Open in Colab

T81-558: Applications of Deep Neural Networks

Module 8: Kaggle Data Sets

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

Module 8 Material

- **Part 8.1: Introduction to Kaggle** [Video] [Notebook]
- Part 8.2: Building Ensembles with Scikit-Learn and Keras [Video] [Notebook]
- Part 8.3: How Should you Architect Your Keras Neural Network: Hyperparameters [Video] [Notebook]
- Part 8.4: Bayesian Hyperparameter Optimization for Keras [Video] [Notebook]
- Part 8.5: Current Semester's Kaggle [Video] [Notebook]

Part 8.1: Introduction to Kaggle

Kaggle runs competitions where data scientists compete to provide the best model to fit the data. A simple project to get started with Kaggle is the Titanic data set. Most Kaggle competitions end on a specific date. Website organizers have scheduled the Titanic competition to end on December 31, 20xx (with the year usually rolling forward). However, they have already extended the deadline several times, and an extension beyond 2014 is also possible. Second, the Titanic data set is considered a tutorial data set. There is no prize, and your score in the competition does not count towards becoming a Kaggle Master.

Kaggle Ranks

You achieve Kaggle ranks by earning gold, silver, and bronze medals.

- Kaggle Top Users
- Current Top Kaggle User's Profile Page
- Jeff Heaton's (your instructor) Kaggle Profile

- Current Kaggle Ranking System

Typical Kaggle Competition

A typical Kaggle competition will have several components. Consider the Titanic tutorial:

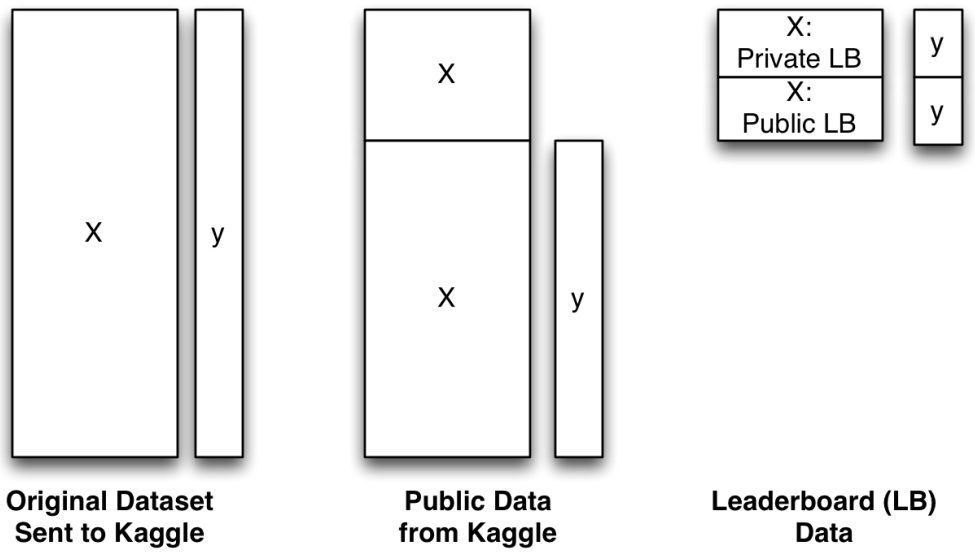
- Competition Summary Page
- Data Page
- Evaluation Description Page
- Leaderboard

How Kaggle Competition Scoring

Kaggle is provided with a data set by the competition sponsor, as seen in Figure 8.SCORE. Kaggle divides this data set as follows:

- **Complete Data Set** - This is the complete data set.
  - **Training Data Set** - This dataset provides both the inputs and the outcomes for the training portion of the data set.
  - **Test Data Set** - This dataset provides the complete test data; however, it does not give the outcomes. Your submission file should contain the predicted results for this data set.
    - **Public Leaderboard** - Kaggle does not tell you what part of the test data set contributes to the public leaderboard. Your public score is calculated based on this part of the data set.
    - **Private Leaderboard** - Likewise, Kaggle does not tell you what part of the test data set contributes to the public leaderboard. Your final score/rank is calculated based on this part. You do not see your private leaderboard score until the end.

Figure 8.SCORE: How Kaggle Competition Scoring



Preparing a Kaggle Submission

You do not submit the code to your solution to Kaggle. For competitions, you are scored entirely on the accuracy of your submission file. A Kaggle submission file is always a CSV file that contains the **Id** of the row you are predicting and the answer. For the titanic competition, a submission file looks something like this:

```
PassengerId,Survived
892,0
893,1
894,1
895,0
896,0
897,1
...
```

The above file states the prediction for each of the various passengers. You should only predict on ID's that are in the test file. Likewise, you should render a prediction for every row in the test file. Some competitions will have different formats for their answers. For example, a multi-classification will usually have a column for each class and your predictions for each class.

Select Kaggle Competitions

There have been many exciting competitions on Kaggle; these are some of my favorites. Some select predictive modeling competitions which use tabular data include:

- Otto Group Product Classification Challenge
- Galaxy Zoo - The Galaxy Challenge
- Practice Fusion Diabetes Classification
- Predicting a Biological Response

Many Kaggle competitions include computer vision datasets, such as:

- Diabetic Retinopathy Detection

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 8: Kaggle Data Sets

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

Module 8 Material

- Part 8.1: Introduction to Kaggle [Video] [Notebook]
- **Part 8.2: Building Ensembles with Scikit-Learn and Keras** [Video] [Notebook]
- Part 8.3: How Should you Architect Your Keras Neural Network: Hyperparameters [Video] [Notebook]
- Part 8.4: Bayesian Hyperparameter Optimization for Keras [Video] [Notebook]
- Part 8.5: Current Semester's Kaggle [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

# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return "{}:{:>02}:{:>05.2f}".format(h, m, s)
```

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

Part 8.2: Building Ensembles with Scikit-Learn and Keras

Evaluating Feature Importance

Feature importance tells us how important each feature (from the feature/import vector) is to predicting a neural network or another model. There are many different ways to evaluate the feature importance of neural networks. The following paper presents an excellent (and readable) overview of the various means of assessing the significance of neural network inputs/features.

- An accurate comparison of methods for quantifying variable importance in artificial neural networks using simulated data [Cite:olden2004accurate]. *Ecological Modelling*, 178(3), 389-397.

In summary, the following methods are available to neural networks:

- Connection Weights Algorithm
- Partial Derivatives
- Input Perturbation
- Sensitivity Analysis
- Forward Stepwise Addition
- Improved Stepwise Selection 1
- Backward Stepwise Elimination
- Improved Stepwise Selection

For this chapter, we will use the input Perturbation feature ranking algorithm. This algorithm will work with any regression or classification network. In the next section, I provide an implementation of the input perturbation algorithm for scikit-learn. This code implements a function below that will work with any scikit-learn model.

Leo Breiman provided this algorithm in his seminal paper on random forests. [Citebreiman2001random:] Although he presented this algorithm in conjunction with random forests, it is model-independent and appropriate for any supervised learning model. This algorithm, known as the input perturbation algorithm, works by evaluating a trained model's accuracy with each input individually shuffled from a data set. Shuffling an input causes it to become useless—effectively removing it from the model. More important inputs will produce a less accurate score when they are removed by shuffling them. This process makes sense because important features will contribute to the model's accuracy. I first presented the TensorFlow implementation of this algorithm in the following paper.

- Early stabilizing feature importance for TensorFlow deep neural networks[Cite:heaton2017early]

This algorithm will use log loss to evaluate a classification problem and RMSE for regression.

In [2]:
```
from sklearn import metrics
import scipy as sp
import numpy as np
import math
from sklearn import metrics

def perturbation_rank(model, x, y, names, regression):
    errors = []

    for i in range(x.shape[1]):
        hold = np.array(x[:, i])
        np.random.shuffle(x[:, i])

        if regression:
            pred = model.predict(x)
            error = metrics.mean_squared_error(y, pred)
        else:
            pred = model.predict(x)
            error = metrics.log_loss(y, pred)

        errors.append(error)
        x[:, i] = hold

    max_error = np.max(errors)
    importance = [e/max_error for e in errors]

    data = {'name':names,'error':errors,'importance':importance}
    result = pd.DataFrame(data, columns = ['name','error','importanc
    result.sort_values(by=['importance'], ascending=[0], inplace=Tru
    result.reset_index(inplace=True, drop=True)
    return result
```

Classification and Input Perturbation Ranking

We now look at the code to perform perturbation ranking for a classification neural network. The implementation technique is slightly different for classification vs. regression, so I must provide two different implementations. The primary difference between classification and regression is how we evaluate the accuracy of the neural network in each of these two network types. We will use the Root Mean Square (RMSE) error calculation, whereas we will use log loss for classification.

The code presented below creates a classification neural network that will predict the classic iris dataset.

In [3]:
```
# HIDE OUTPUT
import pandas as pd
import io
```

```
import requests
import numpy as np
from sklearn import metrics
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split

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

# Convert to numpy - Classification
x = df[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
dummies = pd.get_dummies(df['species']) # Classification
species = dummies.columns
y = dummies.values

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build neural network
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidd
model.add(Dense(25, activation='relu')) # Hidden 2
model.add(Dense(y.shape[1],activation='softmax')) # Output
model.compile(loss='categorical_crossentropy', optimizer='adam')
model.fit(x_train,y_train,verbose=2,epochs=100)
```

```
Epoch 1/100
4/4 - 1s - loss: 2.0814 - 1s/epoch - 292ms/step
Epoch 2/100
4/4 - 0s - loss: 1.6125 - 14ms/epoch - 4ms/step
Epoch 3/100
4/4 - 0s - loss: 1.3316 - 26ms/epoch - 7ms/step
Epoch 4/100
4/4 - 0s - loss: 1.2246 - 13ms/epoch - 3ms/step
Epoch 5/100
4/4 - 0s - loss: 1.1989 - 13ms/epoch - 3ms/step
Epoch 6/100
4/4 - 0s - loss: 1.1349 - 14ms/epoch - 4ms/step
Epoch 7/100
4/4 - 0s - loss: 1.0543 - 21ms/epoch - 5ms/step
Epoch 8/100
4/4 - 0s - loss: 0.9987 - 25ms/epoch - 6ms/step
Epoch 9/100
4/4 - 0s - loss: 0.9449 - 20ms/epoch - 5ms/step
Epoch 10/100
4/4 - 0s - loss: 0.9032 - 16ms/epoch - 4ms/step
Epoch 11/100
4/4 - 0s - loss: 0.8623 - 20ms/epoch - 5ms/step
Epoch 12/100
4/4 - 0s - loss: 0.8274 - 12ms/epoch - 3ms/step
Epoch 13/100
4/4 - 0s - loss: 0.8013 - 18ms/epoch - 4ms/step
Epoch 14/100
4/4 - 0s - loss: 0.7718 - 18ms/epoch - 5ms/step
Epoch 15/100
4/4 - 0s - loss: 0.7426 - 19ms/epoch - 5ms/step
Epoch 16/100
4/4 - 0s - loss: 0.7163 - 13ms/epoch - 3ms/step
Epoch 17/100
```

```
4/4 - 0s - loss: 0.6933 - 13ms/epoch - 3ms/step
Epoch 18/100
4/4 - 0s - loss: 0.6689 - 14ms/epoch - 3ms/step
Epoch 19/100
4/4 - 0s - loss: 0.6488 - 11ms/epoch - 3ms/step
Epoch 20/100
4/4 - 0s - loss: 0.6294 - 11ms/epoch - 3ms/step
Epoch 21/100
4/4 - 0s - loss: 0.6094 - 20ms/epoch - 5ms/step
Epoch 22/100
4/4 - 0s - loss: 0.5911 - 18ms/epoch - 4ms/step
Epoch 23/100
4/4 - 0s - loss: 0.5725 - 16ms/epoch - 4ms/step
Epoch 24/100
4/4 - 0s - loss: 0.5550 - 13ms/epoch - 3ms/step
Epoch 25/100
4/4 - 0s - loss: 0.5389 - 14ms/epoch - 3ms/step
Epoch 26/100
4/4 - 0s - loss: 0.5207 - 15ms/epoch - 4ms/step
Epoch 27/100
4/4 - 0s - loss: 0.5041 - 14ms/epoch - 4ms/step
Epoch 28/100
4/4 - 0s - loss: 0.4901 - 14ms/epoch - 3ms/step
Epoch 29/100
4/4 - 0s - loss: 0.4765 - 14ms/epoch - 4ms/step
Epoch 30/100
4/4 - 0s - loss: 0.4619 - 16ms/epoch - 4ms/step
Epoch 31/100
4/4 - 0s - loss: 0.4489 - 16ms/epoch - 4ms/step
Epoch 32/100
4/4 - 0s - loss: 0.4366 - 13ms/epoch - 3ms/step
Epoch 33/100
4/4 - 0s - loss: 0.4243 - 13ms/epoch - 3ms/step
Epoch 34/100
4/4 - 0s - loss: 0.4124 - 14ms/epoch - 3ms/step
Epoch 35/100
4/4 - 0s - loss: 0.4015 - 14ms/epoch - 3ms/step
Epoch 36/100
4/4 - 0s - loss: 0.3917 - 21ms/epoch - 5ms/step
Epoch 37/100
4/4 - 0s - loss: 0.3826 - 30ms/epoch - 7ms/step
Epoch 38/100
4/4 - 0s - loss: 0.3713 - 18ms/epoch - 4ms/step
Epoch 39/100
4/4 - 0s - loss: 0.3621 - 16ms/epoch - 4ms/step
Epoch 40/100
4/4 - 0s - loss: 0.3543 - 14ms/epoch - 4ms/step
Epoch 41/100
4/4 - 0s - loss: 0.3460 - 14ms/epoch - 4ms/step
Epoch 42/100
4/4 - 0s - loss: 0.3385 - 28ms/epoch - 7ms/step
Epoch 43/100
4/4 - 0s - loss: 0.3280 - 31ms/epoch - 8ms/step
Epoch 44/100
4/4 - 0s - loss: 0.3211 - 15ms/epoch - 4ms/step
Epoch 45/100
4/4 - 0s - loss: 0.3144 - 14ms/epoch - 3ms/step
Epoch 46/100
4/4 - 0s - loss: 0.3068 - 15ms/epoch - 4ms/step
Epoch 47/100
4/4 - 0s - loss: 0.2992 - 19ms/epoch - 5ms/step
Epoch 48/100
4/4 - 0s - loss: 0.2922 - 18ms/epoch - 4ms/step
```

```
4/4 - 0s - loss: 0.2922 - 18ms/epoch - 4ms/step
Epoch 49/100
4/4 - 0s - loss: 0.2847 - 37ms/epoch - 9ms/step
Epoch 50/100
4/4 - 0s - loss: 0.2803 - 13ms/epoch - 3ms/step
Epoch 51/100
4/4 - 0s - loss: 0.2756 - 13ms/epoch - 3ms/step
Epoch 52/100
4/4 - 0s - loss: 0.2665 - 18ms/epoch - 4ms/step
Epoch 53/100
4/4 - 0s - loss: 0.2632 - 16ms/epoch - 4ms/step
Epoch 54/100
4/4 - 0s - loss: 0.2571 - 20ms/epoch - 5ms/step
Epoch 55/100
4/4 - 0s - loss: 0.2499 - 17ms/epoch - 4ms/step
Epoch 56/100
4/4 - 0s - loss: 0.2458 - 17ms/epoch - 4ms/step
Epoch 57/100
4/4 - 0s - loss: 0.2399 - 23ms/epoch - 6ms/step
Epoch 58/100
4/4 - 0s - loss: 0.2340 - 16ms/epoch - 4ms/step
Epoch 59/100
4/4 - 0s - loss: 0.2318 - 16ms/epoch - 4ms/step
Epoch 60/100
4/4 - 0s - loss: 0.2225 - 12ms/epoch - 3ms/step
Epoch 61/100
4/4 - 0s - loss: 0.2266 - 15ms/epoch - 4ms/step
Epoch 62/100
4/4 - 0s - loss: 0.2178 - 12ms/epoch - 3ms/step
Epoch 63/100
4/4 - 0s - loss: 0.2116 - 15ms/epoch - 4ms/step
Epoch 64/100
4/4 - 0s - loss: 0.2137 - 21ms/epoch - 5ms/step
Epoch 65/100
4/4 - 0s - loss: 0.2030 - 17ms/epoch - 4ms/step
Epoch 66/100
4/4 - 0s - loss: 0.2041 - 14ms/epoch - 3ms/step
Epoch 67/100
4/4 - 0s - loss: 0.2001 - 15ms/epoch - 4ms/step
Epoch 68/100
4/4 - 0s - loss: 0.1919 - 25ms/epoch - 6ms/step
Epoch 69/100
4/4 - 0s - loss: 0.1894 - 23ms/epoch - 6ms/step
Epoch 70/100
4/4 - 0s - loss: 0.1863 - 17ms/epoch - 4ms/step
Epoch 71/100
4/4 - 0s - loss: 0.1823 - 16ms/epoch - 4ms/step
Epoch 72/100
4/4 - 0s - loss: 0.1790 - 24ms/epoch - 6ms/step
Epoch 73/100
4/4 - 0s - loss: 0.1780 - 16ms/epoch - 4ms/step
Epoch 74/100
4/4 - 0s - loss: 0.1755 - 15ms/epoch - 4ms/step
Epoch 75/100
4/4 - 0s - loss: 0.1719 - 31ms/epoch - 8ms/step
Epoch 76/100
4/4 - 0s - loss: 0.1767 - 21ms/epoch - 5ms/step
Epoch 77/100
4/4 - 0s - loss: 0.1694 - 17ms/epoch - 4ms/step
Epoch 78/100
4/4 - 0s - loss: 0.1655 - 27ms/epoch - 7ms/step
Epoch 79/100
4/4 - 0s - loss: 0.1634 - 23ms/epoch - 6ms/step
Epoch 80/100
```

```
Epoch 80/100
4/4 - 0s - loss: 0.1566 - 17ms/epoch - 4ms/step
Epoch 81/100
4/4 - 0s - loss: 0.1563 - 17ms/epoch - 4ms/step
Epoch 82/100
4/4 - 0s - loss: 0.1536 - 15ms/epoch - 4ms/step
Epoch 83/100
4/4 - 0s - loss: 0.1504 - 18ms/epoch - 4ms/step
Epoch 84/100
4/4 - 0s - loss: 0.1502 - 16ms/epoch - 4ms/step
Epoch 85/100
4/4 - 0s - loss: 0.1469 - 17ms/epoch - 4ms/step
Epoch 86/100
4/4 - 0s - loss: 0.1448 - 28ms/epoch - 7ms/step
Epoch 87/100
4/4 - 0s - loss: 0.1424 - 23ms/epoch - 6ms/step
Epoch 88/100
4/4 - 0s - loss: 0.1401 - 25ms/epoch - 6ms/step
Epoch 89/100
4/4 - 0s - loss: 0.1386 - 47ms/epoch - 12ms/step
Epoch 90/100
4/4 - 0s - loss: 0.1365 - 30ms/epoch - 7ms/step
Epoch 91/100
4/4 - 0s - loss: 0.1383 - 41ms/epoch - 10ms/step
Epoch 92/100
4/4 - 0s - loss: 0.1332 - 12ms/epoch - 3ms/step
Epoch 93/100
4/4 - 0s - loss: 0.1311 - 20ms/epoch - 5ms/step
Epoch 94/100
4/4 - 0s - loss: 0.1320 - 20ms/epoch - 5ms/step
Epoch 95/100
4/4 - 0s - loss: 0.1302 - 17ms/epoch - 4ms/step
Epoch 96/100
4/4 - 0s - loss: 0.1311 - 19ms/epoch - 5ms/step
Epoch 97/100
4/4 - 0s - loss: 0.1248 - 14ms/epoch - 3ms/step
Epoch 98/100
4/4 - 0s - loss: 0.1254 - 12ms/epoch - 3ms/step
Epoch 99/100
4/4 - 0s - loss: 0.1275 - 13ms/epoch - 3ms/step
Epoch 100/100
4/4 - 0s - loss: 0.1225 - 41ms/epoch - 10ms/step
```

Out[3]: `<keras.callbacks.History at 0x7fc869fd2950>`

Next, we evaluate the accuracy of the trained model. Here we see that the neural network performs great, with an accuracy of 1.0. We might fear overfitting with such high accuracy for a more complex dataset. However, for this example, we are more interested in determining the importance of each column.

In [4]:

```
from sklearn.metrics import accuracy_score

pred = model.predict(x_test)
predict_classes = np.argmax(pred,axis=1)
expected_classes = np.argmax(y_test,axis=1)
correct = accuracy_score(expected_classes,predict_classes)
print(f"Accuracy: {correct}")
```

```
Accuracy: 1.0
```

We are now ready to call the input perturbation algorithm. First, we extract the column names and remove the target column. The target column is not important, as it is the objective, not one of the inputs. In supervised learning, the target is of the utmost importance.

We can see the importance displayed in the following table. The most important column is always 1.0, and lessor columns will continue in a downward trend. The least important column will have the lowest rank.

In [5]:
```
# Rank the features
from IPython.display import display, HTML

names = list(df.columns) # x+y column names
names.remove("species") # remove the target(y)
rank = perturbation_rank(model, x_test, y_test, names, False)
display(rank)
```

	name	error	importance
0	petal_l	2.609378	1.000000
1	petal_w	0.480387	0.184100
2	sepal_l	0.223239	0.085553
3	sepal_w	0.128518	0.049252

Regression and Input Perturbation Ranking

We now see how to use input perturbation ranking for a regression neural network. We will use the MPG dataset as a demonstration. The code below loads the MPG dataset and creates a regression neural network for this dataset. The code trains the neural network and calculates an RMSE evaluation.

In [6]:
```
# HIDE OUTPUT
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.model_selection import train_test_split
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
```

```
# 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

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

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

# Predict
pred = model.predict(x)
```

```
Epoch 1/100
10/10 - 1s - loss: 328433.8125 - 898ms/epoch - 90ms/step
Epoch 2/100
10/10 - 0s - loss: 78914.6406 - 26ms/epoch - 3ms/step
Epoch 3/100
10/10 - 0s - loss: 5371.1025 - 50ms/epoch - 5ms/step
Epoch 4/100
10/10 - 0s - loss: 4021.9753 - 34ms/epoch - 3ms/step
Epoch 5/100
10/10 - 0s - loss: 4438.5728 - 33ms/epoch - 3ms/step
Epoch 6/100
10/10 - 0s - loss: 1030.3115 - 34ms/epoch - 3ms/step
Epoch 7/100
10/10 - 0s - loss: 594.9177 - 31ms/epoch - 3ms/step
Epoch 8/100
10/10 - 0s - loss: 655.3908 - 31ms/epoch - 3ms/step
Epoch 9/100
10/10 - 0s - loss: 465.0457 - 25ms/epoch - 2ms/step
Epoch 10/100
10/10 - 0s - loss: 458.7520 - 30ms/epoch - 3ms/step
Epoch 11/100
10/10 - 0s - loss: 452.4102 - 22ms/epoch - 2ms/step
Epoch 12/100
10/10 - 0s - loss: 439.8730 - 25ms/epoch - 3ms/step
Epoch 13/100
10/10 - 0s - loss: 434.8245 - 27ms/epoch - 3ms/step
Epoch 14/100
10/10 - 0s - loss: 433.7303 - 25ms/epoch - 3ms/step
Epoch 15/100
10/10 - 0s - loss: 427.2859 - 46ms/epoch - 5ms/step
Epoch 16/100
10/10 - 0s - loss: 424.1164 - 50ms/epoch - 5ms/step
Epoch 17/100
10/10 - 0s - loss: 422.3007 - 42ms/epoch - 4ms/step
Epoch 18/100
10/10 - 0s - loss: 418.4877 - 31ms/epoch - 3ms/step
Epoch 19/100
10/10 - 0s - loss: 414.2283 - 23ms/epoch - 2ms/step
Epoch 20/100
10/10 - 0s - loss: 410.2691 - 34ms/epoch - 3ms/step
```

```
Epoch 21/100
10/10 - 0s - loss: 407.0490 - 29ms/epoch - 3ms/step
Epoch 22/100
10/10 - 0s - loss: 406.2433 - 46ms/epoch - 5ms/step
Epoch 23/100
10/10 - 0s - loss: 399.7404 - 37ms/epoch - 4ms/step
Epoch 24/100
10/10 - 0s - loss: 396.3280 - 66ms/epoch - 7ms/step
Epoch 25/100
10/10 - 0s - loss: 391.0629 - 28ms/epoch - 3ms/step
Epoch 26/100
10/10 - 0s - loss: 387.3203 - 26ms/epoch - 3ms/step
Epoch 27/100
10/10 - 0s - loss: 382.7670 - 54ms/epoch - 5ms/step
Epoch 28/100
10/10 - 0s - loss: 380.6316 - 21ms/epoch - 2ms/step
Epoch 29/100
10/10 - 0s - loss: 375.9518 - 30ms/epoch - 3ms/step
Epoch 30/100
10/10 - 0s - loss: 372.7001 - 24ms/epoch - 2ms/step
Epoch 31/100
10/10 - 0s - loss: 366.7871 - 24ms/epoch - 2ms/step
Epoch 32/100
10/10 - 0s - loss: 363.4180 - 42ms/epoch - 4ms/step
Epoch 33/100
10/10 - 0s - loss: 359.6006 - 47ms/epoch - 5ms/step
Epoch 34/100
10/10 - 0s - loss: 359.4055 - 46ms/epoch - 5ms/step
Epoch 35/100
10/10 - 0s - loss: 350.7181 - 29ms/epoch - 3ms/step
Epoch 36/100
10/10 - 0s - loss: 348.6260 - 42ms/epoch - 4ms/step
Epoch 37/100
10/10 - 0s - loss: 343.6122 - 28ms/epoch - 3ms/step
Epoch 38/100
10/10 - 0s - loss: 339.6165 - 32ms/epoch - 3ms/step
Epoch 39/100
10/10 - 0s - loss: 334.5634 - 32ms/epoch - 3ms/step
Epoch 40/100
10/10 - 0s - loss: 332.6061 - 34ms/epoch - 3ms/step
Epoch 41/100
10/10 - 0s - loss: 326.7434 - 22ms/epoch - 2ms/step
Epoch 42/100
10/10 - 0s - loss: 323.8063 - 40ms/epoch - 4ms/step
Epoch 43/100
10/10 - 0s - loss: 320.2585 - 29ms/epoch - 3ms/step
Epoch 44/100
10/10 - 0s - loss: 315.3609 - 23ms/epoch - 2ms/step
Epoch 45/100
10/10 - 0s - loss: 311.4920 - 23ms/epoch - 2ms/step
Epoch 46/100
10/10 - 0s - loss: 308.9212 - 29ms/epoch - 3ms/step
Epoch 47/100
10/10 - 0s - loss: 303.1410 - 24ms/epoch - 2ms/step
Epoch 48/100
10/10 - 0s - loss: 299.9317 - 24ms/epoch - 2ms/step
Epoch 49/100
10/10 - 0s - loss: 294.4305 - 23ms/epoch - 2ms/step
Epoch 50/100
10/10 - 0s - loss: 291.4469 - 24ms/epoch - 2ms/step
Epoch 51/100
10/10 - 0s - loss: 287.3263 - 41ms/epoch - 4ms/step
Epoch 52/100
```

```
10/10 - 0s - loss: 284.3096 - 49ms/epoch - 5ms/step
Epoch 53/100
10/10 - 0s - loss: 280.5522 - 30ms/epoch - 3ms/step
Epoch 54/100
10/10 - 0s - loss: 276.1487 - 26ms/epoch - 3ms/step
Epoch 55/100
10/10 - 0s - loss: 271.3444 - 42ms/epoch - 4ms/step
Epoch 56/100
10/10 - 0s - loss: 280.0936 - 33ms/epoch - 3ms/step
Epoch 57/100
10/10 - 0s - loss: 263.7166 - 40ms/epoch - 4ms/step
Epoch 58/100
10/10 - 0s - loss: 261.6750 - 56ms/epoch - 6ms/step
Epoch 59/100
10/10 - 0s - loss: 258.5714 - 45ms/epoch - 4ms/step
Epoch 60/100
10/10 - 0s - loss: 252.6791 - 31ms/epoch - 3ms/step
Epoch 61/100
10/10 - 0s - loss: 250.1348 - 53ms/epoch - 5ms/step
Epoch 62/100
10/10 - 0s - loss: 246.4157 - 72ms/epoch - 7ms/step
Epoch 63/100
10/10 - 0s - loss: 242.3768 - 46ms/epoch - 5ms/step
Epoch 64/100
10/10 - 0s - loss: 238.7874 - 28ms/epoch - 3ms/step
Epoch 65/100
10/10 - 0s - loss: 235.8578 - 42ms/epoch - 4ms/step
Epoch 66/100
10/10 - 0s - loss: 233.7492 - 24ms/epoch - 2ms/step
Epoch 67/100
10/10 - 0s - loss: 229.0066 - 26ms/epoch - 3ms/step
Epoch 68/100
10/10 - 0s - loss: 225.7449 - 25ms/epoch - 3ms/step
Epoch 69/100
10/10 - 0s - loss: 223.5038 - 25ms/epoch - 2ms/step
Epoch 70/100
10/10 - 0s - loss: 219.9561 - 39ms/epoch - 4ms/step
Epoch 71/100
10/10 - 0s - loss: 215.1055 - 58ms/epoch - 6ms/step
Epoch 72/100
10/10 - 0s - loss: 211.9364 - 39ms/epoch - 4ms/step
Epoch 73/100
10/10 - 0s - loss: 208.1019 - 55ms/epoch - 5ms/step
Epoch 74/100
10/10 - 0s - loss: 207.4119 - 34ms/epoch - 3ms/step
Epoch 75/100
10/10 - 0s - loss: 206.8693 - 40ms/epoch - 4ms/step
Epoch 76/100
10/10 - 0s - loss: 197.9749 - 49ms/epoch - 5ms/step
Epoch 77/100
10/10 - 0s - loss: 196.9090 - 34ms/epoch - 3ms/step
Epoch 78/100
10/10 - 0s - loss: 192.6349 - 45ms/epoch - 4ms/step
Epoch 79/100
10/10 - 0s - loss: 189.6783 - 31ms/epoch - 3ms/step
Epoch 80/100
10/10 - 0s - loss: 186.6584 - 25ms/epoch - 2ms/step
Epoch 81/100
10/10 - 0s - loss: 186.1920 - 29ms/epoch - 3ms/step
Epoch 82/100
10/10 - 0s - loss: 181.1735 - 31ms/epoch - 3ms/step
Epoch 83/100
10/10 - 0s - loss: 177.9338 - 51ms/epoch - 5ms/step
```

```
Epoch 84/100
10/10 - 0s - loss: 174.6662 - 87ms/epoch - 9ms/step
Epoch 85/100
10/10 - 0s - loss: 172.9421 - 90ms/epoch - 9ms/step
Epoch 86/100
10/10 - 0s - loss: 169.1906 - 58ms/epoch - 6ms/step
Epoch 87/100
10/10 - 0s - loss: 166.4181 - 57ms/epoch - 6ms/step
Epoch 88/100
10/10 - 0s - loss: 163.7466 - 36ms/epoch - 4ms/step
Epoch 89/100
10/10 - 0s - loss: 161.4653 - 29ms/epoch - 3ms/step
Epoch 90/100
10/10 - 0s - loss: 158.6274 - 30ms/epoch - 3ms/step
Epoch 91/100
10/10 - 0s - loss: 159.4237 - 32ms/epoch - 3ms/step
Epoch 92/100
10/10 - 0s - loss: 159.2035 - 31ms/epoch - 3ms/step
Epoch 93/100
10/10 - 0s - loss: 150.2793 - 38ms/epoch - 4ms/step
Epoch 94/100
10/10 - 0s - loss: 148.9276 - 36ms/epoch - 4ms/step
Epoch 95/100
10/10 - 0s - loss: 146.7706 - 34ms/epoch - 3ms/step
Epoch 96/100
10/10 - 0s - loss: 144.4946 - 29ms/epoch - 3ms/step
Epoch 97/100
10/10 - 0s - loss: 141.5782 - 28ms/epoch - 3ms/step
Epoch 98/100
10/10 - 0s - loss: 139.3355 - 27ms/epoch - 3ms/step
Epoch 99/100
10/10 - 0s - loss: 136.9762 - 56ms/epoch - 6ms/step
Epoch 100/100
10/10 - 0s - loss: 135.6660 - 23ms/epoch - 2ms/step
```

Just as before, we extract the column names and discard the target. We can now create a ranking of the importance of each of the input features. The feature with a ranking of 1.0 is the most important.

In [7]:

```
# Rank the features
from IPython.display import display, HTML

names = list(df.columns) # x+y column names
names.remove("name")
names.remove("mpg") # remove the target(y)
rank = perturbation_rank(model, x_test, y_test, names, True)
display(rank)
```

	name	error	importance
0	displacement	139.657598	1.000000
1	acceleration	139.261508	0.997164
2	origin	134.637690	0.964056
3	year	134.177126	0.960758
4	cylinders	132.747246	0.950519
5	horsepower	121.501102	0.869993

6	weight	75.244610	0.538779

Biological Response with Neural Network

The following sections will demonstrate how to use feature importance ranking and ensembling with a more complex dataset. Ensembling is the process where you combine multiple models for greater accuracy. Kaggle competition winners frequently make use of ensembling for high-ranking solutions.

We will use the biological response dataset, a Kaggle dataset, where there is an unusually high number of columns. Because of the large number of columns, it is essential to use feature ranking to determine the importance of these columns. We begin by loading the dataset and preprocessing. This Kaggle dataset is a binary classification problem. You must predict if certain conditions will cause a biological response.

- Predicting a Biological Response

In [8]:
```
import pandas as pd
import os
import numpy as np
from sklearn import metrics
from scipy.stats import zscore
from sklearn.model_selection import KFold
from IPython.display import HTML, display

URL = "https://data.heatonresearch.com/data/t81-558/kaggle/"

df_train = pd.read_csv(
    URL+"bio_train.csv",
    na_values=['NA', '?'])

df_test = pd.read_csv(
    URL+"bio_test.csv",
    na_values=['NA', '?'])

activity_classes = df_train['Activity']
```

A large number of columns is evident when we display the shape of the dataset.

In [9]:
```
print(df_train.shape)
```

```
(3751, 1777)
```

The following code constructs a classification neural network and trains it for the biological response dataset. Once trained, the accuracy is measured.

In [10]:
```
import os
import pandas as pd
```

```
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
import numpy as np
import sklearn

# Encode feature vector
# Convert to numpy - Classification
x_columns = df_train.columns.drop('Activity')
x = df_train[x_columns].values
y = df_train['Activity'].values # Classification
x_submit = df_test[x_columns].values.astype(np.float32)


# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

print("Fitting/Training...")
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu'))
model.add(Dense(10))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                        patience=5, verbose=1, mode='auto')
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=0,epochs=1000)
print("Fitting done...")

# Predict
pred = model.predict(x_test).flatten()


# Clip so that min is never exactly 0, max never 1
pred = np.clip(pred,a_min=1e-6,a_max=(1-1e-6))
print("Validation logloss: {}".format(
    sklearn.metrics.log_loss(y_test,pred)))

# Evaluate success using accuracy
pred = pred>0.5 # If greater than 0.5 probability, then true
score = metrics.accuracy_score(y_test, pred)
print("Validation accuracy score: {}".format(score))

# Build real submit file
pred_submit = model.predict(x_submit)

# Clip so that min is never exactly 0, max never 1 (would be a NaN s
pred = np.clip(pred,a_min=1e-6,a_max=(1-1e-6))
submit_df = pd.DataFrame({'MoleculeId':[x+1 for x \
        in range(len(pred_submit))],'PredictedProbability':\
                           pred_submit.flatten()})
submit_df.to_csv("submit.csv", index=False)
```

```
Fitting/Training...
Epoch 7: early stopping
Fitting done...
Validation logloss: 0.5564708781752792
Validation accuracy score: 0.7515991471215352
```

What Features/Columns are Important

The following uses perturbation ranking to evaluate the neural network.

In [11]:
```
# Rank the features
from IPython.display import display, HTML

names = list(df_train.columns) # x+y column names
names.remove("Activity") # remove the target(y)
rank = perturbation_rank(model, x_test, y_test, names, False)
display(rank[0:10])
```

	name	error	importance
0	D27	0.603974	1.000000
1	D1049	0.565997	0.937122
2	D51	0.565883	0.936934
3	D998	0.563872	0.933604
4	D1059	0.563745	0.933394
5	D961	0.563723	0.933357
6	D1407	0.563532	0.933041
7	D1309	0.562244	0.930908
8	D1100	0.561902	0.930341
9	D1275	0.561659	0.929940

Neural Network Ensemble

A neural network ensemble combines neural network predictions with other models. The program determines the exact blend of these models by logistic regression. The following code performs this blend for a classification. If you present the final predictions from the ensemble to Kaggle, you will see that the result is very accurate.

In [12]:
```
# HIDE OUTPUT
import numpy as np
import os
import pandas as pd
import math
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression

SHUFFLE = False
```

```
FOLDS = 10

def build_ann(input_size,classes,neurons):
    model = Sequential()
    model.add(Dense(neurons, input_dim=input_size, activation='relu'
    model.add(Dense(1))
    model.add(Dense(classes,activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam')
    return model

def mlogloss(y_test, preds):
    epsilon = 1e-15
    sum = 0
    for row in zip(preds,y_test):
        x = row[0][row[1]]
        x = max(epsilon,x)
        x = min(1-epsilon,x)
        sum+=math.log(x)
    return( (-1/len(preds))*sum)

def stretch(y):
    return (y - y.min()) / (y.max() - y.min())


def blend_ensemble(x, y, x_submit):
    kf = StratifiedKFold(FOLDS)
    folds = list(kf.split(x,y))

    models = [
        KerasClassifier(build_fn=build_ann,neurons=20,
                    input_size=x.shape[1],classes=2),
        KNeighborsClassifier(n_neighbors=3),
        RandomForestClassifier(n_estimators=100, n_jobs=-1,
                               criterion='gini'),
        RandomForestClassifier(n_estimators=100, n_jobs=-1,
                               criterion='entropy'),
        ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
                             criterion='gini'),
        ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
                             criterion='entropy'),
        GradientBoostingClassifier(learning_rate=0.05,
                subsample=0.5, max_depth=6, n_estimators=50)]

    dataset_blend_train = np.zeros((x.shape[0], len(models)))
    dataset_blend_test = np.zeros((x_submit.shape[0], len(models)))

    for j, model in enumerate(models):
        print("Model: {} : {}".format(j, model) )
        fold_sums = np.zeros((x_submit.shape[0], len(folds)))
        total_loss = 0
        for i, (train, test) in enumerate(folds):
            x_train = x[train]
            y_train = y[train]
            x_test = x[test]
            y_test = y[test]
            model.fit(x_train, y_train)
            pred = np.array(model.predict_proba(x_test))
            dataset_blend_train[test, j] = pred[:, 1]
            pred2 = np.array(model.predict_proba(x_submit))
            fold_sums[:, i] = pred2[:, 1]
            loss = mlogloss(y_test, pred)
            total_loss+=loss
            print("Fold #{}: loss={}".format(i,loss))
```

```
            print("Fold #{}: loss={}".format(i,loss))
        print("{}: Mean loss={}".format(model.__class__.__name__,
                                        total_loss/len(folds)))
        dataset_blend_test[:, j] = fold_sums.mean(1)

    print()
    print("Blending models.")
    blend = LogisticRegression(solver='lbfgs')
    blend.fit(dataset_blend_train, y)
    return blend.predict_proba(dataset_blend_test)

if __name__ == '__main__':

    np.random.seed(42)  # seed to shuffle the train set

    print("Loading data...")
    URL = "https://data.heatonresearch.com/data/t81-558/kaggle/"

    df_train = pd.read_csv(
        URL+"bio_train.csv",
        na_values=['NA', '?'])

    df_submit = pd.read_csv(
        URL+"bio_test.csv",
        na_values=['NA', '?'])

    predictors = list(df_train.columns.values)
    predictors.remove('Activity')
    x = df_train[predictors].values
    y = df_train['Activity']
    x_submit = df_submit.values

    if SHUFFLE:
        idx = np.random.permutation(y.size)
        x = x[idx]
        y = y[idx]

    submit_data = blend_ensemble(x, y, x_submit)
    submit_data = stretch(submit_data)

    ####################
    # Build submit file
    ####################
    ids = [id+1 for id in range(submit_data.shape[0])]
    submit_df = pd.DataFrame({'MoleculeId': ids,
                              'PredictedProbability':
                              submit_data[:, 1]},
                             columns=['MoleculeId',
                            'PredictedProbability'])
    submit_df.to_csv("submit.csv", index=False)
```

```
Loading data...
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:44: Depr
ecationWarning: KerasClassifier is deprecated, use Sci-Keras (http
s://github.com/adriangb/scikeras) instead. See https://www.adriangb.c
om/scikeras/stable/migration.html for help migrating.
Model: 0 : <keras.wrappers.scikit_learn.KerasClassifier object at 0x7
fc869809610>
106/106 [==============================] - 1s 2ms/step - loss: 0.6048
Fold #0: loss=0.5544745638322883
106/106 [==============================] - 1s 2ms/step - loss: 0.6046
Fold #1: loss=0.5684765604955473
106/106 [==============================] - 1s 2ms/step - loss: 0.5943
```

```
Fold #2: loss=0.5214491621944897
106/106 [==============================] - 1s 2ms/step - loss: 0.6301
Fold #3: loss=0.5264746750391351
106/106 [==============================] - 1s 2ms/step - loss: 0.5905
Fold #4: loss=0.5327822461352748
106/106 [==============================] - 1s 2ms/step - loss: 0.5993
Fold #5: loss=0.5800157462831582
106/106 [==============================] - 1s 2ms/step - loss: 0.5877
Fold #6: loss=0.5189563830365144
106/106 [==============================] - 1s 2ms/step - loss: 0.6038
Fold #7: loss=0.5625417655617023
106/106 [==============================] - 1s 2ms/step - loss: 0.5935
Fold #8: loss=0.5238374326475557
106/106 [==============================] - 1s 2ms/step - loss: 0.5991
Fold #9: loss=0.5322226787930878
KerasClassifier: Mean loss=0.5421231214018752
Model: 1 : KNeighborsClassifier(n_neighbors=3)
Fold #0: loss=3.606678388314123
Fold #1: loss=2.2256421551487593
Fold #2: loss=3.6815437059542186
Fold #3: loss=2.416161292225968
Fold #4: loss=4.442472310149748
Fold #5: loss=4.321350530738247
Fold #6: loss=3.400455469543658
Fold #7: loss=3.1724147110842513
Fold #8: loss=2.117356283193681
Fold #9: loss=3.0532135963322586
KNeighborsClassifier: Mean loss=3.243728844268491
Model: 2 : RandomForestClassifier(n_jobs=-1)
Fold #0: loss=0.4657177982691548
Fold #1: loss=0.4346825805694879
Fold #2: loss=0.4593868993445528
Fold #3: loss=0.41674899522216713
Fold #4: loss=0.4851849131056564
Fold #5: loss=0.48473291073937
Fold #6: loss=0.41274608628217674
Fold #7: loss=0.47405291219252377
Fold #8: loss=0.44974230059938286
Fold #9: loss=0.46340159258241087
RandomForestClassifier: Mean loss=0.45463969889068834
Model: 3 : RandomForestClassifier(criterion='entropy', n_jobs=-1)
Fold #0: loss=0.4511847247326708
Fold #1: loss=0.42707704254926593
Fold #2: loss=0.5550335199035183
Fold #3: loss=0.42186970733328516
```

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 8: Kaggle Data Sets

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

Module 8 Material

- Part 8.1: Introduction to Kaggle [Video] [Notebook]
- Part 8.2: Building Ensembles with Scikit-Learn and Keras [Video] [Notebook]
- **Part 8.3: How Should you Architect Your Keras Neural Network: Hyperparameters** [Video] [Notebook]
- Part 8.4: Bayesian Hyperparameter Optimization for Keras [Video] [Notebook]
- Part 8.5: Current Semester's Kaggle [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
# Startup CoLab
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False


# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return "{}:{:>02}:{:>05.2f}".format(h, m, s)
```

```
Note: not using Google CoLab
```

Part 8.3: Architecting Network: Hyperparameters

You have probably noticed several hyperparameters introduced previously in this course that you need to choose for your neural network. The number of layers, neuron counts per layer, layer types, and activation functions are all choices you must make to optimize your neural network. Some of the categories of hyperparameters for you to choose from coming from the following list:

- Number of Hidden Layers and Neuron Counts
- Activation Functions
- Advanced Activation Functions
- Regularization: L1, L2, Dropout
- Batch Normalization
- Training Parameters

The following sections will introduce each of these categories for Keras. While I will provide some general guidelines for hyperparameter selection, no two tasks are the same. You will benefit from experimentation with these values to determine what works best for your neural network. In the next part, we will see how machine learning can select some of these values independently.

Number of Hidden Layers and Neuron Counts

The structure of Keras layers is perhaps the hyperparameters that most become aware of first. How many layers should you have? How many neurons are on each layer? What activation function and layer type should you use? These are all questions that come up when designing a neural network. There are many different types of layer in Keras, listed here:

- **Activation** - You can also add activation functions as layers. Using the activation layer is the same as specifying the activation function as part of a Dense (or other) layer type.
- **ActivityRegularization** Used to add L1/L2 regularization outside of a layer. You can specify L1 and L2 as part of a Dense (or other) layer type.
- **Dense** - The original neural network layer type. In this layer type, every neuron connects to the next layer. The input vector is one-dimensional, and placing specific inputs next does not affect each other.
- **Dropout** - Dropout consists of randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting. Dropout only occurs during training.
- **Flatten** - Flattens the input to 1D and does not affect the batch size.

- **Input** - A Keras tensor is a tensor object from the underlying back end (Theano, TensorFlow, or CNTK), which we augment with specific attributes to build a Keras by knowing the inputs and outputs of the model.
- **Lambda** - Wraps arbitrary expression as a Layer object.
- **Masking** - Masks a sequence using a mask value to skip timesteps.
- **Permute** - Permutes the input dimensions according to a given pattern. Useful for tasks such as connecting RNNs and convolutional networks.
- **RepeatVector** - Repeats the input n times.
- **Reshape** - Similar to Numpy reshapes.
- **SpatialDropout1D** - This version performs the same function as Dropout; however, it drops entire 1D feature maps instead of individual elements.
- **SpatialDropout2D** - This version performs the same function as Dropout; however, it drops entire 2D feature maps instead of individual elements
- **SpatialDropout3D** - This version performs the same function as Dropout; however, it drops entire 3D feature maps instead of individual elements.

There is always trial and error for choosing a good number of neurons and hidden layers. Generally, the number of neurons on each layer will be larger closer to the hidden layer and smaller towards the output layer. This configuration gives the neural network a somewhat triangular or trapezoid appearance.

Activation Functions

Activation functions are a choice that you must make for each layer. Generally, you can follow this guideline:

- Hidden Layers - RELU
- Output Layer - Softmax for classification, linear for regression.

Some of the common activation functions in Keras are listed here:

- **softmax** - Used for multi-class classification. Ensures all output neurons behave as probabilities and sum to 1.0.
- **elu** - Exponential linear unit. Exponential Linear Unit or its widely known name ELU is a function that tends to converge cost to zero faster and produce more accurate results. Can produce negative outputs.
- **selu** - Scaled Exponential Linear Unit (SELU), essentially **elu** multiplied by a scaling constant.
- **softplus** - Softplus activation function. $log(exp(x) + 1)$ Introduced in 2001.
- **softsign** Softsign activation function. $x/(abs(x) + 1)$ Similar to tanh, but not widely used.
- **relu** - Very popular neural network activation function. Used for hidden layers, cannot output negative values. No trainable parameters.
- **tanh** Classic neural network activation function, though often replaced by relu family on modern networks.

- **sigmoid** - Classic neural network activation. Often used on output layer of a binary classifier.
- **hard_sigmoid** - Less computationally expensive variant of sigmoid.
- **exponential** - Exponential (base e) activation function.
- **linear** - Pass-through activation function. Usually used on the output layer of a regression neural network.

For more information about Keras activation functions refer to the following:

- Keras Activation Functions
- Activation Function Cheat Sheets

Advanced Activation Functions

Hyperparameters are not changed when the neural network trains. You, the network designer, must define the hyperparameters. The neural network learns regular parameters during neural network training. Neural network weights are the most common type of regular parameter. The "advanced

T81-558: Applications of Deep Neural Networks

Module 8: Kaggle Data Sets

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

Module 8 Material

- Part 8.1: Introduction to Kaggle [Video] [Notebook]
- Part 8.2: Building Ensembles with Scikit-Learn and Keras [Video] [Notebook]
- Part 8.3: How Should you Architect Your Keras Neural Network: Hyperparameters [Video] [Notebook]
- **Part 8.4: Bayesian Hyperparameter Optimization for Keras** [Video] [Notebook]
- Part 8.5: Current Semester's Kaggle [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
# Startup Google CoLab
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
```

```
    print("Note: not using Google CoLab")
    COLAB = False

# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return "{}:{:>02}:{:>05.2f}".format(h, m, s)
```

```
Note: using Google CoLab
```

Part 8.4: Bayesian Hyperparameter Optimization for Keras

Bayesian Hyperparameter Optimization is a method of finding hyperparameters more efficiently than a grid search. Because each candidate set of hyperparameters requires a retraining of the neural network, it is best to keep the number of candidate sets to a minimum. Bayesian Hyperparameter Optimization achieves this by training a model to predict good candidate sets of hyperparameters. [Cite:snoek2012practical]

- bayesian-optimization
- hyperopt
- spearmint

In [2]:

```
# Ignore useless W0819 warnings generated by TensorFlow 2.0.
# Hopefully can remove this ignore in the future.
# See https://github.com/tensorflow/tensorflow/issues/31308
import logging, os
logging.disable(logging.WARNING)
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"

import pandas as pd
from scipy.stats import zscore

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])
```

```
# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['age'] = zscore(df['age'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('product').drop('id')
x = df[x_columns].values
dummies = pd.get_dummies(df['product']) # Classification
products = dummies.columns
y = dummies.values
```

Now that we've preprocessed the data, we can begin the hyperparameter optimization. We start by creating a function that generates the model based on just three parameters. Bayesian optimization works on a vector of numbers, not on a problematic notion like how many layers and neurons are on each layer. To represent this complex neuron structure as a vector, we use several numbers to describe this structure.

- **dropout** - The dropout percent for each layer.
- **neuronPct** - What percent of our fixed 5,000 maximum number of neurons do we wish to use? This parameter specifies the total count of neurons in the entire network.
- **neuronShrink** - Neural networks usually start with more neurons on the first hidden layer and then decrease this count for additional layers. This percent specifies how much to shrink subsequent layers based on the previous layer. We stop adding more layers once we run out of neurons (the count specified by neuronPct).

These three numbers define the structure of the neural network. The commands in the below code show exactly how the

These three numbers define the structure of the neural network. The comments in the below code show exactly how the program constructs the network.

In [3]:

```
import pandas as pd
import os
import numpy as np
import time
import tensorflow.keras.initializers
import statistics
import tensorflow.keras
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout, InputLayer
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import ShuffleSplit
from tensorflow.keras.layers import LeakyReLU,PReLU
from tensorflow.keras.optimizers import Adam

def generate_model(dropout, neuronPct, neuronShrink):
    # We start with some percent of 5000 starting neurons on
    # the first hidden layer.
    neuronCount = int(neuronPct * 5000)

    # Construct neural network
    model = Sequential()

    # So long as there would have been at least 25 neurons and
    # fewer than 10
    # layers, create a new layer.
    layer = 0
    while neuronCount>25 and layer<10:
        # The first (0th) layer needs an input input_dim(neuronCount)
        if layer==0:
            model.add(Dense(neuronCount,
                input_dim=x.shape[1],
                activation=PReLU()))
        else:
            model.add(Dense(neuronCount, activation=PReLU()))
        layer += 1
```

```
        # Add dropout after each hidden layer
        model.add(Dropout(dropout))

        # Shrink neuron count for each layer
        neuronCount = neuronCount * neuronShrink

    model.add(Dense(y.shape[1],activation='softmax')) # Output
    return model
```

We can test this code to see how it creates a neural network based on three such parameters.

In [4]:
```
# Generate a model and see what the resulting structure looks like.
model = generate_model(dropout=0.2, neuronPct=0.1, neuronShrink=0.25)
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 500)               24500

 dropout (Dropout)           (None, 500)               0

 dense_1 (Dense)             (None, 125)               62750

 dropout_1 (Dropout)         (None, 125)               0

 dense_2 (Dense)             (None, 31)                3937

 dropout_2 (Dropout)         (None, 31)                0

 dense_3 (Dense)             (None, 7)                 224

=================================================================
Total params: 91,411
Trainable params: 91,411
Non-trainable params: 0
_________________________________________________________________
```

We will now create a function to evaluate the neural network using three such parameters. We use bootstrapping because one

training run might have "bad luck" with the assigned random weights. We use this function to train and then evaluate the neural network.

In [5]:

```
SPLITS = 2
EPOCHS = 500
PATIENCE = 10

def evaluate_network(dropout,learning_rate,neuronPct,neuronShrink):
    # Bootstrap

    # for Classification
    boot = StratifiedShuffleSplit(n_splits=SPLITS, test_size=0.1)
    # for Regression
    # boot = ShuffleSplit(n_splits=SPLITS, test_size=0.1)

    # Track progress
    mean_benchmark = []
    epochs_needed = []
    num = 0

    # Loop through samples
    for train, test in boot.split(x,df['product']):
        start_time = time.time()
        num+=1

        # Split train and test
        x_train = x[train]
        y_train = y[train]
        x_test = x[test]
        y_test = y[test]

        model = generate_model(dropout, neuronPct, neuronShrink)
        model.compile(loss='categorical_crossentropy',
                      optimizer=Adam(learning_rate=learning_rate))
        monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
        patience=PATIENCE, verbose=0, mode='auto',
                                restore_best_weights=True)

        # Train on the bootstrap sample
        model.fit(x_train,y_train,validation_data=(x_test,y_test),
                  callbacks=[monitor],verbose=0,epochs=EPOCHS)
        epochs = monitor.stopped_epoch
```

```
        epochs = monitor.stopped_epoch
        epochs_needed.append(epochs)

        # Predict on the out of boot (validation)
        pred = model.predict(x_test)

        # Measure this bootstrap's log loss
        y_compare = np.argmax(y_test,axis=1) # For log loss calculation
        score = metrics.log_loss(y_compare, pred)
        mean_benchmark.append(score)
        m1 = statistics.mean(mean_benchmark)
        m2 = statistics.mean(epochs_needed)
        mdev = statistics.pstdev(mean_benchmark)

        # Record this iteration
        time_took = time.time() - start_time

    tensorflow.keras.backend.clear_session()
    return (-m1)
```

You can try any combination of our three hyperparameters, plus the learning rate, to see how effective these four numbers are. Of course, our goal is not to manually choose different combinations of these four hyperparameters; we seek to automate.

In [6]:
```
print(evaluate_network(
    dropout=0.2,
    learning_rate=1e-3,
    neuronPct=0.2,
    neuronShrink=0.2))
```

```
-0.6668764846259546
```

First, we must install the Bayesian optimization package if we are in Colab.

In [7]:
```
# HIDE OUTPUT
!pip install bayesian-optimization
```

```
Requirement already satisfied: bayesian-optimization in /usr/local/lib/python3.7/dist-packages (1.2.0)
Requirement already satisfied: scipy>=0.14.0 in /usr/local/lib/python3.7/dist-packages (from bayesian-optimizat
ion) (1.4.1)
Requirement already satisfied: scikit-learn>=0.18.0 in /usr/local/lib/python3.7/dist-packages (from bayesian-op
```

```
timization) (1.0.2)
Requirement already satisfied: numpy>=1.9.0 in /usr/local/lib/python3.7/dist-packages (from bayesian-optimizati
on) (1.21.5)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.7/dist-packages (from scikit-lear
n>=0.18.0->bayesian-optimization) (3.1.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-packages (from scikit-learn>=0.1
8.0->bayesian-optimization) (1.1.0)
```

We will now automate this process. We define the bounds for each of these four hyperparameters and begin the Bayesian optimization. Once the program finishes, the best combination of hyperparameters found is displayed. The **optimize** function accepts two parameters that will significantly impact how long the process takes to complete:

- **n_iter** - How many steps of Bayesian optimization that you want to perform. The more steps, the more likely you will find a reasonable maximum.
- **init_points**: How many steps of random exploration that you want to perform. Random exploration can help by diversifying the exploration space.

In [8]:

```
from bayes_opt import BayesianOptimization
import time

# Supress NaN warnings
import warnings
warnings.filterwarnings("ignore",category =RuntimeWarning)

# Bounded region of parameter space
pbounds = {'dropout': (0.0, 0.499),
           'learning_rate': (0.0, 0.1),
           'neuronPct': (0.01, 1),
           'neuronShrink': (0.01, 1)
          }

optimizer = BayesianOptimization(
    f=evaluate_network,
    pbounds=pbounds,
    verbose=2,  # verbose = 1 prints only when a maximum
    # is observed, verbose = 0 is silent
    random_state=1,
)

start_time = time.time()
```

```
optimizer.maximize(init_points=10, n_iter=20,)
time_took = time.time() - start_time

print(f"Total runtime: {hms_string(time_took)}")
print(optimizer.max)
```

```
|   iter    |  target   |  dropout  | learni... | neuronPct | neuron... |
-------------------------------------------------------------------------
|  1        | -0.8092   |  0.2081   |  0.07203  |  0.01011  |  0.3093   |
|  2        | -0.7167   |  0.07323  |  0.009234 |  0.1944   |  0.3521   |
|  3        | -17.87    |  0.198    |  0.05388  |  0.425    |  0.6884   |
|  4        | -0.8022   |  0.102    |  0.08781  |  0.03711  |  0.6738   |
|  5        | -0.9209   |  0.2082   |  0.05587  |  0.149    |  0.2061   |
|  6        | -17.96    |  0.3996   |  0.09683  |  0.3203   |  0.6954   |
|  7        | -4.223    |  0.4373   |  0.08946  |  0.09419  |  0.04866  |
|  8        | -0.7025   |  0.08475  |  0.08781  |  0.1074   |  0.4269   |
|  9        | -8.666    |  0.478    |  0.05332  |  0.695    |  0.3224   |
|  10       | -9.785    |  0.3426   |  0.08346  |  0.02811  |  0.7526   |
|  11       | -4.881    |  0.0      |  0.0      |  0.01     |  0.01     |
|  12       | -21.59    |  0.2208   |  0.04135  |  0.5523   |  0.7468   |
|  13       | -1.819    |  0.0      |  0.0      |  1.0      |  0.01     |
|  14       | -33.33    |  0.01058  |  0.08079  |  0.9652   |  0.7051   |
|  15       | -1.418    |  0.4963   |  0.02476  |  0.9744   |  0.01896  |
|  16       | -1.876    |  0.1247   |  0.0      |  0.5781   |  0.01     |
|  17       | -1.898    |  0.0      |  0.0      |  0.01     |  1.0      |
|  18       | -17.7     |  0.1621   |  0.06358  |  0.4065   |  0.754    |
|  19       | -2.674    |  0.0      |  0.0      |  0.01     |  0.5464   |
|  20       | -1.931    |  0.499    |  0.0      |  0.5538   |  0.01     |
|  21       | -3.402    |  0.004722 |  0.05502  |  0.1704   |  0.483    |
|  22       | -2.8      |  0.08639  |  0.0838   |  0.04668  |  0.6864   |
|  23       | -20.98    |  0.1168   |  0.0447   |  0.5546   |  0.9497   |
|  24       | -4.565    |  0.2554   |  0.1      |  0.8418   |  0.01     |
|  25       | -4.724    |  0.0      |  0.1      |  0.3368   |  0.01     |
|  26       | -0.6956   |  0.2505   |  0.007623 |  0.01265  |  0.523    |
|  27       | -0.7139   |  0.2967   |  0.01162  |  0.3735   |  0.01708  |
|  28       | -2.145    |  0.499    |  0.0      |  0.3053   |  0.2207   |
|  29       | -2.069    |  0.0      |  0.0      |  0.4808   |  0.2473   |
|  30       | -0.7155   |  0.4082   |  0.01635  |  0.02488  |  0.1694   |
=========================================================================
Total runtime: 1:36:11.56
{'target': -0.6955536706512794, 'params': {'dropout': 0.2504561773412203, 'learning_rate': 0.007623234670914292
4, 'neuronPct': 0.012648791521811826, 'neuronShrink': 0.5229748831552032}}
```

As you can see, the algorithm performed 30 total iterations. This total iteration count includes ten random and 20 optimization iterations.

Open in Colab

T81-558: Applications of Deep Neural Networks

Module 8: Kaggle Data Sets

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

Module 8 Material

- Part 8.1: Introduction to Kaggle [Video] [Notebook]
- Part 8.2: Building Ensembles with Scikit-Learn and Keras [Video] [Notebook]
- Part 8.3: How Should you Architect Your Keras Neural Network: Hyperparameters [Video] [Notebook]
- Part 8.4: Bayesian Hyperparameter Optimization for Keras [Video] [Notebook]
- **Part 8.5: Current Semester's Kaggle** [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
# Start CoLab
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False

# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return "{}:{:>02}:{:>05.2f}".format(h, m, s)
```

```
Note: using Google CoLab
```

Part 8.5: Current Semester's Kaggle

Part 8.5: Current Semester's Kaggle

Kaggke competition site for current semester:

- Spring 2023 Kaggle Assignment

Previous Kaggle competition sites for this class (NOT this semester's assignment, feel free to use code):

- Fall 2022 Kaggle Assignment
- Spring 2022 Kaggle Assignment
- Fall 2021 Kaggle Assignment
- Spring 2021 Kaggle Assignment
- Fall 2020 Kaggle Assignment
- Spring 2020 Kaggle Assignment
- Fall 2019 Kaggle Assignment
- Spring 2019 Kaggle Assignment
- Fall 2018 Kaggle Assignment
- Spring 2018 Kaggle Assignment
- Fall 2017 Kaggle Assignment
- Spring 2017 Kaggle Assignment
- Fall 2016 Kaggle Assignment

Iris as a Kaggle Competition

If I used the Iris data as a Kaggle, I would give you the following three files:

- kaggle_iris_test.csv - The data that Kaggle will evaluate you on. It contains only input; you must provide answers. (contains x)
- kaggle_iris_train.csv - The data that you will use to train. (contains x and y)
- kaggle_iris_sample.csv - A sample submission for Kaggle. (contains x and y)

Important features of the Kaggle iris files (that differ from how we've previously seen files):

- The iris species is already index encoded.
- Your training data is in a separate file.
- You will load the test data to generate a submission file.

The following program generates a submission file for "Iris Kaggle". You can use it as a starting point for assignment 3.

In [2]:
```
import os
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow as tf
import numpy as np
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

df_train = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/datasets/"+\
    "kaggle_iris_train.csv", na_values=['NA','?'])

# Encode feature vector
df_train.drop('id', axis=1, inplace=True)

num_classes = len(df_train.groupby('species').species.nunique())

print("Number of classes: {}".format(num_classes))

# Convert to numpy - Classification
x = df_train[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
dummies = pd.get_dummies(df_train['species']) # Classification
species = dummies.columns
y = dummies.values

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=45)

# Train, with early stopping
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu'))
model.add(Dense(25))
model.add(Dense(y.shape[1],activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
                        patience=5, verbose=1, mode='auto',
                       restore_best_weights=True)

model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=0,epochs=1000)
```

```
Number of classes: 3
Restoring model weights from the end of the best epoch: 103.
Epoch 108: early stopping
```

Out[2]: <keras.callbacks.History at 0x7f05e7452710>

Now that we've trained the neural network, we can check its log loss.

In [3]:

```
from sklearn import metrics

# Calculate multi log loss error
pred = model.predict(x_test)
score = metrics.log_loss(y_test, pred)
print("Log loss score: {}".format(score))
```

```
Log loss score: 0.10988010508939623
```

Now we are ready to generate the Kaggle submission file. We will use the iris test data that does not contain a y target value. It is our job to predict this value and submit it to Kaggle.

In [4]:
```
# Generate Kaggle submit file

# Encode feature vector
df_test = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/datasets/"+\
    "kaggle_iris_test.csv", na_values=['NA','?'])

# Convert to numpy - Classification
ids = df_test['id']
df_test.drop('id', axis=1, inplace=True)
x = df_test[['sepal_l', 'sepal_w', 'petal_l', 'petal_w']].values
y = dummies.values

# Generate predictions
pred = model.predict(x)
#pred

# Create submission data set

df_submit = pd.DataFrame(pred)
df_submit.insert(0,'id',ids)
df_submit.columns = ['id','species-0','species-1','species-2']

# Write submit file locally
df_submit.to_csv("iris_submit.csv", index=False)

print(df_submit[:5])
```

```
    id  species-0  species-1  species-2
0  100   0.022300   0.777859   0.199841
1  101   0.001309   0.273849   0.724842
2  102   0.001153   0.319349   0.679498
3  103   0.958006   0.041989   0.000005
4  104   0.976932   0.023066   0.000002
```

MPG as a Kaggle Competition (Regression)

If the Auto MPG data were used as a Kaggle, you would be given the following three files:

- kaggle_mpg_test.csv - The data that Kaggle will evaluate you on. Contains only input, you must provide answers. (contains x)
- kaggle_mpg_train.csv - The data that you will use to train. (contains x and y)
- kaggle_mpg_sample.csv - A sample submission for Kaggle. (contains x and y)

Important features of the Kaggle iris files (that differ from how we've previously seen files):

The following program generates a submission file for "MPG Kaggle".

In [5]:
```
# HIDE OUTPUT
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
```

```
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/datasets/"+\
    "kaggle_auto_train.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

# Split into train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam')
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5,
                        verbose=1, mode='auto', restore_best_weights=Tr
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          verbose=2,callbacks=[monitor],epochs=1000)

# Predict
pred = model.predict(x_test)
```

```
Epoch 1/1000
9/9 - 1s - loss: 1797.5945 - val_loss: 1272.4421 - 1s/epoch - 144ms/step
Epoch 2/1000
9/9 - 0s - loss: 574.7726 - val_loss: 734.3082 - 92ms/epoch - 10ms/step
Epoch 3/1000
9/9 - 0s - loss: 487.3118 - val_loss: 446.3558 - 76ms/epoch - 8ms/step
Epoch 4/1000
9/9 - 0s - loss: 326.7128 - val_loss: 321.7191 - 96ms/epoch - 11ms/step
Epoch 5/1000
9/9 - 0s - loss: 294.8217 - val_loss: 271.3473 - 70ms/epoch - 8ms/step
Epoch 6/1000
9/9 - 0s - loss: 259.8376 - val_loss: 239.6796 - 116ms/epoch - 13ms/step
Epoch 7/1000
9/9 - 0s - loss: 250.4708 - val_loss: 227.4295 - 73ms/epoch - 8ms/step
Epoch 8/1000
9/9 - 0s - loss: 227.1252 - val_loss: 198.4167 - 125ms/epoch - 14ms/step
Epoch 9/1000
```

```
9/9 - 0s - loss: 225.6681 - val_loss: 195.5055 - 95ms/epoch - 11ms/step
Epoch 10/1000
9/9 - 0s - loss: 209.1198 - val_loss: 184.1092 - 121ms/epoch - 13ms/step
Epoch 11/1000
9/9 - 0s - loss: 195.4801 - val_loss: 176.0311 - 108ms/epoch - 12ms/step
Epoch 12/1000
9/9 - 0s - loss: 198.6493 - val_loss: 168.1613 - 163ms/epoch - 18ms/step
Epoch 13/1000
9/9 - 0s - loss: 198.5606 - val_loss: 196.0306 - 114ms/epoch - 13ms/step
Epoch 14/1000
9/9 - 0s - loss: 184.3067 - val_loss: 179.8450 - 99ms/epoch - 11ms/step
Epoch 15/1000
9/9 - 0s - loss: 178.6627 - val_loss: 148.1014 - 80ms/epoch - 9ms/step
Epoch 16/1000
9/9 - 0s - loss: 154.0201 - val_loss: 129.9253 - 74ms/epoch - 8ms/step
Epoch 17/1000
9/9 - 0s - loss: 145.2373 - val_loss: 124.0609 - 79ms/epoch - 9ms/step
Epoch 18/1000
9/9 - 0s - loss: 140.0318 - val_loss: 116.7844 - 86ms/epoch - 10ms/step
Epoch 19/1000
9/9 - 0s - loss: 135.1688 - val_loss: 115.0745 - 136ms/epoch - 15ms/step
Epoch 20/1000
9/9 - 0s - loss: 132.8391 - val_loss: 106.9831 - 169ms/epoch - 19ms/step
Epoch 21/1000
9/9 - 0s - loss: 123.6673 - val_loss: 105.7211 - 95ms/epoch - 11ms/step
Epoch 22/1000
9/9 - 0s - loss: 123.7169 - val_loss: 99.6713 - 112ms/epoch - 12ms/step
Epoch 23/1000
9/9 - 0s - loss: 118.0815 - val_loss: 96.0683 - 150ms/epoch - 17ms/step
Epoch 24/1000
9/9 - 0s - loss: 114.6363 - val_loss: 99.1486 - 153ms/epoch - 17ms/step
Epoch 25/1000
9/9 - 0s - loss: 112.3965 - val_loss: 93.8642 - 180ms/epoch - 20ms/step
Epoch 26/1000
9/9 - 0s - loss: 111.2470 - val_loss: 88.3417 - 139ms/epoch - 15ms/step
Epoch 27/1000
9/9 - 0s - loss: 107.8639 - val_loss: 86.7927 - 135ms/epoch - 15ms/step
Epoch 28/1000
9/9 - 0s - loss: 103.0426 - val_loss: 89.0441 - 101ms/epoch - 11ms/step
Epoch 29/1000
9/9 - 0s - loss: 110.6277 - val_loss: 82.4294 - 159ms/epoch - 18ms/step
Epoch 30/1000
9/9 - 0s - loss: 100.3681 - val_loss: 90.8037 - 82ms/epoch - 9ms/step
Epoch 31/1000
9/9 - 0s - loss: 105.4711 - val_loss: 79.2106 - 76ms/epoch - 8ms/step
Epoch 32/1000
9/9 - 0s - loss: 98.7603 - val_loss: 79.9620 - 73ms/epoch - 8ms/step
Epoch 33/1000
9/9 - 0s - loss: 94.7678 - val_loss: 76.8616 - 78ms/epoch - 9ms/step
Epoch 34/1000
9/9 - 0s - loss: 93.8199 - val_loss: 77.0823 - 76ms/epoch - 8ms/step
Epoch 35/1000
9/9 - 0s - loss: 94.8746 - val_loss: 73.9967 - 62ms/epoch - 7ms/step
Epoch 36/1000
9/9 - 0s - loss: 95.3178 - val_loss: 73.0059 - 60ms/epoch - 7ms/step
Epoch 37/1000
9/9 - 0s - loss: 91.1315 - val_loss: 80.8389 - 57ms/epoch - 6ms/step
Epoch 38/1000
9/9 - 0s - loss: 96.4810 - val_loss: 77.8854 - 59ms/epoch - 7ms/step
Epoch 39/1000
```

```
9/9 - 0s - loss: 91.1039 - val_loss: 69.9539 - 40ms/epoch - 4ms/step
Epoch 40/1000
9/9 - 0s - loss: 86.9596 - val_loss: 69.3511 - 43ms/epoch - 5ms/step
Epoch 41/1000
9/9 - 0s - loss: 87.6142 - val_loss: 70.1390 - 57ms/epoch - 6ms/step
Epoch 42/1000
9/9 - 0s - loss: 88.0185 - val_loss: 73.6168 - 38ms/epoch - 4ms/step
Epoch 43/1000
9/9 - 0s - loss: 92.8655 - val_loss: 67.5213 - 38ms/epoch - 4ms/step
Epoch 44/1000
9/9 - 0s - loss: 88.5278 - val_loss: 69.9708 - 59ms/epoch - 7ms/step
Epoch 45/1000
9/9 - 0s - loss: 82.9339 - val_loss: 70.3786 - 39ms/epoch - 4ms/step
Epoch 46/1000
9/9 - 0s - loss: 81.7092 - val_loss: 63.3550 - 59ms/epoch - 7ms/step
Epoch 47/1000
9/9 - 0s - loss: 81.1514 - val_loss: 78.7681 - 59ms/epoch - 7ms/step
Epoch 48/1000
9/9 - 0s - loss: 99.3562 - val_loss: 62.8894 - 64ms/epoch - 7ms/step
Epoch 49/1000
9/9 - 0s - loss: 96.8292 - val_loss: 67.8047 - 55ms/epoch - 6ms/step
Epoch 50/1000
9/9 - 0s - loss: 88.7995 - val_loss: 67.5249 - 56ms/epoch - 6ms/step
Epoch 51/1000
9/9 - 0s - loss: 80.6064 - val_loss: 96.2975 - 58ms/epoch - 6ms/step
Epoch 52/1000
9/9 - 0s - loss: 95.2732 - val_loss: 62.4323 - 39ms/epoch - 4ms/step
Epoch 53/1000
9/9 - 0s - loss: 75.1992 - val_loss: 64.0174 - 39ms/epoch - 4ms/step
Epoch 54/1000
9/9 - 0s - loss: 75.5173 - val_loss: 57.8594 - 40ms/epoch - 4ms/step
Epoch 55/1000
9/9 - 0s - loss: 72.6369 - val_loss: 56.2216 - 47ms/epoch - 5ms/step
Epoch 56/1000
9/9 - 0s - loss: 72.8636 - val_loss: 55.3956 - 54ms/epoch - 6ms/step
Epoch 57/1000
9/9 - 0s - loss: 69.0251 - val_loss: 70.7940 - 56ms/epoch - 6ms/step
Epoch 58/1000
9/9 - 0s - loss: 75.8152 - val_loss: 63.7728 - 37ms/epoch - 4ms/step
Epoch 59/1000
9/9 - 0s - loss: 71.6866 - val_loss: 59.5908 - 41ms/epoch - 5ms/step
Epoch 60/1000
9/9 - 0s - loss: 69.3349 - val_loss: 52.7848 - 38ms/epoch - 4ms/step
Epoch 61/1000
9/9 - 0s - loss: 67.8410 - val_loss: 53.5977 - 54ms/epoch - 6ms/step
Epoch 62/1000
9/9 - 0s - loss: 68.4640 - val_loss: 53.6664 - 39ms/epoch - 4ms/step
Epoch 63/1000
9/9 - 0s - loss: 63.7229 - val_loss: 52.4224 - 44ms/epoch - 5ms/step
Epoch 64/1000
9/9 - 0s - loss: 69.8485 - val_loss: 59.1973 - 53ms/epoch - 6ms/step
Epoch 65/1000
9/9 - 0s - loss: 75.7193 - val_loss: 70.1342 - 37ms/epoch - 4ms/step
Epoch 66/1000
9/9 - 0s - loss: 87.7418 - val_loss: 55.3687 - 38ms/epoch - 4ms/step
Epoch 67/1000
9/9 - 0s - loss: 72.8599 - val_loss: 52.9028 - 44ms/epoch - 5ms/step
Epoch 68/1000
9/9 - 0s - loss: 69.9528 - val_loss: 49.9109 - 38ms/epoch - 4ms/step
Epoch 69/1000
9/9 - 0s - loss: 62.7782 - val_loss: 46.6361 - 39ms/epoch - 4ms/step
```

```
Epoch 70/1000
9/9 - 0s - loss: 58.4024 - val_loss: 50.8190 - 38ms/epoch - 4ms/step
Epoch 71/1000
9/9 - 0s - loss: 63.5687 - val_loss: 46.6161 - 44ms/epoch - 5ms/step
Epoch 72/1000
9/9 - 0s - loss: 65.9290 - val_loss: 47.1278 - 40ms/epoch - 4ms/step
Epoch 73/1000
9/9 - 0s - loss: 74.9235 - val_loss: 61.1282 - 42ms/epoch - 5ms/step
Epoch 74/1000
9/9 - 0s - loss: 63.6773 - val_loss: 45.0233 - 39ms/epoch - 4ms/step
Epoch 75/1000
9/9 - 0s - loss: 55.8287 - val_loss: 59.8986 - 41ms/epoch - 5ms/step
Epoch 76/1000
9/9 - 0s - loss: 58.9969 - val_loss: 52.0535 - 39ms/epoch - 4ms/step
Epoch 77/1000
9/9 - 0s - loss: 60.7104 - val_loss: 43.0530 - 46ms/epoch - 5ms/step
Epoch 78/1000
9/9 - 0s - loss: 59.7358 - val_loss: 45.3669 - 41ms/epoch - 5ms/step
Epoch 79/1000
9/9 - 0s - loss: 60.9792 - val_loss: 40.7967 - 41ms/epoch - 5ms/step
Epoch 80/1000
9/9 - 0s - loss: 58.0294 - val_loss: 49.0612 - 42ms/epoch - 5ms/step
Epoch 81/1000
9/9 - 0s - loss: 57.6733 - val_loss: 41.7604 - 44ms/epoch - 5ms/step
Epoch 82/1000
9/9 - 0s - loss: 50.3309 - val_loss: 39.1461 - 38ms/epoch - 4ms/step
Epoch 83/1000
9/9 - 0s - loss: 54.2316 - val_loss: 40.8561 - 36ms/epoch - 4ms/step
Epoch 84/1000
9/9 - 0s - loss: 66.4084 - val_loss: 38.1869 - 60ms/epoch - 7ms/step
Epoch 85/1000
9/9 - 0s - loss: 50.0778 - val_loss: 37.8852 - 56ms/epoch - 6ms/step
Epoch 86/1000
9/9 - 0s - loss: 47.0763 - val_loss: 37.3743 - 39ms/epoch - 4ms/step
Epoch 87/1000
9/9 - 0s - loss: 46.1752 - val_loss: 45.8444 - 45ms/epoch - 5ms/step
Epoch 88/1000
9/9 - 0s - loss: 49.4047 - val_loss: 37.3778 - 40ms/epoch - 4ms/step
Epoch 89/1000
9/9 - 0s - loss: 46.5478 - val_loss: 36.2859 - 38ms/epoch - 4ms/step
Epoch 90/1000
9/9 - 0s - loss: 44.7429 - val_loss: 47.2213 - 38ms/epoch - 4ms/step
Epoch 91/1000
9/9 - 0s - loss: 49.7726 - val_loss: 52.5501 - 42ms/epoch - 5ms/step
Epoch 92/1000
9/9 - 0s - loss: 53.5449 - val_loss: 62.3078 - 57ms/epoch - 6ms/step
Epoch 93/1000
9/9 - 0s - loss: 54.7558 - val_loss: 51.2010 - 43ms/epoch - 5ms/step
Epoch 94/1000
Restoring model weights from the end of the best epoch: 89.
9/9 - 0s - loss: 52.3631 - val_loss: 42.2640 - 69ms/epoch - 8ms/step
Epoch 94: early stopping
```

Now that we've trained the neural network, we can check its RMSE error.

In [6]:

```
import numpy as np

# Measure RMSE error.  RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y_test))
```

```
print("Final score (RMSE): {}".format(score))
```

```
Final score (RMSE): 6.023776405947501
```

Now we are ready to generate the Kaggle submission file. We will use the MPG test data that does not contain a y target value. It is our job to predict this value and submit it to Kaggle.

In [7]:

```
import pandas as pd

# Generate Kaggle submit file

# Encode feature vector
df_test = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/datasets/"+\
    "kaggle_auto_test.csv", na_values=['NA','?'])

# Convert to numpy - regression
ids = df_test['id']
df_test.drop('id', axis=1, inplace=True)

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

x = df_test[['cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin']].values

# Generate predictions
pred = model.predict(x)
#pred
```