Open in Colab

T81-558: Applications of Deep Neural Networks

Module 3: Introduction to TensorFlow

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 3 Material

- Part 3.1: Deep Learning and Neural Network Introduction [Video] [Notebook]
- Part 3.2: Introduction to Tensorflow and Keras [Video] [Notebook]
- **Part 3.3: Saving and Loading a Keras Neural Network** [Video] [Notebook]
- Part 3.4: Early Stopping in Keras to Prevent Overfitting [Video] [Notebook]
- Part 3.5: Extracting Weights and Manual Calculation [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow. Running the following code will map your GDrive to `/content/drive`.

In [1]:
```
try:
    from google.colab import drive
    drive.mount('/content/drive', force_remount=True)
    COLAB = True
    print("Note: using Google CoLab")
    %tensorflow_version 2.x
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

```
Mounted at /content/drive
Note: using Google CoLab
```

Part 3.3: Saving and Loading a Keras Neural Network

Complex neural networks will take a long time to fit/train. It is helpful to be able to save these neural networks so that you can reload them later. A reloaded neural network will not require retraining. Keras provides three formats for neural network saving.

- **JSON** - Stores the neural network structure (no weights) in the JSON file format.
- **HDF5** - Stores the complete neural network (with weights) in the HDF5 file format. Do not confuse HDF5 with HDFS. They are different. We do not use HDFS in this class.

Usually, you will want to save in HDF5.

In [2]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import pandas as pd
import io
import os
import requests
import numpy as np
from sklearn import metrics

save_path = "."

df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/auto-mpg.csv",
    na_values=['NA', '?'])

cars = df['name']

# Handle missing value
df['horsepower'] = df['horsepower'].fillna(df['horsepower'].median())

# Pandas to Numpy
x = df[['cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']].values
y = df['mpg'].values # regression

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(x,y,verbose=2,epochs=100)

# Predict
pred = model.predict(x)

# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print(f"Before save score (RMSE): {score}")
```

```
# save neural network structure to JSON (no weights)
model_json = model.to_json()
with open(os.path.join(save_path,"network.json"), "w") as json_file:
    json_file.write(model_json)

# save entire network to HDF5 (save everything, suggested)
model.save(os.path.join(save_path,"network.h5"))
```

```
Epoch 1/100
13/13 - 1s - loss: 223035.4531 - 729ms/epoch - 56ms/step
Epoch 2/100
13/13 - 0s - loss: 94454.9609 - 26ms/epoch - 2ms/step
Epoch 3/100
13/13 - 0s - loss: 31163.9199 - 26ms/epoch - 2ms/step
Epoch 4/100
13/13 - 0s - loss: 6590.2344 - 24ms/epoch - 2ms/step
Epoch 5/100
13/13 - 0s - loss: 692.0208 - 23ms/epoch - 2ms/step
Epoch 6/100
13/13 - 0s - loss: 110.7149 - 23ms/epoch - 2ms/step
Epoch 7/100
13/13 - 0s - loss: 154.5042 - 22ms/epoch - 2ms/step
Epoch 8/100
13/13 - 0s - loss: 118.5529 - 29ms/epoch - 2ms/step
Epoch 9/100
13/13 - 0s - loss: 91.5691 - 27ms/epoch - 2ms/step
Epoch 10/100
13/13 - 0s - loss: 85.7397 - 24ms/epoch - 2ms/step
Epoch 11/100
13/13 - 0s - loss: 85.6981 - 23ms/epoch - 2ms/step
Epoch 12/100
13/13 - 0s - loss: 85.2837 - 24ms/epoch - 2ms/step
Epoch 13/100
13/13 - 0s - loss: 84.9037 - 23ms/epoch - 2ms/step
Epoch 14/100
13/13 - 0s - loss: 84.6506 - 30ms/epoch - 2ms/step
Epoch 15/100
13/13 - 0s - loss: 84.4048 - 26ms/epoch - 2ms/step
Epoch 16/100
13/13 - 0s - loss: 84.1072 - 24ms/epoch - 2ms/step
Epoch 17/100
13/13 - 0s - loss: 83.9168 - 23ms/epoch - 2ms/step
Epoch 18/100
13/13 - 0s - loss: 83.7391 - 24ms/epoch - 2ms/step
Epoch 19/100
13/13 - 0s - loss: 83.1922 - 21ms/epoch - 2ms/step
Epoch 20/100
13/13 - 0s - loss: 82.9178 - 27ms/epoch - 2ms/step
Epoch 21/100
13/13 - 0s - loss: 82.5835 - 28ms/epoch - 2ms/step
Epoch 22/100
13/13 - 0s - loss: 82.2728 - 24ms/epoch - 2ms/step
Epoch 23/100
13/13 - 0s - loss: 81.9899 - 24ms/epoch - 2ms/step
Epoch 24/100
13/13 - 0s - loss: 81.7262 - 23ms/epoch - 2ms/step
Epoch 25/100
13/13 - 0s - loss: 81.2958 - 26ms/epoch - 2ms/step
Epoch 26/100
13/13 - 0s - loss: 80.9488 - 30ms/epoch - 2ms/step
Epoch 27/100
13/13 - 0s - loss: 80.5811 - 33ms/epoch - 3ms/step
Epoch 28/100
13/13 - 0s - loss: 80.3213 - 25ms/epoch - 2ms/step
```

```
Epoch 29/100
13/13 - 0s - loss: 79.8659 - 27ms/epoch - 2ms/step
Epoch 30/100
13/13 - 0s - loss: 79.5628 - 24ms/epoch - 2ms/step
Epoch 31/100
13/13 - 0s - loss: 79.2613 - 24ms/epoch - 2ms/step
Epoch 32/100
13/13 - 0s - loss: 78.8549 - 23ms/epoch - 2ms/step
Epoch 33/100
13/13 - 0s - loss: 78.3649 - 23ms/epoch - 2ms/step
Epoch 34/100
13/13 - 0s - loss: 78.0478 - 23ms/epoch - 2ms/step
Epoch 35/100
13/13 - 0s - loss: 77.6581 - 30ms/epoch - 2ms/step
Epoch 36/100
13/13 - 0s - loss: 77.1970 - 24ms/epoch - 2ms/step
Epoch 37/100
13/13 - 0s - loss: 76.8659 - 23ms/epoch - 2ms/step
Epoch 38/100
13/13 - 0s - loss: 76.6319 - 23ms/epoch - 2ms/step
Epoch 39/100
13/13 - 0s - loss: 76.0007 - 24ms/epoch - 2ms/step
Epoch 40/100
13/13 - 0s - loss: 75.5929 - 25ms/epoch - 2ms/step
Epoch 41/100
13/13 - 0s - loss: 75.2667 - 26ms/epoch - 2ms/step
Epoch 42/100
13/13 - 0s - loss: 75.3607 - 24ms/epoch - 2ms/step
Epoch 43/100
13/13 - 0s - loss: 74.5779 - 27ms/epoch - 2ms/step
Epoch 44/100
13/13 - 0s - loss: 73.9867 - 21ms/epoch - 2ms/step
Epoch 45/100
13/13 - 0s - loss: 73.7650 - 25ms/epoch - 2ms/step
Epoch 46/100
13/13 - 0s - loss: 73.0263 - 24ms/epoch - 2ms/step
Epoch 47/100
13/13 - 0s - loss: 72.7102 - 23ms/epoch - 2ms/step
Epoch 48/100
13/13 - 0s - loss: 72.2177 - 28ms/epoch - 2ms/step
Epoch 49/100
13/13 - 0s - loss: 71.8469 - 22ms/epoch - 2ms/step
Epoch 50/100
13/13 - 0s - loss: 71.4904 - 28ms/epoch - 2ms/step
Epoch 51/100
13/13 - 0s - loss: 71.1223 - 25ms/epoch - 2ms/step
Epoch 52/100
13/13 - 0s - loss: 70.5943 - 25ms/epoch - 2ms/step
Epoch 53/100
13/13 - 0s - loss: 70.1748 - 21ms/epoch - 2ms/step
Epoch 54/100
13/13 - 0s - loss: 69.8101 - 25ms/epoch - 2ms/step
Epoch 55/100
13/13 - 0s - loss: 69.3219 - 23ms/epoch - 2ms/step
Epoch 56/100
13/13 - 0s - loss: 68.7525 - 22ms/epoch - 2ms/step
```

```
Epoch 57/100
13/13 - 0s - loss: 68.4256 - 22ms/epoch - 2ms/step
Epoch 58/100
13/13 - 0s - loss: 67.8394 - 23ms/epoch - 2ms/step
Epoch 59/100
13/13 - 0s - loss: 67.4138 - 22ms/epoch - 2ms/step
Epoch 60/100
13/13 - 0s - loss: 66.9941 - 33ms/epoch - 3ms/step
Epoch 61/100
13/13 - 0s - loss: 66.6573 - 29ms/epoch - 2ms/step
Epoch 62/100
13/13 - 0s - loss: 66.1712 - 22ms/epoch - 2ms/step
Epoch 63/100
13/13 - 0s - loss: 65.8375 - 29ms/epoch - 2ms/step
Epoch 64/100
13/13 - 0s - loss: 65.3441 - 23ms/epoch - 2ms/step
Epoch 65/100
13/13 - 0s - loss: 64.9143 - 22ms/epoch - 2ms/step
Epoch 66/100
13/13 - 0s - loss: 64.7354 - 24ms/epoch - 2ms/step
Epoch 67/100
13/13 - 0s - loss: 63.8731 - 30ms/epoch - 2ms/step
Epoch 68/100
13/13 - 0s - loss: 63.5211 - 26ms/epoch - 2ms/step
Epoch 69/100
13/13 - 0s - loss: 62.9679 - 22ms/epoch - 2ms/step
Epoch 70/100
13/13 - 0s - loss: 62.6917 - 21ms/epoch - 2ms/step
Epoch 71/100
13/13 - 0s - loss: 62.1212 - 22ms/epoch - 2ms/step
Epoch 72/100
13/13 - 0s - loss: 62.1577 - 32ms/epoch - 2ms/step
Epoch 73/100
13/13 - 0s - loss: 61.1758 - 22ms/epoch - 2ms/step
Epoch 74/100
13/13 - 0s - loss: 61.0303 - 24ms/epoch - 2ms/step
Epoch 75/100
13/13 - 0s - loss: 60.5673 - 23ms/epoch - 2ms/step
Epoch 76/100
13/13 - 0s - loss: 60.0197 - 24ms/epoch - 2ms/step
Epoch 77/100
13/13 - 0s - loss: 59.7046 - 24ms/epoch - 2ms/step
Epoch 78/100
13/13 - 0s - loss: 59.0460 - 25ms/epoch - 2ms/step
Epoch 79/100
13/13 - 0s - loss: 58.6879 - 27ms/epoch - 2ms/step
Epoch 80/100
13/13 - 0s - loss: 58.2086 - 28ms/epoch - 2ms/step
Epoch 81/100
13/13 - 0s - loss: 58.1870 - 40ms/epoch - 3ms/step
Epoch 82/100
13/13 - 0s - loss: 57.3580 - 35ms/epoch - 3ms/step
Epoch 83/100
13/13 - 0s - loss: 57.0140 - 24ms/epoch - 2ms/step
Epoch 84/100
13/13 - 0s - loss: 56.5466 - 36ms/epoch - 3ms/step
```

```
Epoch 85/100
13/13 - 0s - loss: 56.2083 - 30ms/epoch - 2ms/step
Epoch 86/100
13/13 - 0s - loss: 55.7131 - 24ms/epoch - 2ms/step
Epoch 87/100
13/13 - 0s - loss: 55.2924 - 28ms/epoch - 2ms/step
Epoch 88/100
13/13 - 0s - loss: 54.9157 - 26ms/epoch - 2ms/step
Epoch 89/100
13/13 - 0s - loss: 54.8022 - 27ms/epoch - 2ms/step
Epoch 90/100
13/13 - 0s - loss: 53.9416 - 25ms/epoch - 2ms/step
Epoch 91/100
13/13 - 0s - loss: 53.6013 - 30ms/epoch - 2ms/step
Epoch 92/100
13/13 - 0s - loss: 53.3547 - 25ms/epoch - 2ms/step
Epoch 93/100
13/13 - 0s - loss: 52.7261 - 39ms/epoch - 3ms/step
Epoch 94/100
13/13 - 0s - loss: 52.3562 - 25ms/epoch - 2ms/step
Epoch 95/100
13/13 - 0s - loss: 51.9567 - 23ms/epoch - 2ms/step
Epoch 96/100
13/13 - 0s - loss: 51.4552 - 29ms/epoch - 2ms/step
Epoch 97/100
13/13 - 0s - loss: 51.4597 - 23ms/epoch - 2ms/step
Epoch 98/100
13/13 - 0s - loss: 50.6219 - 24ms/epoch - 2ms/step
Epoch 99/100
13/13 - 0s - loss: 50.2118 - 25ms/epoch - 2ms/step
Epoch 100/100
13/13 - 0s - loss: 49.8828 - 25ms/epoch - 2ms/step
Before save score (RMSE): 7.044431690300903
```

The code below sets up a neural network and reads the data (for predictions), but it does not clear the model directory or fit the neural network. The code loads the weights from the previous fit. Now we reload the network and perform another prediction. The RMSE should match the previous one exactly if we saved and reloaded the neural network correctly.

In [3]:
```
from tensorflow.keras.models import load_model
model2 = load_model(os.path.join(save_path,"network.h5"))
pred = model2.predict(x)
# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print(f"After load score (RMSE): {score}")
```

```
After load score (RMSE): 7.044431690300903
```