FourthBrain

⛽ Fuel efficiency Prediction

Provided with the classic [Auto MPG]() dataset, we will predict the **fuel efficiency** of the late-1970s and early 1980s automobiles, leveraging features such as cylinders, displacement, horsepower, weight, etc.

It is a very small dataset and there are only a few features. We will first build a linear model and a neural network, evaluate their performances, and then leverage an auto-machine learning (AutoML) library called [TPOT]() to see how it can be used to search over many ML model acchitectures.

📚 Learning Objectives

By the end of this session, you will be able to

- understand the core building blocks of a neural network
- understand what dense and activation layers do
- build, train, and evaluate neural networks
- perform AutoML to search for optimal tree-based pipeline for a regression task

Note: [State of Data Science and Machine Learning 2021]() by Kaggle shows that the most commonly used algorithms were linear and logtistic regressions, followed closely by decision trees, random forests, and gradient boosting machines (are you surprised?). Multilayer perceptron, or artificial neural networks are not yet the popular tools for tabular/structured data; see more technical reasons in papers: [Deep Neural Networks and Tabular Data: A Survey](), [Tabular Data: Deep Learning is Not All You Need](). For this assignment, the main purpose is for you to get familiar with the basic building blocks in constructing neural networks before we dive into more specialized neural network architectures.

IMPORTANT

You only need to run the following cells if you're completing the assignment in Google Collab. If you've already installed these libraries locally, you can skip installing these libraries.

```
# Connect colab to your Google Drive

from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
!pip install -q seaborn ## Use seaborn for pairplot
!pip install -q tpot  # Use TPOT for automl
```

```
     |████████████████████████████████| 87 kB 3.3 MB/s
     |████████████████████████████████| 192.9 MB 70 kB/s
     |████████████████████████████████| 139 kB 56.0 MB/s
  Building wheel for stopit (setup.py) ... done
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Make NumPy printouts easier to read.
suppress=True)
```

Saved successfully!

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

print(tf.__version__)
```

```
2.8.2
```

Task 1 - Data: Auto MPG dataset

1. The dataset is available from the [UCI Machine Learning Repository](). First download and import the dataset using `pandas`:

```
url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.dat
column_names = [
  'MPG', 'Cylinders', 'Displacement', 'Horsepower', 'Weight',
  'Acceleration', 'Model Year', 'Origin'
```

```
]

dataset = pd.read_csv(url, names=column_names, na_values='?',
                      comment='\t', sep=' ', skipinitialspace=True)
```

```
dataset.tail()
```

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year
393	27.0	4	140.0	86.0	2790.0	15.6	82
394	44.0	4	97.0	52.0	2130.0	24.6	82
395	32.0	4	135.0	84.0	2295.0	11.6	82
396	28.0	4	120.0	79.0	2625.0	18.6	82
397	31.0	4	119.0	82.0	2720.0	19.4	82

2. The dataset contains a few unknown values, we drop those rows to keep this initial tutorial simple. Use `pd.DataFrame.dropna()`:

```
dataset = dataset.dropna()
```

Saved successfully!

rical, not numeric. So the next step is to one-hot encode the values in the column with pd.get_dummies.

```
dataset['Origin'] = dataset['Origin'].replace({1: 'USA', 2: 'Europe', 3: 'Japan'})
```

```
dataset = pd.get_dummies(dataset, columns=['Origin'], prefix='', prefix_sep='')
dataset.tail()
```

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year
393	27.0	4	140.0	86.0	2790.0	15.6	82
394	44.0	4	97.0	52.0	2130.0	24.6	82
395	32.0	4	135.0	84.0	2295.0	11.6	82
396	28.0	4	120.0	79.0	2625.0	18.6	82
397	31.0	4	119.0	82.0	2720.0	19.4	82

4. Split the data into training and test sets. To reduce the module importing overhead, instead of `sklearn.model_selection.train_test_split()`, use `pd.DataFrame.sample()` to save 80% of the data aside to `train_dataset`, set the random state to be 0 for reproducibility.

   Then use `pd.DataFrame.drop()` to obtain the `test_dataset`.

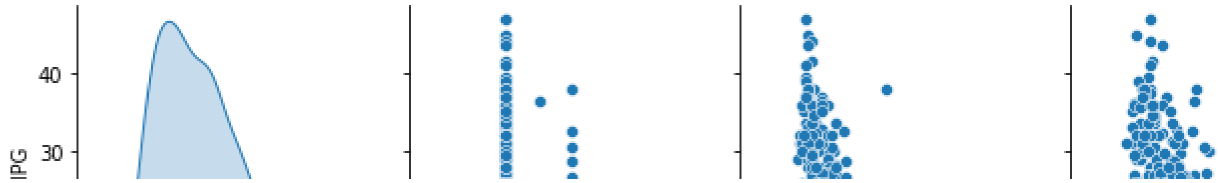
```
train_dataset = dataset.sample(frac=0.8, random_state=0)
test_dataset = dataset.drop(train_dataset.index)
```

5. Review the pairwise relationships of a few pairs of columns from the training set.

   The top row suggests that the fuel efficiency (MPG) is a function of all the other parameters. The other rows indicate they are functions of each other.

```
sns.pairplot(train_dataset[['MPG', 'Cylinders', 'Displacement', 'Weight']], diag_kind=
```

Saved successfully!



Let's also check the overall statistics. Note how each feature covers a very different range:

```
train_dataset.describe().transpose()
```

	count	mean	std	min	25%	50%	75%	max
MPG	314.0	23.310510	7.728652	10.0	17.00	22.0	28.95	46.6
Cylinders	314.0	5.477707	1.699788	3.0	4.00	4.0	8.00	8.0
Displacement	314.0	195.318471	104.331589	68.0	105.50	151.0	265.75	455.0
Horsepower	314.0	104.869427	38.096214	46.0	76.25	94.5	128.00	225.0
Weight	314.0	2990.251592	843.898596	1649.0	2256.50	2822.5	3608.00	5140.0
Acceleration	314.0	15.559236	2.789230	8.0	13.80	15.5	17.20	24.8
Model Year	314.0	75.898089	3.675642	70.0	73.00	76.0	79.00	82.0
Europe	314.0	0.178344	0.383413	0.0	0.00	0.0	0.00	1.0
Japan	314.0	0.197452	0.398712	0.0	0.00	0.0	0.00	1.0
[illegible]	[illegible]	04	0.485101	0.0	0.00	1.0	1.00	1.0

Saved successfully!

5. Split features from labels

   Separate the target value—the "label"—from the features. This label is the value that you will train the model to predict.

```
train_features = train_dataset.iloc[:, 1:]
test_features = test_dataset.iloc[:, 1:]

train_labels = train_dataset['MPG']
test_labels = test_dataset['MPG']
```

```
train_features
```

	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	Europe
146	4	90.0	75.0	2125.0	14.5	74	(
282	4	140.0	88.0	2890.0	17.3	79	(
69	8	350.0	160.0	4456.0	13.5	72	(
378	4	105.0	63.0	2125.0	14.7	82	(
331	4	97.0	67.0	2145.0	18.0	80	(
...	...	...	...	...	...	...	..
281	6	200.0	85.0	2990.0	18.2	79	(
229	8	400.0	180.0	4220.0	11.1	77	(
150	4	108.0	93.0	2391.0	15.5	74	(

Task 2 - Normalization Layer

314 rows × 9 columns

It is good practice to normalize features that use different scales and ranges. Although a model *might* converge without feature normalization, normalization makes training much more stable.

Similar to scikit-learn, tensorflow.keras offers a list of [preprocessing layers]() so that you can build and export models that are truly end-to-end.

Saved successfully!

1. The Normalization layer (`tf.keras.layers.Normalization` is a clean and simple way to add feature normalization into your model. The first step is to create the layer:

```
normalizer =  tf.keras.layers.Normalization(axis=-1)
```

2. Then, fit the state of the preprocessing layer to the data by calling `Normalization.adapt`:

```
normalizer.adapt(np.array(train_features))
```

We can see the feature mean and variance are stored in the layer:

```
print(f'feature mean: {normalizer.mean.numpy().squeeze()}\n')
print(f'feature variance: {normalizer.variance.numpy().squeeze()}')
```

```
feature mean: [   5.478  195.318  104.869 2990.252   15.559   75.898    0.178
    0.624]
```

```
feature variance: [     2.88    10850.413    1446.699 709896.9          7.755      13
       0.147        0.158        0.235]
```

When the layer is called, it returns the input data, with each feature independently normalized:

```
first = np.array(train_features[:1])

with np.printoptions(precision=2, suppress=True):
  print('First example:', first)
  print()
  print('Normalized:', normalizer(first).numpy())
```

```
First example: [[   4.     90.     75.   2125.     14.5    74.      0.      0.      1. ]

Normalized: [[-0.87 -1.01 -0.79 -1.03 -0.38 -0.52 -0.47 -0.5   0.78]]
```

Task 3 - Linear regression 📈

Before building a deep neural network model, start with linear regression using all the features.

Saved successfully!

...ally starts by defining the model architecture. Use a ...represents a sequence of steps.

There are two steps in this multivariate linear regression model:

- Normalize all the input features using the `tf.keras.layers.Normalization` preprocessing layer. You have defined this ealier as `normalizer`.
- Apply a linear transformation ($y = mx + b$ where m is a matrix and b is a vector.) to produce 1 output using a linear layer (`tf.keras.layers.Dense`).

The number of *inputs* can either be set by the `input_shape` argument, or automatically when the model is run for the first time.

1. Build the Keras Sequential model:

```
linear_model = tf.keras.Sequential([
    normalizer,
    tf.keras.layers.Dense(1)
])
```

```
linear_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization (Normalizatio  (None, 9)                19
 n)

 dense (Dense)               (None, 1)                 10

=================================================================
Total params: 29
Trainable params: 10
Non-trainable params: 19
_________________________________________________________________
```

2. This model will predict `'MPG'` from all features in `train_features`. Run the untrained model on the first 10 data points / rows using `Model.predict()`. The output won't be good, but notice that it has the expected shape of `(10, 1)`:

```
linear_model.predict(np.array(train_features)[0:10])
```

```
array([[-1.623],
       [-0.886],
       [ 0.931],
```



```
       [-0.795],
       [ 1.945],
       [-0.384],
       [-1.205]], dtype=float32)
```

3. When you call the model, its weight matrices will be built—check that the `kernel` weights (the m in $y = mx + b$) have a shape of (9, 1):

```
linear_model.layers[1].kernel
```

```
<tf.Variable 'dense/kernel:0' shape=(9, 1) dtype=float32, numpy=
array([[ 0.138],
       [ 0.758],
       [-0.613],
       [ 0.622],
       [-0.15 ],
       [ 0.124],
       [ 0.767],
       [ 0.14 ],
       [-0.187]], dtype=float32)>
```

4. Once the model is built, configure the training procedure using the Keras `Model.compile` method. The most important arguments to compile are the `loss` and the `optimizer`, since these define what will be optimized and how (using the `tf.keras.optimizers.Adam`).

   Here's a list of built-in loss functions in `tf.keras.losses`. For regression tasks, common loss functions include mean squared error (MSE) and mean absolute error (MAE). Here, MAE is preferred such that the model is more robust against outliers.

   For optimizers, gradient descent (check this video Gradient Descent, Step-by-Step for a refresher) is the preferred way to optimize neural networks and many other machine learning algorithms. Read an overview of graident descent optimizer algorithms for several popular gradient descent algorithms. Here, we use the popular `tf.keras.optimizers.Adam`, and set the learning rate at 0.1 for faster learning.

```
linear_model.compile(
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.1),
    loss = tf.keras.losses.MeanAbsoluteError()
    )
```

5. Use Keras `Model.fit` to execute the training for 100 epochs, set the verbose to 0 to suppress logging and keep 20% of the data for validation:

Saved successfully!

```
%%time

# history = linear_model.fit(train_features.astype("float32") , train_labels.astype("f

history = linear_model.fit(np.array(train_features), np.array(train_labels) , epochs=1
```

```
CPU times: user 4.01 s, sys: 220 ms, total: 4.23 s
Wall time: 3.99 s
```

6. Visualize the model's training progress using the stats stored in the `history` object:

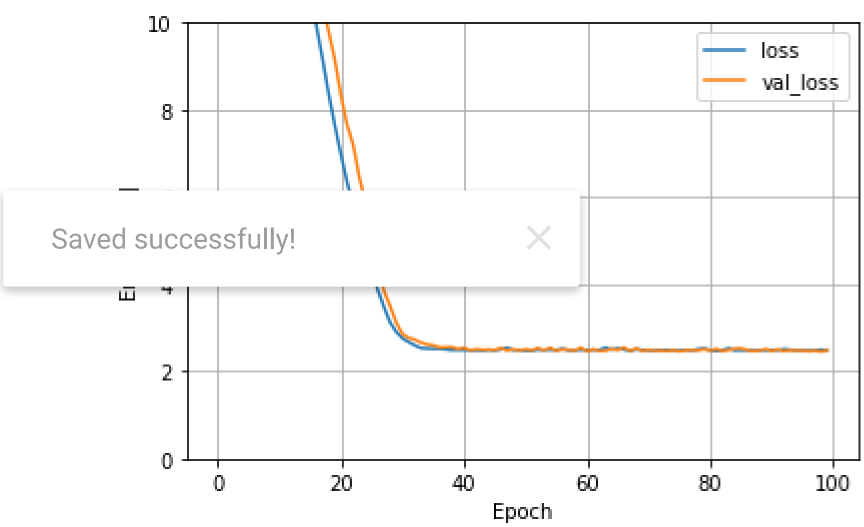
```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist.tail()
```

	loss	val_loss	epoch
95	2.473868	2.473518	95
96	2.471930	2.453896	96

```
def plot_loss(history):
  plt.plot(history.history['loss'], label='loss')
  plt.plot(history.history['val_loss'], label='val_loss')
  plt.ylim([0, 10])
  plt.xlabel('Epoch')
  plt.ylabel('Error [MPG]')
  plt.legend()
  plt.grid(True)
```

Use `plot_loss(history)` provided to visualize the progression in loss function for training and validation data sets.

```
plot_loss(history)
```



7. Collect the results on the test set for later using [`Model.evaluate()`]()

```
test_results = {}

test_results['linear_model'] = linear_model.evaluate(np.array(test_features), np.array
```

```
3/3 [==============================] - 0s 4ms/step - loss: 2.5147
```

```
test_results
```

```
{'linear_model': 2.51470947265625}
```

Task 4 - Regression with a deep neural network (DNN)

You just implemented a linear model for multiple inputs. Now, you are ready to implement multiple-input DNN models.

The code is very similar except the model is expanded to include some "hidden" **non-linear** layers. The name "hidden" here just means not directly connected to the inputs or outputs.

- The normalization layer, as before (with `normalizer` for a multiple-input model).
- Two hidden, non-linear, [`Dense`](#) layers with the ReLU (`relu`) activation function nonlinearity. One way is to set parameter `activation` inside `Dense` Set the number of neurons at each layer to be 64.
- A linear `Dense` single-output layer.

1. Include the model and `compile` method in the `build_and_compile_model` function below.

```
def build_and_compile_model(norm):
  model = keras.Sequential([
      norm,
                  ='relu'),
                  ='relu'),

  ])
  model.compile(loss='mean_absolute_error',
                optimizer=tf.keras.optimizers.Adam())
  return model
```

Saved successfully!

2. Create a DNN model with `normalizer` (defined earlier) as the normalization layer:

```
dnn_model =  build_and_compile_model(normalizer)
```

3. Inspect the model using `Model.summary()` . This model has quite a few more trainable parameters than the linear models:

```
dnn_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
```

```
 normalization (Normalizatio  (None, 9)                19
 n)

 dense_1 (Dense)             (None, 64)                640

 dense_2 (Dense)             (None, 64)                4160

 dense_3 (Dense)             (None, 1)                 65

=================================================================
Total params: 4,884
Trainable params: 4,865
Non-trainable params: 19
_________________________________________________________________
```

4. Train the model with Keras `Model.fit`:

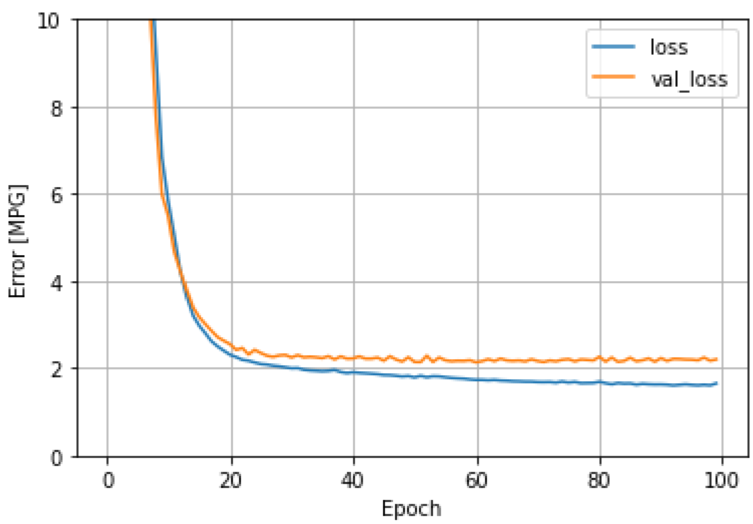
```
%%time
history = dnn_model.fit(
    train_features,
    train_labels,
    validation_split=0.2,
    verbose=0, epochs=100)
```

```
CPU times: user 4.34 s, sys: 233 ms, total: 4.58 s
Wall time: 5.63 s
```

Saved successfully!

gress using the stats stored in the history object.

```
plot_loss(history)
```



Do you think the DNN model is overfitting? What gives away?

YOUR ANSWER HERE

6. Let's save the results for later comparison.

```
test_results['dnn_model'] = dnn_model.evaluate(test_features, test_labels, verbose=0)
```

Task 5 - Make predictions 🔮

1. Since both models have been trained, we can review their test set performance:

```
pd.DataFrame(test_results, index=['Mean absolute error [MPG]']).T
```

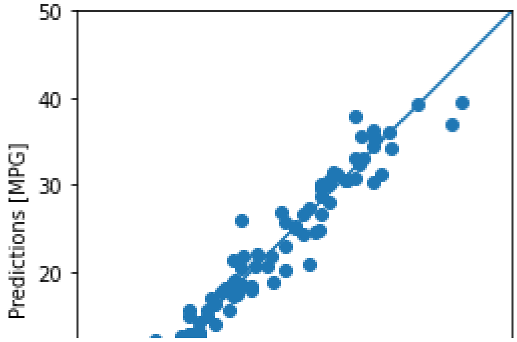
	Mean absolute error [MPG]
linear_model	2.514709
dnn_model	1.654161

These results match the validation error observed during training.

Saved successfully!

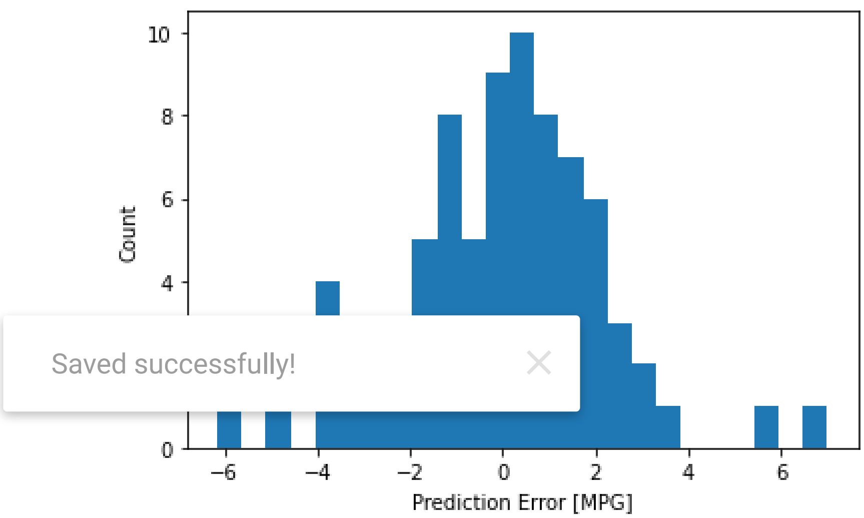
2. We can now make predictions with the `dnn_model` on the test set using Keras `Model.predict` and review the loss. Use `.flatten()`.

```
test_predictions = dnn_model.predict(test_features).flatten()

a = plt.axes(aspect='equal')
plt.scatter(test_labels, test_predictions)
plt.xlabel('True Values [MPG]')
plt.ylabel('Predictions [MPG]')
lims = [0, 50]
plt.xlim(lims)
plt.ylim(lims)
_ = plt.plot(lims, lims)
```



3. It appears that the model predicts reasonably well. Now, check the error distribution:

```
error = test_predictions - test_labels
plt.hist(error, bins=25)
plt.xlabel('Prediction Error [MPG]')
_ = plt.ylabel('Count')
```



4. Save it for later use with `Model.save`:

```
dnn_model.save('dnn_model')
```

5. Reload the model with `Model.load_model`; it gives identical output:

```
reloaded = tf.keras.models.load_model('dnn_model')

test_results['reloaded'] = reloaded.evaluate(
    test_features, test_labels, verbose=0)
```

```
pd.DataFrame(test_results, index=['Mean absolute error [MPG]']).T
```

	Mean absolute error [MPG]
linear_model	2.514709
dnn_model	1.654161
reloaded	1.654161

Task 6 - Nonlinearity

We mentioned that the `relu` activation function introduce non-linearity; let's visualize it. Yet there are six numerical features and 1 categorical features, it is impossible to plot all the dimensions on a 2D plot; we need to simplify/isolate it.

Note: in this task, code is provided; the focus in on understanding.

1. We focus on the relationship between feature `Displacement` and target `MPG`.

   To do so, create a new dataset of the same size as `train_features`, but all other features are set at their median values; then set the `Displacement` between 0 and 500.

```
fake = np.outer(np.ones(train_features.shape[0]), train_features.median())
                     = train_features.columns)
                    , 500, train_features.shape[0])
```

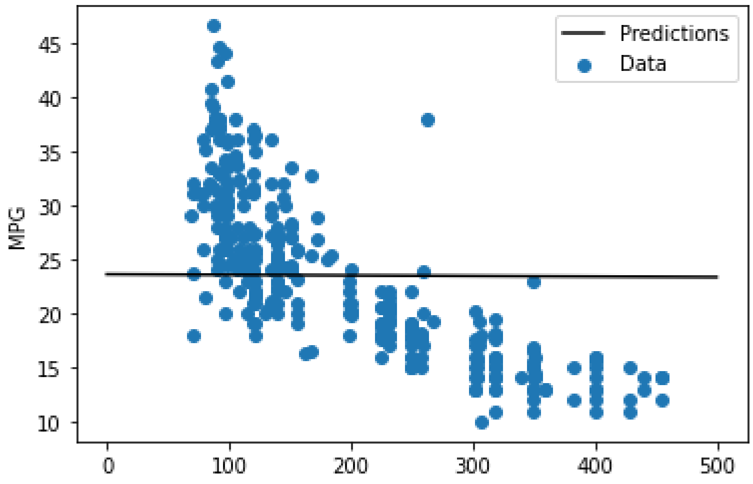
Saved successfully!

2. Create a plotting function to a) visualize real values between `Displacement` and `MPG` from the training dataset in scatter plot b) overlay the predicted MPG from Displacement varying from 0 to 500, but holding all other features constant.

```
def plot_displacement(x, y):
  plt.scatter(train_features['Displacement'], train_labels, label='Data')
  plt.plot(x, y, color='k', label='Predictions')
  plt.xlabel('Displacement')
  plt.ylabel('MPG')
  plt.legend()
```

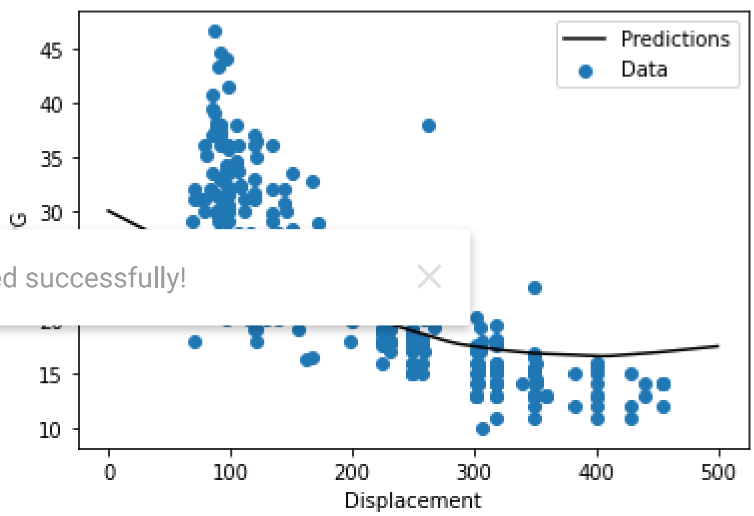
3. Visualize predicted MPG using the linear model.

```
plot_displacement(fake.Displacement, linear_model(fake))
```



4. Visualize predicted MPG using the neural network model. Do you see an improvement/non-linearity from the linear model?

```
plot_displacement(fake.Displacement, dnn_model.predict(fake))
```



Saved successfully!

5. What are the other activation functions? Check the list of [activations](#).

   Optional. Modify the DNN model with a different activation function, and fit it on the data; does it perform better?

6. Overfitting is a common problem for DNN models, how should we deal with it? Check [Regularizers](#) on tf.keras. Any other techiniques that are invented for neural networks?

Task 7 - AutoML with TPOT 🫖

1. Instantiate and train a TPOT auto-ML regressor.

   The parameters are set fairly arbitrarily (if time permits, you shall experiment with different sets of parameters after reading what each parameter does). Use these parameter values:

   `generations`: 10

   `population_size`: 40

   `scoring`: negative mean absolute error; read more in scoring functions in TPOT

   `verbosity`: 2 (so you can see each generation's performance)

   The final line with create a Python script `tpot_products_pipeline.py` with the code to create the optimal model found by TPOT.

```
%%time
from tpot import TPOTRegressor
tpot = TPOTRegressor(generations=10,
                     population_size=40,
                     scoring='neg_median_absolute_error',
                     verbosity=2,
                     random_state=42)
tpot.fit(train_features, train_labels)
print(f"Tpop score on test data: {tpot.score(test_features, test_labels):.2f}")
tpot.export('tpot_mpg_pipeline.py')
```

Saved successfully!

```
Generation 1 - Current best internal CV score: -1.3989326355090712
```

2. Examine the model pipeline that TPOT regressor offers. If you see any model, function, or class that are not familiar, look them up!

   Note: There is randomness to the way the TPOT searches, so it's possible you won't have exactly the same result as your classmate.

```
Generation 5 - Current best internal CV score: -1.247981910198969
```

```
cat tpot_mpg_pipeline.py
```

```
import numpy as np
import pandas as pd
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.linear_model import RidgeCV
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline, make_union
from tpot.builtins import StackingEstimator
from tpot.export_utils import set_param_recursive

# NOTE: Make sure that the outcome column is labeled 'target' in the data file
tpot_data = pd.read_csv('PATH/TO/DATA/FILE', sep='COLUMN_SEPARATOR', dtype=np.fl
features = tpot_data.drop('target', axis=1)
training_features, testing_features, training_target, testing_target = \
            train_test_split(features, tpot_data['target'], random_state=42)

# Average CV score on the training set was: -1.247981910198969
                              eline(
                              tor=ExtraTreesRegressor(bootstrap=True, max_features
)
# Fix random state for all the steps in exported pipeline
set_param_recursive(exported_pipeline.steps, 'random_state', 42)

exported_pipeline.fit(training_features, training_target)
results = exported_pipeline.predict(testing_features)
```

Saved successfully!

3. Optional: Take the appropriate lines (e.g., updating path to data and the variable names) from `tpot_mpg_pipeline.py` to build a model on our training set and make predictions on the test set. Save the predictions as `y_pred`, and compute appropriate evaluation metric. You may find that for this simple data set, the nueral network we built outperforms the tree-based model, yet note it is not a conclusion that we can be generalized for all tabular data.

Task 8 - Model Explainability

Last week, we introduced model explainability with SHAP and will continue to incorporate it as part our model output this week. You can use the [Kernal Explainer]() for explainability of both the Neural Networks and the TPOT classifier.

NOTE: If you're using Collab to complete this assignment, please run the following cell - otherwise you can skip it.

```
!pip install -q shap
```

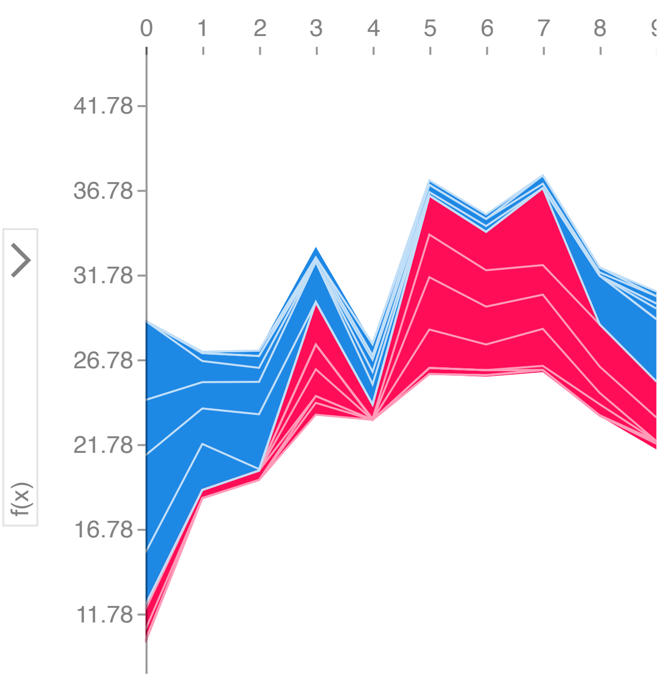
```
569 kB 5.3 MB/s
```

```
import shap

def f(X):
    return dnn_model.predict(X).flatten()

def custom_masker(mask, x):
    return (x * mask).reshape(1,len(x))

shap.initjs()
explainer = shap.KernelExplainer(f, train_features.iloc[:10, :])
shap_values = explainer.shap_values(train_features.iloc[:10, :])
shap.force_plot(explainer.expected_value, shap_values, train_features.iloc[:50, :])
```

Saved successfully!

100% 10/10 [00:04<00:00, 1.74it/s]

sample order by similarity



Task 9 - Taking it to the Next Level! 📶

Let's take our models and make a model comparison demo like we did last week, but this time you're taking the lead!

1. Save your training dataset as a CSV file so that it can be used in the Streamlit app.
2. Build a results DataFrame and save it as a CSV so that it can be used in the Streamlit app.
3. In Tab 1 - Raw Data:

- Display your training dataset in a Streamlit DataFrame (`st.DataFrame`).
- Build 1-2 interactive Plotly visualizations that explore the dataset (correlations, scatterplot, etc.)

2. In Tab 2 - Model Results:

- Display your performance metrics appropriately using 2-3 metrics for model comparison.

3. In Tab 3 - Model Explainability:

- Make local and global explainability plots to compare two models at a time side-by-side.

Here is a good example if how to create some different explainability plots using Plotly.

Saved successfully!

- Tensorflow playground for an interactive experience to understand how nueral networkds work.
- An Introduction to Deep Learning for Tabular Data covers embeddings for categorical variables.
- Imbalanced classification: credit card fraud detection demonstrates using `class_weight` to handle imbalanced classification problems.

Acknowledgement and Copyright

Acknowledgement

This notebook is adapted from tensorflow/keras tuorial - regression

✓ 4s completed at 1:26 PM

Saved successfully!