a way to convert our categorical values to numeric values. Luckily this is possible via a process called one hot encoding. One hot encoding is a method use to convert each category variable in our data to a new column based on that category. We use the get\_dummies function in pandas to achieve this.

#Converting categorical variables to numeric values

df = pd.get\_dummies(df, drop\_first=True)

df.head()

When we run the code, we used drop\_first to drop the first category and we get the dataframe displayed in figure 3.4.

Graphical user interface, application

Description automatically generated

**Figure 3.4–: Dataframe showing numeric values**

If you are confused as to why we dropped the one of the categorical columns, lets look at the Cert column which was made up of yes or no. If we performed one hot encoding, without dropping any columns we would have 2 cert columns as displayed in figure 3.5. In the Cert\_No column when the employee had done relevant certification the column gets a value of 0 and when the employee did not have any certification the column gets a value of 1. If we look at the Cert\_Yes column, when an employee has a certificate, the column gets a value of 1 otherwise 0.

Graphical user interface, application

Description automatically generated

**Figure 3.5–: Dummy variables from the Cert Column**

This means both columns can effectively capture both scenarios, so we can use either Cert\_No or Cert\_yes column to capture both categories, if the categories are three, we only need two columns to capture all three categories, if we have four categories, we will only need 3 columns to capture four categories and so on. Hence, we can drop the extra columns as this will help speed up our modelling process since the model would have less data to process. We use corr() function to get the correlation of our refined dataset. We can see that there is a strong correlation between salary and years of experience, also there is a strong correlation between Role\_Senior and Salary as shown in figure 3.6.

Graphical user interface

Description automatically generated with medium confidence

**Figure 3.6–: Correlation values for our data**

We have completed the preprocessing phase of our task or atleast for now. We have removed all irrelevant columns; we also removed the missing values by dropping rows with missing values and finally we used one hot encoding to convert our categorical values to numeric values. Next let us move to the modelling phase.

Model Building

To build a model we will have to sort our data into features(X) and the target(y). To do we run this code block.

# We split the attributes and labels into X and y variables

X = df.drop("Salary", axis=1)

y = df["Salary"]

We use the drop function to drop the salary column from the X variable and we make the y variable the salary column alone since this is our target. Now we have our features and target sorted, we proceed to split our data into training and testing set. The idea here is to teach our model how to make salary predictions with the training set and to test how well the model can predict on the testing set. We talked about this in our ML lifecycle in chapter one, it is a very important process as we use the test set to evaluate our model’s generalization capability before we can deploy our model to real world use. To split our data into training and testing we use sklearn library.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,

                                                    y,

                                                    test\_size=0.2,

                                                    random\_state=10)

With the Scikit learn library we split our data into training and testing data sets with a test size of 0.2 and we set the random\_state =10 to ensure reproducibility. Usually, we use 80 percent of the data for training the model and 20 percent of the data to test the model’s generalization capability. That’s why we set the test\_size to 0.2 of our dataset. Now we have everything in place, we should now start the modelling process in earnest.

#Set random set for reproducibility

tf.random.set\_seed(10)

#create a model using the Keras API

model =Sequential([Dense(units=1,input\_shape=[len(X\_train.columns)])])

#compile the model

model.compile(loss=tf.keras.losses.mae, optimizer=tf.keras.optimizers.SGD(), metrics = ['mae'])

#Fit the model

model.fit(X\_train,y\_train, epochs =50)

With the Scikit learn library We use the random.set\_seed for reproducibility of our model, now we let’s see the power of TensorFlow in its full glory. Let’s unpack the code line by line, in figure 3.7 we see the first line of code for our model. Here we created a single layer using the Sequential class as an array, the Sequential class is used for layer definition. The Dense function is used to create a layer fully connected neurons, in this case we have just one unit. Next, we have to pass in the input shape of our data which in this case is 8, which is the number of attributes (columns) in X\_train.

A picture containing text, whiteboard

Description automatically generated

**Figure 3.7–: Building a model in TensorFlow**

Now let us move to the next step which is to compile our model.

#compile the model

model.compile(loss=tf.keras.losses.mae, optimizer=tf.keras.optimizers.SGD(), metrics = ['mae'])

The good part is feed our model with training data and the corresponding labels which our model uses to make intelligent guess to arrive at a suitable mathematical model which can accurately predict the target numeric values which in our case is the expected salary. Every time the model makes a guess, the loss function compares the difference between the model’s prediction and the ground truth. This information is passed to the optimizer which uses the information to make an improved guess until the model can fashion out the right mathematical equation which will accurately predict our employee’s salary.

#Fit the model

model.fit(X\_train,y\_train, epochs =50)

Finally, we use model.fit to fit our training data and labels and set the number of tries(epochs) the model has to 50. The goal here is we want the model to be able to build a model which can predict salaries with the number of tries based on the layers we initially created with the help of the optimizer which is armed with the loss from each try. In just a few lines of code we have created a mini brain which we can train over time to make sensible predictions. Let’s run the code and see what the output looks like.

6/6 [==============================] - 0s 4ms/step - loss: 99083.4609 - mae: 99083.4609

Epoch 41/50

6/6 [==============================] - 0s 3ms/step - loss: 99082.6328 - mae: 99082.6328

Epoch 42/50

6/6 [==============================] - 0s 3ms/step - loss: 99081.7812 - mae: 99081.7812

Epoch 43/50

6/6 [==============================] - 0s 6ms/step - loss: 99080.9375 - mae: 99080.9375

Epoch 44/50

6/6 [==============================] - 0s 3ms/step - loss: 99080.1016 - mae: 99080.1016

Epoch 45/50

6/6 [==============================] - 0s 3ms/step - loss: 99079.2734 - mae: 99079.2734

Epoch 46/50

6/6 [==============================] - 0s 3ms/step - loss: 99078.4297 - mae: 99078.4297

Epoch 47/50

6/6 [==============================] - 0s 4ms/step - loss: 99077.5859 - mae: 99077.5859

Epoch 48/50

6/6 [==============================] - 0s 3ms/step - loss: 99076.7500 - mae: 99076.7500

Epoch 49/50

6/6 [==============================] - 0s 4ms/step - loss: 99075.9062 - mae: 99075.9062

Epoch 50/50

6/6 [==============================] - 0s 4ms/step - loss: 99075.0625 - mae: 99075.0625

We print the last 10 tries (epoch 41-50) the error drops gradually, however we end up with a very large error after 50 tries. Perhaps we can try our model for longer like we did in chapter 2. Why not?

#Set the random seed

tf.random.set\_seed(10)

#create a model

model1 =Sequential([Dense(units=1,input\_shape=[len(X\_train.columns)])])

#compile the model

model1.compile(loss="mae", optimizer="SGD")

#fit the model

history1= model1.fit(X\_train,y\_train, epochs =500) #number of times the model will go through training examples

Now we simply change the number of epochs to 500 using our single layer model, the loss and optimizers are the same as our initial model.

Epoch 495/500

6/6 [==============================] - 0s 3ms/step - loss: 98701.3516

Epoch 496/500

6/6 [==============================] - 0s 2ms/step - loss: 98700.5078

Epoch 497/500

6/6 [==============================] - 0s 2ms/step - loss: 98699.6641

Epoch 498/500

6/6 [==============================] - 0s 4ms/step - loss: 98698.8359

Epoch 499/500

6/6 [==============================] - 0s 3ms/step - loss: 98697.9922

Epoch 500/500

6/6 [==============================] - 0s 3ms/step - loss: 98697.1484

From the last 5 lines of our model output, we can see that the loss is still quite high after 500 epochs. In our notebook we increase the number of epochs to 1000 and in figure 3.8 we plot the model loss after 500 and 1000 epochs respectively, although we can see the loss falling, but the rate at which it falls is too slow. In your spare time you could try to train the model for 2000 or more epochs. It will not be able to generalize well as the model is too simple to handle the complexity of the data, in machine learning lingo we say the model is underfitting.

Chart

Description automatically generated with low confidence Chart

Description automatically generated with low confidence

**Figure 3.8–: Model Losses for 500 and 1000 epochs**

Hence let us try to build a more complex model and see how if we can push the loss lower and quicker than our initial model.

#Set random set

tf.random.set\_seed(10)

#create a model

model3 =Sequential([

                   Dense(units=64, input\_shape=[len(X\_train.columns)]),

                   Dense(units=1)

                   ])

#compile the model

model3.compile(loss="mae", optimizer="SGD")

#Fit the model

history3 =model3.fit(

    X\_train, y\_train, epochs=500)

Now we create a new model, we stack a 64-neuron layer on top of our single unit layer, everything else is the same. Let’s run it for 500 epochs and see what if this would make any difference. Let’s do this. From figure 3.9, we can see that in less than 500 epochs our loss has falling below 5000, this is a massive improvement in comparison to our previous model. However, we don’t have the silver bullet which we will like to show the HR team yet because if an employee earns $50,000, the model could predict $45,000, the employee will not be happy about or $55000 which HR will not be happy about, so we need to figure out how to improve our result. From the plot we can see the loss seems to fall sharply and settles below 5000 and nothing significant seems to happening after about 300 epochs, hence training the model for longer just as we did in our previous model may not be the optimal solution. So what can we do to improve our model?

A picture containing shape

Description automatically generated

**Figure 3.9–: Model Losses after 500 epochs**

Perhaps we add another layer? Let’s mix it up a little with a different optimizer and see what we get in our experimentation, as we initially agreed, our job requires a lot of experimentation, then we are able to learn how to do things better and faster. Let’s try it together and see what our result will look like.

#Set random set

tf.random.set\_seed(10)

#create a model

model4 =Sequential([

                   Dense(units=64,input\_shape=[len(X\_train.columns)]),

                    Dense(units=32),

                    Dense(units=1)

                   ])

#compile the model

model4.compile(loss="mae", optimizer="Adam")

#fit the model

history4 =model4.fit(

    X\_train, y\_train, epochs=500)

We added another dense layer of 32 neurons, and we changed the optimizer to Adam, yes we haven’t discussed about the various types of optimizers and loss metrics, we will cover the metrics soon and in the second section of this book we will cover optimizers in details for now you can interchange the optimizers and see what results you will come up with in your own experiments. Let us see what our result looks like.

Epoch 496/500

6/6 [==============================] - 0s 3ms/step - loss: 5188.7056

Epoch 497/500

6/6 [==============================] - 0s 3ms/step - loss: 5181.2739

Epoch 498/500

6/6 [==============================] - 0s 3ms/step - loss: 5175.6411

Epoch 499/500

6/6 [==============================] - 0s 3ms/step - loss: 5174.4092

Epoch 500/500

6/6 [==============================] - 0s 4ms/step - loss: 5164.8862

We display only the last 5 epochs, and we can see the loss is around 5164, which is slightly worse than the previous model. So how do we know how many layers to use in our modelling process? The answer is by experimenting. We use trial and error, backed with our understanding of what the results looks like we can decide if we need to add more layers as we did initially when the model was underfitting, and should the model get to complex such that it masters the training data so well but does not generalize well on our test data (hold out data) such a model is said to be overfitting in machine learning lingo.

Now that we have tried smaller and larger models, yet we cannot say we have achieved a suitable result and the HR manager has checked on us to find out how we are doing in terms of the prediction modelling task. So we did some research, all ML Engineers do and we found out a very important step we can try out. What step? Let’s see.

Normalization

Its good practice to know that improving your model output can also lie strongly with your data preparation process. Hence let us apply this here. Let’s take a step back from model building and take a look at our features after we had converted all the columns to numeric values.

X.describe()

We use the describe function to get vital statistics of our data. This information shows us that most of the columns have a min value of 0 and max of 1, but the experience column is of a different scale as shown in figure 3.9.

Table

Description automatically generated with medium confidence

Why does this matter you may ask? When the scale of our data is different, our model will arbitrarily attach more importance to columns with higher values which could affect the model’s ability to predict our target less accurately, to resolve this issue we use normalization to scale the data between 0 and 1 so as to bring all our features to the same scale, hence giving every feature an equal chance when our model begins to learn how they relate to our target (y).

To normalize our data, we use the following equation to scale our data.

A whiteboard with writing on it

Description automatically generated with low confidence

Where X is our data, Xmin is the minimum value of X, Xmax is the maximum value of X. In our case study the minimum value of X in the experience column is 1 and the maximum value of X in the experience column is 7. The good part is we can easily implement this step using the MinMaxScaler from scikit learn library.

# create a scaler object

scaler = MinMaxScaler()

# fit and transform the data

X\_norm = pd.DataFrame(scaler.fit\_transform(X), columns=X.columns)

X\_norm.describe()

Now we scale our data and use the describe function to view the key statistics again.

A picture containing table

Description automatically generated

Now all our data are of the same scale, so we have successfully implemented normalization of our data in a few lines of code.

# Create training and test sets with the normalized data (X\_norm)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_norm,

                                                    y,

                                                    test\_size=0.2,

                                                    random\_state=42) # set random state for reproducible splits

Next, we split our data into training and testing set. But this time we use our normalized X (X\_norm) in the code.

#Set random set

tf.random.set\_seed(10)

#create a model

model5 =Sequential([

                   Dense(units=64, activation='relu', input\_shape=

[len(X\_train.columns)]),

                   Dense(units=1)

                   ])

#compile the model

model5.compile(loss="mae", optimizer=tf.keras.optimizers.RMSprop(0.001), metrics = ['mae'])

#fit the model

early\_stop=keras.callbacks.EarlyStopping(monitor='loss', patience=10)

history5 =model5.fit(

    X\_train, y\_train, epochs=1000, callbacks=[early\_stop])

Now we use our best model from the initial experiments, lets see how our model performs after normalization.

Epoch 996/1000

6/6 [==============================] - 0s 2ms/step - loss: 91522.7734 - mae: 91522.7734

Epoch 997/1000

6/6 [==============================] - 0s 3ms/step - loss: 91511.1250 - mae: 91511.1250

Epoch 998/1000

6/6 [==============================] - 0s 3ms/step - loss: 91499.4297 - mae: 91499.4297

Epoch 999/1000

6/6 [==============================] - 0s 3ms/step - loss: 91487.6562 - mae: 91487.6562

Epoch 1000/1000

6/6 [==============================] - 0s 3ms/step - loss: 91475.8438 - mae: 91475.8438

From the results we can see that the model did not generalize properly after normalization. Does this mean normalization did not help? Before you conclude let us run a larger model and see what results we get.

#Set random set

tf.random.set\_seed(10)

#create a model

model6 =Sequential([

                   Dense(units=64, input\_shape=[len(X\_train.columns)]),

                    Dense(units=64),

                    Dense(units=1)

                   ])

#compile the model

model6.compile(loss="mae", optimizer="Adam")

#fit the model

early\_stop=keras.callbacks.EarlyStopping(monitor='loss', patience=10)

history6 =model6.fit(

    X\_train, y\_train, epochs=1000, callbacks=[early\_stop])

We introduce an extra layer of 64 neurons, before normalization we used this model, and it did not do a great job. Now let’s check out the result and see what if we should discard the idea of normalizing our data or not. We run the code for 1000 epochs, we introduce early stopping to ensure the model stops training when the loss stops dropping, since there is no point training further when the model is not improving. Also, early stopping is a very useful method to resolving overfitting.

Epoch 738/1000

6/6 [==============================] - 0s 3ms/step - loss: 50.3793

Epoch 739/1000

6/6 [==============================] - 0s 3ms/step - loss: 37.0718

Epoch 740/1000

6/6 [==============================] - 0s 3ms/step - loss: 25.7887

Epoch 741/1000

6/6 [==============================] - 0s 3ms/step - loss: 13.3677

Epoch 742/1000

6/6 [==============================] - 0s 3ms/step - loss: 18.6641

Finally, we have achieved a massive drop in the model loss. Our model has massively improved, and we only needed 742 epochs and the model stopped training due to the early stopping we introduced to our model. Can we do better than this result? Let us try out the same model architecture with a different optimizer and see what will happen?

#Set random set

tf.random.set\_seed(25)

#create a model

model7 =Sequential([

                   Dense(units=64, activation='relu', input\_shape=[len(X\_train.columns)]),

                    Dense(units=64, activation ="relu"),

                   Dense(units=1)

                   ])

#compile the model

model7.compile(loss="mae", optimizer="SGD", metrics ="mae")

#fit the model

early\_stop=keras.callbacks.EarlyStopping(monitor='loss', patience=10)

history7 =model7.fit(

    X\_train, y\_train, epochs=1000, callbacks=[early\_stop])

We run the code and lets see the result of this change in optimizer.

6/6 [==============================] - 0s 3ms/step - loss: 73661.7266 - mae: 73661.7266

Epoch 44/1000

6/6 [==============================] - 0s 3ms/step - loss: 72043.4766 - mae: 72043.4766

Epoch 45/1000

6/6 [==============================] - 0s 2ms/step - loss: 75473.7812 - mae: 75473.7812

Epoch 46/1000

6/6 [==============================] - 0s 2ms/step - loss: 91920.0625 - mae: 91920.0625

Epoch 47/1000

6/6 [==============================] - 0s 2ms/step - loss: 70324.9062 - mae: 70324.9062

The result here is just as bad as before normalization. So, we have tried different experiments, and we can see we got our best result after normalizing our data. But this is not the complete story, we are not done yet. Remember our test data (hold out data set), let our model on the test data and see if it generalizes properly, before we rush off to announce good work. Let’s proceed us discuss regression metrics and evaluate the various models we have built on the test set, lets see what the results will look like.

Model Evaluation

To evaluate our models, we write a function to generate the evaluate metrics for all 7 models.

def eval\_testing(model):

  return model.evaluate(X\_test, y\_test)

models = [model1, model2, model3, model4, model5, model6, model7]

for x in models:

  eval\_testing(x)

We use model.evaluate() function to evaluate the features and attributes. In this function we pass our testing data and target value in. After which we generate the mean absolute error for all the models.

2/2 [==============================] - 0s 6ms/step - loss: 93623.4453

2/2 [==============================] - 0s 7ms/step - loss: 93227.1250

2/2 [==============================] - 0s 6ms/step - loss:

2007.0309

2/2 [==============================] - 0s 10ms/step - loss: 5321.0693

2/2 [==============================] - 0s 7ms/step - loss: 85319.8750

2/2 [==============================] - 0s 5ms/step - loss:

24.2513

2/2 [==============================] - 0s 6ms/step - loss: 94019.1250

After we evaluate the models, we can see model 6 has the lowest loss. Although the loss on the test set is slightly higher than the training set, this is good as the different was not far off. We can now proceed to compare the model’s prediction and the ground truth using the test data.

Making Predictions

Now we are done with our experimenting, and we have evaluated the models. Lets use model6 to predict our test set salaries and see how they compare with the ground truth. To do this we use the function model.predict().

#Lets make predictions on our test data

y\_preds=model6.predict(X\_test).flatten()

y\_preds

After we run this code block. We get the output in an array.

array([ 64510.375, 131534.94 , 116520.95 , 72515.24 , 103016.12 ,

60516.19 , 84514.39 , 119535.78 , 112526.766, 63508.203,

78030.34 , 84506.17 , 112526.766, 91016.97 , 87506.41 ,

100527.72 , 135529.11 , 112518.54 , 119535.78 , 131534.94 ,

109022.95 , 117537.75 , 80511.984, 123529.97 , 112526.766,

117537.75 , 112029.56 , 79017.92 , 135529.11 , 129536.805,

117537.75 , 119535.78 , 100527.72 , 113535.336, 102020.305,

113535.336, 94031.805, 65506.234, 61503.773, 107536.734,

106030.94 , 106526.33 , 72507.016, 135529.11 , 67510.61 ,

107536.734, 117537.75 , 70517.22 , 57509.586], dtype=float32)

For clarity lets build a dataframe with the model’s prediction and ground truth. This should be fun and somewhat magical when you see how good our model has gotten.

#Lets make a dataframe to compare our prediction with the ground truth

df\_predictions = pd.DataFrame({'Ground\_Truth': y\_test, 'Model\_prediction': y\_preds}, columns=[ 'Ground\_Truth', 'Model\_prediction'])

df\_predictions['Model\_prediction']= df\_predictions['Model\_prediction'].astype(int)

Here we create two columns and we convert the model’s prediction from float to int, just to keep it in scope with the ground truth. Ready for the result?

#Lets look at top 10 data points in the test set

df\_predictions.head(10)

We use the head function to print out the first 10 values of the test set.

Table

Description automatically generated

We see our results in figure 3.10. Our model has achieved something majestic, it is real close to the initial salaries in our test data. Now you want to show the HR manager your amazing result tomorrow. Now we have to save the model, so we are able to load it and make predictions on it anytime we want. Lets learn how to do this next.

Saving and Loading Models

The beauty of TensorFlow is the easy with which we can do complex stuff, to save a model we just need one line of code.

#Saving the model in one line of code

model6.save('salarypredictor.h5')

#Alternate method is

#model6.save('salarypredictor')

Either you save it as “your\_model.h5” or “your\_model”, either way works. I prefer the first method, so let’s use it here. When we run the code, we can see the saved model on the left-hand panel in our colab notebook as in figure 3.11.

Graphical user interface, text, application

Description automatically generated

Now that we have saved the model, it is wise to test it out by reloading it and testing it. Lets do that next. Also, its just another one line of code to load the model.

#loading the model

saved\_model =tf.keras.models.load\_model("/content/salarypredictor.h5")

Lets try out our saved\_model and see if it will work as well as model6. We generate y\_pred again and we create a dataframe using y\_test and y\_pred as we did early.

Table

Description automatically generated

The results in figure 3.12 is the same as that in figure 3.10 so we are good to go. Now you deliver your result to the HR manager, and he is excited with your results. So, he wants you to use your model to predict the salary of the new hires. So, let’s do that next.

#Putting everything into a function for our big task

def salary\_predictor(df):

  df\_hires= df.drop(columns=['Name', 'Phone\_Number','Date\_Of\_Birth' ])

  df\_hires = pd.get\_dummies(df\_hires, drop\_first=True)

  X\_norm = pd.DataFrame(scaler.fit\_transform(df\_hires), columns=df.columns)

  y\_preds=saved\_model.predict(X\_norm).flatten()

  df\_predictions = pd.DataFrame({ 'Model\_prediction': y\_preds}, columns=[ 'Model\_prediction'])

  df\_predictions['Model\_prediction']= df\_predictions['Model\_prediction'].astype(int)

  df['Salary']=df\_predictions['Model\_prediction']

  return df

We create a function using our saved model. We simply wrapped all the steps we covered so far into the function and we return a dataframe. Next lets read in our the data of our new hires.

#Load the data

df\_new=pd.read\_csv('https://raw.githubusercontent.com/oluwole-packt/datasets/main/new\_hires.csv')

df\_new

When we run the code block, we see their data in a dataframe as shown in figure 3.13.

Graphical user interface

Description automatically generated

Now we pass the data into our the function we created so as to get the predicted salaries for our new hires.

#Lets see how much

salary\_predictor(df\_new)

We pass df\_new into the salary prediction function and we get a new dataframe as shown in figure 3.14

Graphical user interface

Description automatically generated with medium confidence

Finally, we have achieved our goal. HR is happy, the new hires are happy and everyone in the company thinks you are a magician. Perhaps a pay raise could be on the table, while you are basking in the euphoria around your first success, your manager is back with another task. This time it is a classification task, in the next chapter we will look at it. For now, good job.

Summary

In this chapter we took a deeper dive into supervised learning, with our focus on regression modelling. Here we discussed the difference between Simple and Multiple linear regression, we looked at some important evaluation metrics, then we rolled up our sleeves in our case study, helping our company to build a working regression model to predict salaries of new employees. Here we carried out some data preprocessing steps and saw the important of normalization in our modelling process. At the end of the case study, we successfully built a salary prediction model, evaluated the model on our test set, saved and loaded the model for prediction at a later stage. Now you can confidently build a regression model with TensorFlow.

Questions

Let’s test what we learnt in this chapter.

1. What is Linear Regression?
2. What is the different between Simple and Multiple Linear Regression?
3. What evaluation Metrics penalizes large errors in regression modelling?
4. Use the Fuel Efficiency datasets, forecast the mpg values.

Further Reading

To learn more, you can check out the following resources:

Amr, T., 2020. *Hands-On Machine Learning with scikit-learn and Scientific Python Toolkits*. [S.l.]: Packt Publishing.

Raschka, S. and Mirjalili, V., 2019. *Python Machine Learning*. 3rd ed. Packt Publishing

https://www.TensorFlow.org/guide