Alexsander Jones

Caleb Koresh

Jacob Nakamura

Capstone Final Paper

Overview

The aim of our group's capstone project was to test the effectiveness of neural networks with varying architecture at predicting F1 race times based on practice and qualifying session data. Data was collected from the 2022 racing season, including 21 races and 22 drivers. For each of the 32 recorded statistics used in analysis, there were over 17,000 data points shared across all races and drivers. Several external packages such as fastf1, pandas, matplotlib, numpy, dfply, tensorflow, and sklearn were used. Other than the built-in functionalities offered by these packages, all source code was written originally by the team and was not taken from open source repositories.

In terms of responsibility, Alexsander was responsible for data scraping, Caleb, for neural networks, and Jacob for data visualization. Our group found that even in the presence of sources of error, the best of the neural networks created were fully capable of low-error predictions for qualifying race data.

Background

F1 Racing

Formula One is the highest level of open wheel motorsport in the world, where drivers compete in various Grands Prix throughout the season. Each Grand Prix is in a corresponding Race Weekend, which is scheduled as follows:

- Friday: Free Practice 1 and Free Practice 2, where drivers will practice on the race track to collect data.
- Saturday: Free Practice 3 and Qualifying, where drivers will attempt to set the fastest lap times during Qualifying to determine grid order for the race.
- Sunday: Grand Prix Race, where drivers will race around the track for victory.

  Note: Some Grands Prix are classified as Sprint Race Weekends, where only the first Free Practice is done before qualifying

Our objective for the project is to use data collected from the Free Practice Sessions to determine Qualifying times and as such, grid order as well.

Neural Network Theory

Since none of our group members had real experience with neural networks prior to this project, there was time invested into learning the theory behind the algorithms. The responsibility of gathering information and learning the content in depth fell to Caleb but the concepts were then taught to the entire group in order to provide a level of intuition behind the predictions being made.

The foundational understanding of neural networks came from the YouTube channel 3Blue1Brown, which is an educational channel created by Dr. Grant Sanderson that covers a variety of high level mathematical concepts. The link to the videos is found in the resources section of the paper. The topics covered include basic concepts and a mathematical overview of gradient descent and backpropagation, which are the algorithms that allow the network to discover complex patterns in data. The rest of this section will be dedicated to describing an overview of the theory.

The name neural network stems from the similarities between the models and the human brain, hence the word neural. Although, the similarities do not go much further than the fact that a neural network is a web of neurons that learn to solve problems. Each neuron just represents a function and the most basic form of a network consists of layers of these neurons that pass the output of all of their functions into the input of the following layer. The best intuition we were able to come up with behind why this complex web of functions is able to solve problems and learn is that the first layer starts with raw data and is able to find low level patterns which can be passed on. Each subsequent layer is able to work on higher level concepts and pass on the results to the following layer. For example, in digit recognition each layer may find patterns in a slightly larger collection of pixels than the previous layer until the output is able to tell which digit was written.

This may seem like an impossible math problem to solve with thousands of parameters that must be optimized in order to accurately make predictions based on input data, but it can be represented neatly with linear algebra. Linear algebra allows us to represent each layer as a single function with inputs and outputs that are vectors of numbers rather than individual ones. This looks like:

$$\mathbf{Z = f(W0 + W1X1 + W2X2 + \ldots + WnXn)}$$

Z = The output vector consisting of the output of each neuron

W = The weight matrix which allows each neuron to scale the importance of each neuron in the previous layer

X = The output vector of the previous layer

f() = The function that the linear equation is passed into which returns the output for each neuron

Note: Vectorization makes this process extremely efficient by utilizing a computer's GPU to make many calculations at the same time.

The next problem becomes how to tune the weight matrix in order to find complex patterns in the data and make predictions. The first thing a network needs is a well defined cost function, which is another name for an error function. The function we used was the mean squared error function because it was covered in class. It is defined as the average squared difference of the prediction the network makes on the training data and the desired output. In python code it would look like: (1 / n) * np.sum((y - y_prediction) ** 2). The function takes in all the weights as input and outputs how poorly the network is performing on average. Once this has been defined, calculus can be used to find a minimum of the cost function by following the negative of its gradient. The gradient refers to the matrix of partial derivatives corresponding to the weight matrix. By calculating the gradient and making small changes to each weight according to this gradient repeatedly the network will eventually reach a minimum cost where the partial derivatives are all approximately zero. This process is called gradient descent and when it is complete the network has solved the training data by finding the perfect combination of weights to get from the input vector to the desired output vector.

In practice, the process of computing these partial derivatives is a complex chain rule problem. Since the cost is a function of the outputs of each layer and the output of each layer is a function of its weights, the problem becomes a chain rule problem starting from the output. The process is quite complicated and an intuitive understanding was enough to produce functional code so Caleb did not delve into the complexities of this process, which is known as backpropagation. The intuition behind it is that the function looks at the output and what direction would bring it closer to the desired output. From there it looks at how the previous layers outputs could be adjusted to bring it closer to this goal. This process is then repeated all the way back to input. In summary, it looks at the desired output and travels backwards while making small changes along the way to get to that desired output.

Program Structure

The program is generally compartmentalized into 3 sections: data scraping, neural networks, and data visualization. Data scraping involved cleaning, reorganizing, and compiling data into a master dataframe. The neural network section involved numerous variations of trained models to find a network architecture that performed best on practice data. The data visualization section involved development of a modular framework for auto-generating subplots across races and drivers for a given statistic. Details for each section are provided below.

Data Cleaning and Compiling

For data compiling and cleaning, two main packages were used:

- FastF1: FastF1 is a package allowing easy access to Formula One Session data, including timing data, tire data, weather data, etc.
- Dfply: dfply is a package that mimics the ‘dplyr’ package in R, allowing for similar syntactical code and data-manipulation. The reason dfply was used is because Alex is

most familiar with data manipulation and cleaning in R, and in the interest of keeping all of our code in python, similar packages were needed. In addition to this, dfply allows for piping operations, massively improving the readability of our code.

To create the data-frame we need, three main functions were constructed:

- practiceScrape: practiceScrape was used to collect any relevant data from the practice sessions, most importantly data for timed laps.
- qualiScrape: qualiScrape is much like the practiceScrape function, but only collected data for a driver's fastest lap, since that is the only qualifying data we needed.
- weekendCompiler: weekendCompiler is the central function for data collection. For any given race weekend, weekendCompiler will use practiceScrape to collect any relevant practice data, qualiScrape to collect qualifying data, and will merge them together, so each timed practice lap has a corresponding qualifying time.

For a deeper look into the data collection and cleaning process, we will look at the practiceScrape function.

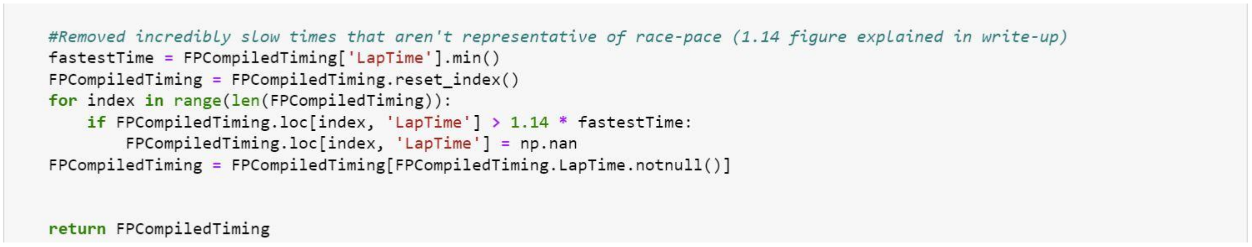
To start off, we need to create a path for fastF1.api to collect data from the correct session. After this is done, data is collected from the timing data (which gives us lap times and information about the specific lap), the timing app data (which gives us information on the tire compounds used), and the weather data. After this, we start rounding the session times to the nearest minute so there's no errors when we merge these separate data frames together.

```
def practiceScrape(grandPrix, raceDate, pracNum, pracDate):
    '''
    Input: Grand Prix, Grand Prix Date, Which Practice, Which Date
    Output: Dataframe with each driver's timed lap matched with tire compound and weather conditions
    '''
    #Make Path to scrape practice data
    FP = fastf1.api.make_path(grandPrix, raceDate, f'Practice {pracNum}', pracDate)

    #Scraping timing data, timing app data (for info on tire compund), and weather data
    FPTiming = fastf1.api.timing_data(FP)[0]
    FPTimingApp = fastf1.api.timing_app_data(FP)
    FPWeatherDict = fastf1.api.weather_data(FP)
    FPWeatherData = pd.DataFrame.from_dict(FPWeatherDict)

    #Remove entries with no laptimes, and round session time to nearest minute (for merging purposes)
    FPTiming = FPTiming[FPTiming.LapTime.notnull()]
    FPTimingApp >>= mutate(Time = X.Time.round('60s')) >> arrange(X.Driver, X.Time)
    FPTimingApp = FPTimingApp.reset_index()
```

Something to note: the “>>=” and “>>” are piping operators used by dfply, which allows for nesting to be written linearly and without having to write in the central dataframe as an argument every single time. Some issues arose in the uncleaned dataframes, namely that there were columns that were completely useless (like “LapCountTime” and “TyresNotChanged”) so those were dropped.

```
    #In timing app dataframe, tire compound is only noted when new set is put on
    #Since dataframe is organized by driver and then time, we can replace empty entries with the previous tire compound
    #Also, dataframe sometime doubles on driver-time combo, so removing the unnecessary ones
    for index in range(len(FPTimingApp)):
        if not FPTimingApp.loc[index,'Compound']:
            FPTimingApp.loc[index, 'Compound'] = FPTimingApp.loc[index-1, 'Compound']
        if not FPTimingApp.loc[index,'New']:
            FPTimingApp.loc[index,'New'] = False
        if index == 0:
            continue
        if FPTimingApp.loc[index,'Time'] == FPTimingApp.loc[index-1,'Time']:
            if not FPTimingApp.loc[index,'LapTime']:
                FPTimingApp.loc[index,'Driver'] = np.nan
            else:
                FPTimingApp.loc[index-1,'Driver'] = np.nan
    FPTimingApp = FPTimingApp[FPTimingApp.Driver.notnull()]

    #Selecting relevant rows, and rounding FPtiming session time to nearest minute
    FPTimingApp >>= select(X.Stint, X.Driver, X.TotalLaps, X.Compound, X.New, X.Time)
    FPTiming = FPTiming >> mutate(Time = X.Time.round('60s')) >> arrange(X.Driver, X.Time) >> \
             drop(contains('Session'), X.PitOutTime, X.PitInTime)

    #Merge Dataframes, keyed by time and driver, then dropping irrelevant columns
    FPCompiledTiming = pd.merge(FPTiming,FPTimingApp, on=['Time', 'Driver'])
    FPWeatherData >>= mutate(Time = X.Time.round('60s'))
    FPCompiledTiming = pd.merge(FPCompiledTiming, FPWeatherData, on=['Time'])

    #Final Cleaning, converting times into numerics
    FPCompiledTiming >>= arrange(X.Driver, X.Time) >> mutate(Practice = pracNum) >> drop(X.WindDirection, X.Stint, X.TotalLaps)\
                     >> mutate(LapTime = X.LapTime/timedelta(seconds = 1)) >> mutate(Sector1Time = X.Sector1Time/timedelta(secon
                     >> mutate(Sector2Time = X.Sector2Time/timedelta(seconds = 1)) >> mutate(Sector3Time = X.Sector3Time/timedelt
```

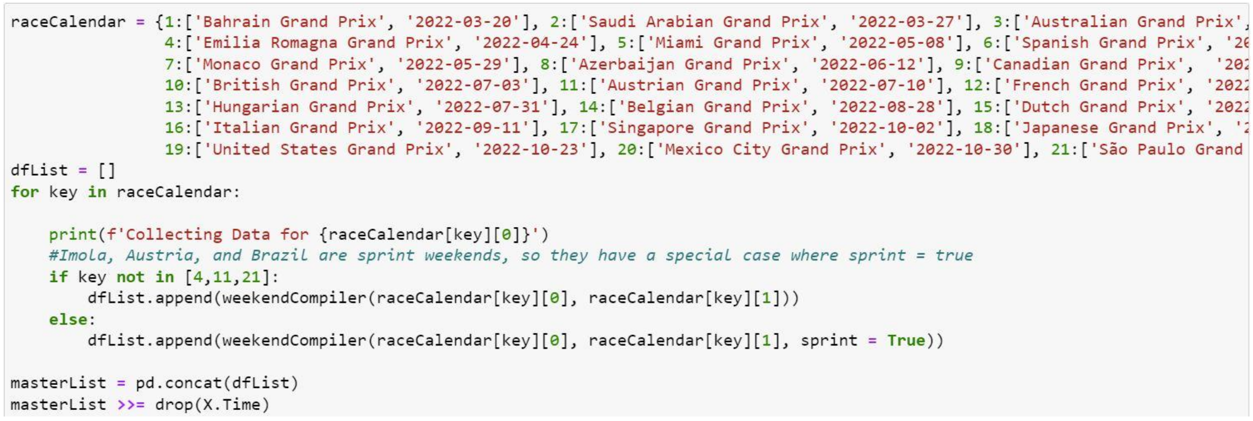
One last thing that was done was removing incredibly slow lap times, or lap times that were not representative of race pace. These lap times exist for a myriad of reasons, generally because the driver just got out of the pitlane and is taking a warm up lap, so this data is not representative of the driver's actual ability.



```
    #Removed incredibly slow times that aren't representative of race-pace (1.14 figure explained in write-up)
    fastestTime = FPCompiledTiming['LapTime'].min()
    FPCompiledTiming = FPCompiledTiming.reset_index()
    for index in range(len(FPCompiledTiming)):
        if FPCompiledTiming.loc[index, 'LapTime'] > 1.14 * fastestTime:
            FPCompiledTiming.loc[index, 'LapTime'] = np.nan
    FPCompiledTiming = FPCompiledTiming[FPCompiledTiming.LapTime.notnull()]


    return FPCompiledTiming
```

To do this, we created a threshold that relevant lap times are within 114% times of the fastest time in the session. The reason 114% is the cut off threshold is because of a simple rule in qualifying. To participate in the Grand Prix, the driver's fastest qualifying time must be within 107% of the pole position time, or they will be disqualified from the race. Keeping with the 7% trend, we decided to give drivers a bit more leniency during practice, which also gave us more data to work with. With this done, the dataframe is sufficiently cleaned and is sent out.

To create a complete dataframe, we created a dictionary with every Grand Prix with its race date, and ran it through the weekendCompiler function to create the dataframe for each weekend. With these created, all the data was concatenated into one master list.

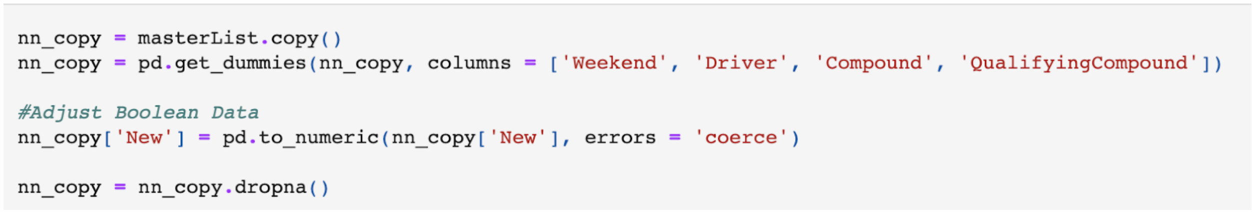


```
raceCalendar = {1:['Bahrain Grand Prix', '2022-03-20'], 2:['Saudi Arabian Grand Prix', '2022-03-27'], 3:['Australian Grand Prix',
                4:['Emilia Romagna Grand Prix', '2022-04-24'], 5:['Miami Grand Prix', '2022-05-08'], 6:['Spanish Grand Prix', '20
                7:['Monaco Grand Prix', '2022-05-29'], 8:['Azerbaijan Grand Prix', '2022-06-12'], 9:['Canadian Grand Prix',  '202
                10:['British Grand Prix', '2022-07-03'], 11:['Austrian Grand Prix', '2022-07-10'], 12:['French Grand Prix', '2022
                13:['Hungarian Grand Prix', '2022-07-31'], 14:['Belgian Grand Prix', '2022-08-28'], 15:['Dutch Grand Prix', '2022
                16:['Italian Grand Prix', '2022-09-11'], 17:['Singapore Grand Prix', '2022-10-02'], 18:['Japanese Grand Prix', '2
                19:['United States Grand Prix', '2022-10-23'], 20:['Mexico City Grand Prix', '2022-10-30'], 21:['São Paulo Grand
dfList = []
for key in raceCalendar:

    print(f'Collecting Data for {raceCalendar[key][0]}')
    #Imola, Austria, and Brazil are sprint weekends, so they have a special case where sprint = true
    if key not in [4,11,21]:
        dfList.append(weekendCompiler(raceCalendar[key][0], raceCalendar[key][1]))
    else:
        dfList.append(weekendCompiler(raceCalendar[key][0], raceCalendar[key][1], sprint = True))

masterList = pd.concat(dfList)
masterList >>= drop(X.Time)
```

Neural Networks

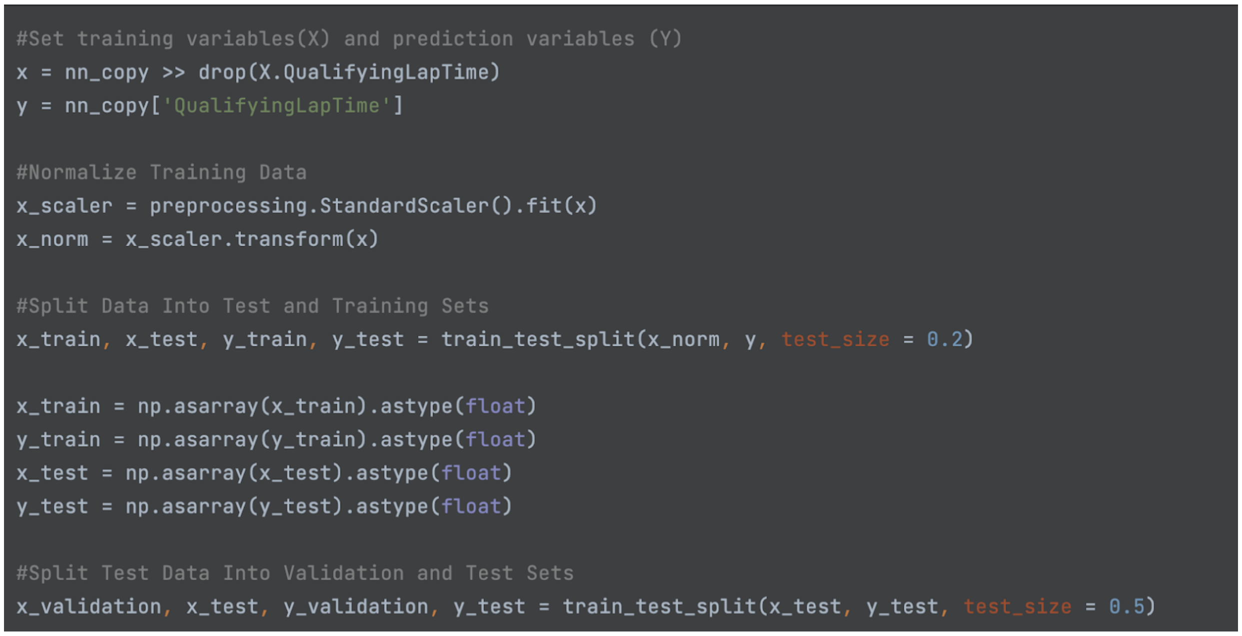
In order to make our neural network model we used Google’s machine learning package TensorFlow. This package allows us to create models without having to manually program gradient descent or backpropagation. All of these complexities are handled by the package, but the understanding of the concepts allowed us to understand the code and how to utilize the functions provided by TensorFlow. The TensorFlow documentations quickstart for beginners tutorial to model our first network after.

The first step was organizing our data in such a way that it was all numeric so that it could be inputted into a neural network. This process was difficult as Caleb did not have experience working with categorical variables. The first models were made with an integer representation for each categorical variable. For example, the Driver variable was changed to an integer from 1-22 with each number representing one of the drivers. This creates a false order in the data and does not work well in functions. Alex has more experience and helped out by creating dummy variables for all the categorical data. This means a boolean variable was created for each category. For example, 22 boolean variables were created with each one corresponding to whether or not the data was associated with a specific driver. This dropped the mean squared error of our models from around six to below one. The final code is:


```
nn_copy = masterList.copy()
nn_copy = pd.get_dummies(nn_copy, columns = ['Weekend', 'Driver', 'Compound', 'QualifyingCompound'])

#Adjust Boolean Data
nn_copy['New'] = pd.to_numeric(nn_copy['New'], errors = 'coerce')

nn_copy = nn_copy.dropna()
```

The next step is splitting the data into training data, validation data, and test data. Since the TensorFlow documentation used preloaded test and training sets, we had to figure out how to split our data with a different source. This source provided a useful sklearn function that was used in our code:

https://towardsdatascience.com/how-to-split-a-dataset-into-training-and-testing-sets-b146b1649830



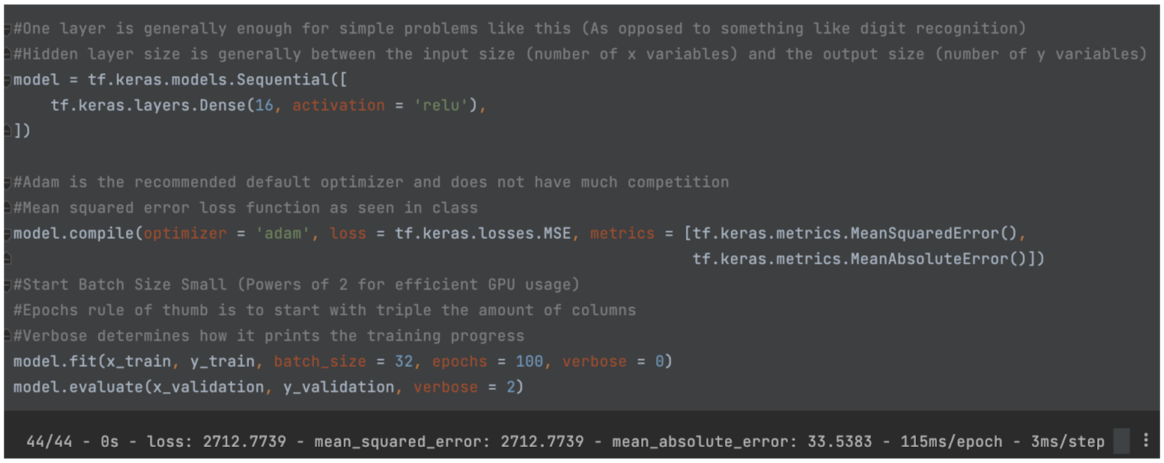
```
#Set training variables(X) and prediction variables (Y)
x = nn_copy >> drop(X.QualifyingLapTime)
y = nn_copy['QualifyingLapTime']

#Normalize Training Data
x_scaler = preprocessing.StandardScaler().fit(x)
x_norm = x_scaler.transform(x)

#Split Data Into Test and Training Sets
x_train, x_test, y_train, y_test = train_test_split(x_norm, y, test_size = 0.2)

x_train = np.asarray(x_train).astype(float)
y_train = np.asarray(y_train).astype(float)
x_test = np.asarray(x_test).astype(float)
y_test = np.asarray(y_test).astype(float)

#Split Test Data Into Validation and Test Sets
x_validation, x_test, y_validation, y_test = train_test_split(x_test, y_test, test_size = 0.5)
```

The data was also normalized during this process since the range of possible values for each variable varied drastically.

This allowed us to create our first model using the following source to determine what parameters to start with:

https://towardsdatascience.com/neural-networks-parameters-hyperparameters-and-optimization-strategies-3f0842fac0a5

Useful information from this article is commented into the code:



```
#One layer is generally enough for simple problems like this (As opposed to something like digit recognition)
#Hidden layer size is generally between the input size (number of x variables) and the output size (number of y variables)
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(16, activation = 'relu'),
])

#Adam is the recommended default optimizer and does not have much competition
#Mean squared error loss function as seen in class
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                          tf.keras.metrics.MeanAbsoluteError()])
#Start Batch Size Small (Powers of 2 for efficient GPU usage)
#Epochs rule of thumb is to start with triple the amount of columns
#Verbose determines how it prints the training progress
model.fit(x_train, y_train, batch_size = 32, epochs = 100, verbose = 0)
model.evaluate(x_validation, y_validation, verbose = 2)
```

```
44/44 - 0s - loss: 2712.7739 - mean_squared_error: 2712.7739 - mean_absolute_error: 33.5383 - 115ms/epoch - 3ms/step
```

The code for our model consists of the following functions:

tf.keras.models.Sequential(): This creates a model that matches the theory we described; each hidden layer takes the output of the previous layer and passes on its own output to the following layer. This is where the term sequential comes from.

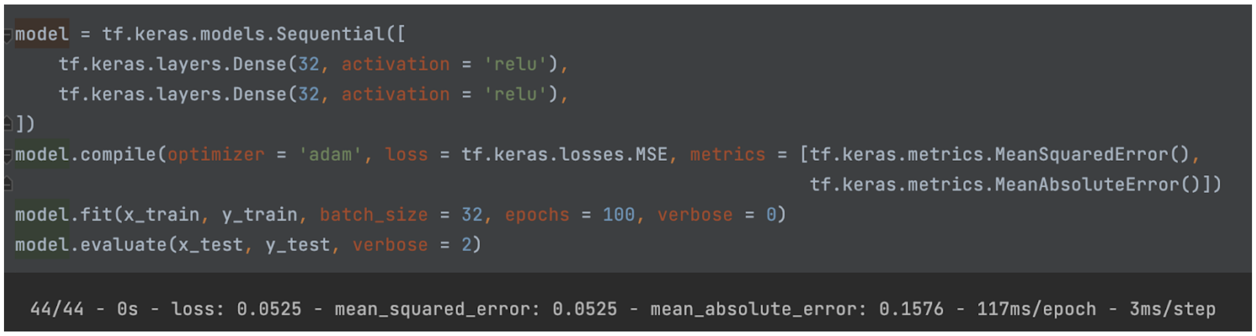
tf.keras.layers.Dense(): This function creates a Dense layer, which describes the connectivity between the layers. Dense means that every neuron receives every output from the previous layer. The integer parameter describes how many neurons in the layer and the activation is the function that the linear equation is inputted into. Relu was recommended in the 3Blue1Brown YouTube playlist so we used it in our first model.

model.compile(): The compile function describes how the model will run gradient descent. The optimizer is the default option and allows it to run more efficiently. The loss function describes the cost/error function described in the theory section. The metrics parameter simply changes what metrics are outputted during the training and evaluation processes.

model.fit(): The fit function trains the network through gradient descent and backpropagation. It takes training input and output as its first two parameters. The batch size is the number of training points the model is trained on at a time. Epochs sets how many times the network is trained on all of the data. Verbose allows the programmer to choose what output to receive during the training process.

model.evaluate(): The evaluate function tests the performance of the network on the validation set and also allows for different output formats using the verbose parameter.

Our first model did not perform well at all with a mean squared error of 2712. This essentially means it was not able to make any sort of accurate prediction. From here, the rest of the process involved modifying the parameters of the above code. This included testing different numbers of neurons, different numbers of layers, different activation functions, different batch sizes, and different numbers of epochs. The way Caleb went about optimizing these parameters was modifying one at a time and finding the optimal setting with all the others remaining constant. There are a near infinite amount of combinations so this process did not test every possible combination of parameters. Every iteration that was tested is shown on github, but the final version was as follows:



```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(32, activation = 'relu'),
    tf.keras.layers.Dense(32, activation = 'relu'),
])
model.compile(optimizer = 'adam', loss = tf.keras.losses.MSE, metrics = [tf.keras.metrics.MeanSquaredError(),
                                                                          tf.keras.metrics.MeanAbsoluteError()])
model.fit(x_train, y_train, batch_size = 32, epochs = 100, verbose = 0)
model.evaluate(x_test, y_test, verbose = 2)
```

```
44/44 - 0s - loss: 0.0525 - mean_squared_error: 0.0525 - mean_absolute_error: 0.1576 - 117ms/epoch - 3ms/step
```

This combination of parameters allowed our model to have a mean squared error of 0.05 on the test data, which it had never seen before. This indicates that it was able to predict a driver's performance extremely well.

Data Visualization

The big-picture purpose of data visualization was to gain a visual understanding and intuition for the distributions of data values for each recorded statistic as well as to evaluate the overall data quality. If the data were more uniform, plotting data would have been trivial. However, data was split across several different races, each with individual drivers. Likewise, plotting a single aggregate statistic from the compiled dataframe would not suffice to properly represent that statistic–many statistics clearly depended on and varied across race locations and individual drivers. For each statistic, data had to be segmented into races and then into drivers within races. Because there were many statistics to observe, the tree-like structure of data within each statistic necessitated many plots (in the hundreds). Such complexity rendered simple loops and dataframe filters insufficient for generating plots, and a more structured approach via data and plotting abstractions became necessary.

The first layer of abstraction was the PlotData class, with the following constructor:

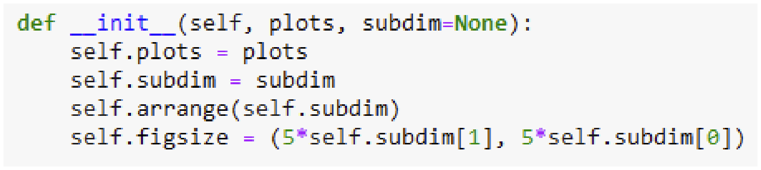
```
def __init__(self, xdata, ydata, ptype='plot', title='Figure', xlabel='x axis', ylabel='y axis', grid=True):
```

At its core, the PlotData object served as a wrapper over plottable data and matplotlib functionality, holding groups of x and y data to plot against each other on a single plot. The x and y data were expected to be lists of lists of data points so that multiple relationships could be displayed on a single plot. If the plot type required both x and y data, the constructor would check to make sure the number of both total x and y groups and the number of points within each corresponding group matched.

The other parameters were optional configurations for plot display. The ptype parameter could be set to plot, scatter, bar, or boxplot and would use the corresponding matplotlib function to display the data. In the case that the type of plot was a box plot, the ydata could be specified as

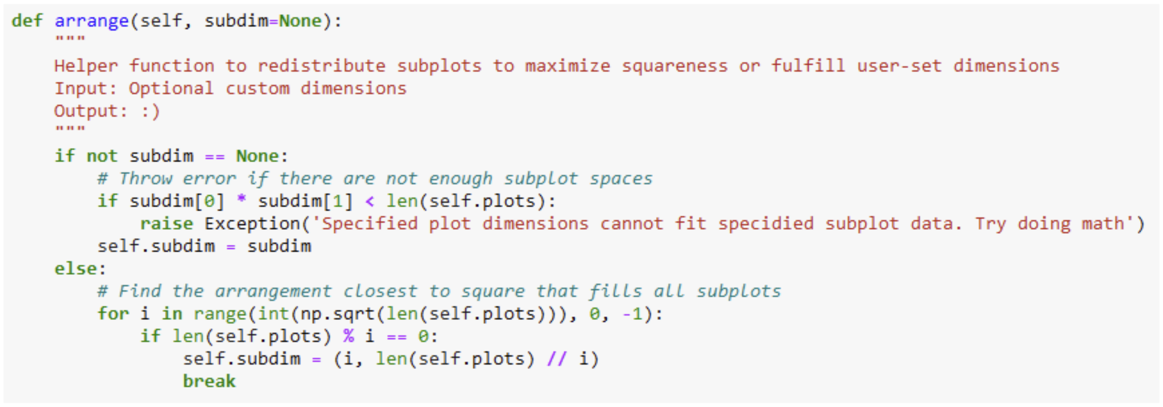
None. The other parameters are fairly self explanatory plot label and display properties. The PlotData may then use the plot function to auto-generate a plot with the provided parameters.

Another layer of abstraction was the Plotter class, which used multiple instances of PlotData to generate figures with subplots. The constructor for this class was simpler:



```python
def __init__(self, plots, subdim=None):
    self.plots = plots
    self.subdim = subdim
    self.arrange(self.subdim)
    self.figsize = (5*self.subdim[1], 5*self.subdim[0])
```

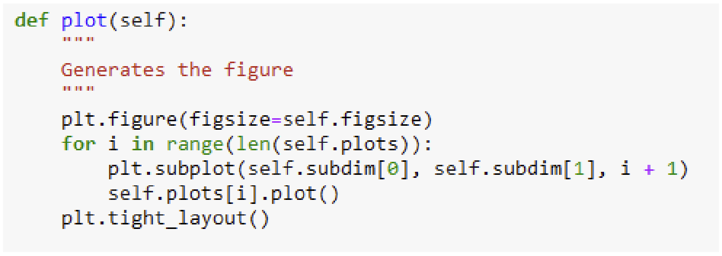
The only parameters are a required list of PlotData classes and an optional sub-dimension size for arranging subplots, entered as a tuple of 2 integers. The 2 main Plotter methods of note are plot() and arrange(). The arrange method is a helper function used for the organization of subplots, used only within the class:



```python
def arrange(self, subdim=None):
    """
    Helper function to redistribute subplots to maximize squareness or fulfill user-set dimensions
    Input: Optional custom dimensions
    Output: :)
    """
    if not subdim == None:
        # Throw error if there are not enough subplot spaces
        if subdim[0] * subdim[1] < len(self.plots):
            raise Exception('Specified plot dimensions cannot fit specidied subplot data. Try doing math')
        self.subdim = subdim
    else:
        # Find the arrangement closest to square that fills all subplots
        for i in range(int(np.sqrt(len(self.plots))), 0, -1):
            if len(self.plots) % i == 0:
                self.subdim = (i, len(self.plots) // i)
                break
```

If the subdim parameter is explicitly specified, then the function checks that the number of PlotData in the list does not exceed the number of allocatable spaces. Otherwise, if no subdim value is specified, the Plotter will automatically find the arrangement of subplots closest to a square, with the larger dimension being horizontal.

The plot function is simple and takes no parameters:

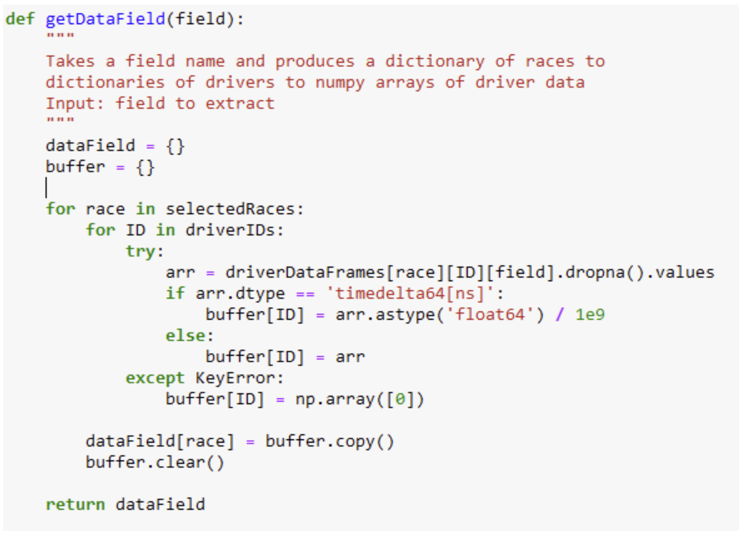


```python
def plot(self):
    """
    Generates the figure
    """
    plt.figure(figsize=self.figsize)
    for i in range(len(self.plots)):
        plt.subplot(self.subdim[0], self.subdim[1], i + 1)
        self.plots[i].plot()
    plt.tight_layout()
```

The function simply goes through each PlotData in the provided list and sequentially places it in the next subplot region, using the PlotData's own plot function. With the above 2 classes, large and complex figures can be generated with relative ease.

However, the issue of visualizing data is not yet completely solved, as the input format of the PlotData class must be a list of lists. The next section of code is responsible for converting a copy of the master dataframe into a dictionary from races to dictionaries from driver ID to a dataframe for a single driver. These nested dictionaries mimic the aforementioned tree-like structure of the data.

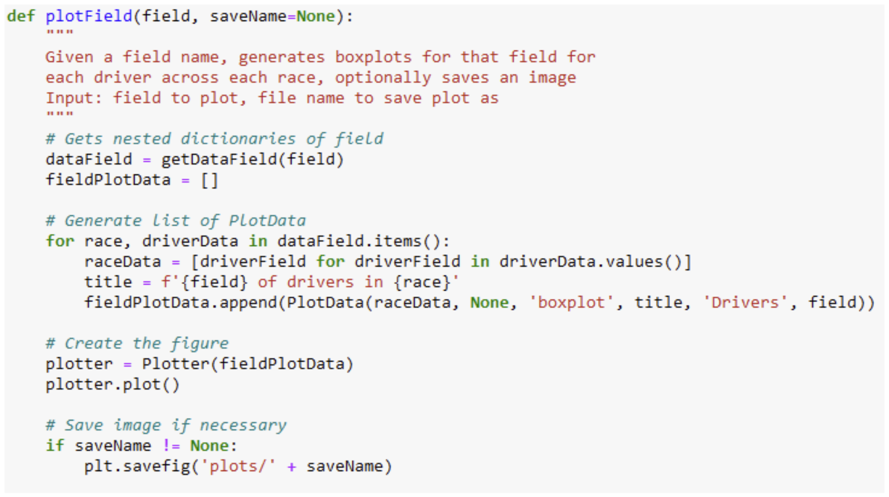
To get lists for a specific statistic, the getDataField() function is used:



```python
def getDataField(field):
    """
    Takes a field name and produces a dictionary of races to
    dictionaries of drivers to numpy arrays of driver data
    Input: field to extract
    """
    dataField = {}
    buffer = {}

    for race in selectedRaces:
        for ID in driverIDs:
            try:
                arr = driverDataFrames[race][ID][field].dropna().values
                if arr.dtype == 'timedelta64[ns]':
                    buffer[ID] = arr.astype('float64') / 1e9
                else:
                    buffer[ID] = arr
            except KeyError:
                buffer[ID] = np.array([0])

        dataField[race] = buffer.copy()
        buffer.clear()

    return dataField
```

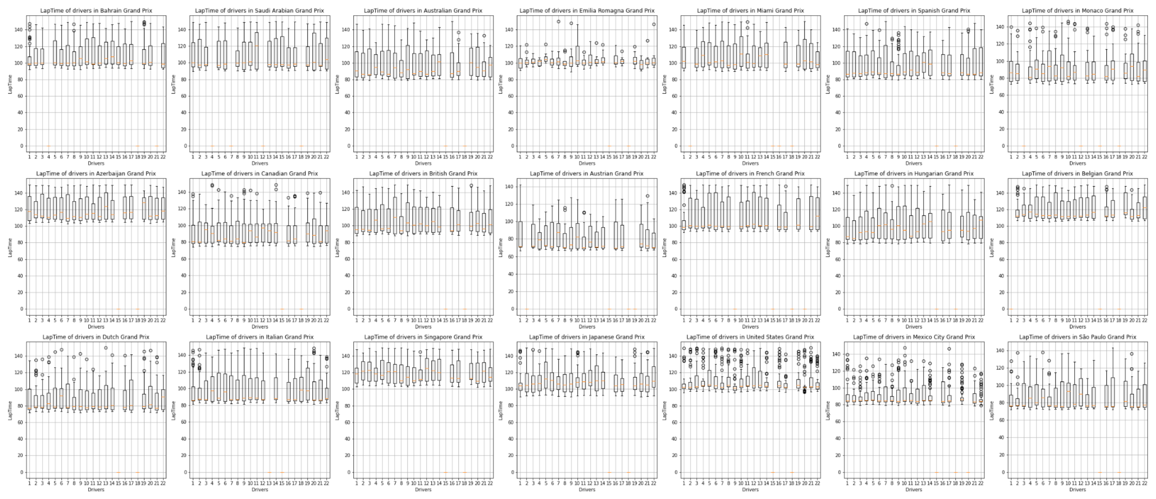
This function takes the name of a statistic and creates a nested dictionary like described above, instead ending at a list of data entries for that statistic rather than the whole driver dataframe.

This more specific nested dictionary is applied in the next step of automating the creation of PlotData classes, with the plotField() function:



```python
def plotField(field, saveName=None):
    """
    Given a field name, generates boxplots for that field for
    each driver across each race, optionally saves an image
    Input: field to plot, file name to save plot as
    """
    # Gets nested dictionaries of field
    dataField = getDataField(field)
    fieldPlotData = []

    # Generate list of PlotData
    for race, driverData in dataField.items():
        raceData = [driverField for driverField in driverData.values()]
        title = f'{field} of drivers in {race}'
        fieldPlotData.append(PlotData(raceData, None, 'boxplot', title, 'Drivers', field))

    # Create the figure
    plotter = Plotter(fieldPlotData)
    plotter.plot()

    # Save image if necessary
    if saveName != None:
        plt.savefig('plots/' + saveName)
```

Like getDataField(), plotField() takes the name of a statistic and creates PlotData instances for every race, with each PlotData instance responsible for box plots of all 22 drivers within the race. All PlotData instances are added to a single list, which is taken in by a Plotter class which generates the whole figure. As an example, this is the figure for the LapTime statistic:



To plot the data for every numeric statistic, in the next few notebook cells, a list of statistic names is defined and a loop executes plotField() for each driver statistic, successfully generating visualizations for all variable distributions.

Outcomes

Final Results

With all said and done, we learned about the inner architecture and process of using neural networks and were able to create a model that could predict lap-times to a precise degree (most qualifying lap time differences between drivers are outside the MSE of 0.05 so the grid order can be reasonably predicted).

Areas for Improvement

One area for improvement is to include more data. FastF1 presents much more data than just the timing and weather, namely data about the cars i.e. their RPMs, throttle inputs, etc. If there was some way to include that data then our neural network would have a lot more to work with. In addition to this, we were omitting data from before the 2022 season because the car specs were changed then. For future seasons, we will be able to incorporate their data since the cars will not change as much.

References

Fast F1 Documentation: https://theoehrly.github.io/Fast-F1/

Dfply Documentation: https://github.com/kieferk/dfply

Pandas Documentation: https://pandas.pydata.org/docs/

NumPy Documentation: https://numpy.org/doc/stable/

Matplotlib Documentation: https://matplotlib.org/stable/index.html

3Blue1Brown Neural Network Videos:

https://youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

TensorFlow Documentation: https://www.tensorflow.org

TensorFlow 2 quickstart for beginners: https://www.tensorflow.org/tutorials/quickstart/beginner