Open in Colab

T81-558: Applications of Deep Neural Networks

Module 4: Training for Tabular Data

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

Module 4 Material

- Part 4.1: Encoding a Feature Vector for Keras Deep Learning [Video] [Notebook]
- Part 4.2: Keras Multiclass Classification for Deep Neural Networks with ROC and AUC [Video] [Notebook]
- **Part 4.3: Keras Regression for Deep Neural Networks with RMSE** [Video] [Notebook]
- Part 4.4: Backpropagation, Nesterov Momentum, and ADAM Neural Network Training [Video] [Notebook]
- Part 4.5: Neural Network RMSE and Log Loss Error Calculation from Scratch [Video] [Notebook]

Google CoLab Instructions

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

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

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

Part 4.3: Keras Regression for Deep Neural Networks with RMSE

We evaluate regression results differently than classification. Consider the following code that trains a neural network for regression on the data set **jh-simple-dataset.csv**. We begin by preparing the data set.

In [2]:
```
import pandas as pd
from scipy.stats import zscore
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# 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)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', 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['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values

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

Next, we create a neural network to fit the data we just loaded.

In [3]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping

# 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')
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=2,epochs=1000)
```

```
Train on 1500 samples, validate on 500 samples
Epoch 1/1000
1500/1500 - 1s - loss: 1905.4454 - val_loss: 1628.1341
Epoch 2/1000
1500/1500 - 0s - loss: 1331.4213 - val_loss: 889.0575
Epoch 3/1000
1500/1500 - 0s - loss: 554.8426 - val_loss: 303.7261
Epoch 4/1000
1500/1500 - 0s - loss: 276.2087 - val_loss: 241.2495
Epoch 5/1000
1500/1500 - 0s - loss: 232.2832 - val_loss: 208.2143
Epoch 6/1000
1500/1500 - 0s - loss: 198.5331 - val_loss: 179.5262
Epoch 7/1000
1500/1500 - 0s - loss: 169.0791 - val_loss: 154.5270
Epoch 8/1000
1500/1500 - 0s - loss: 144.1286 - val_loss: 132.8691
Epoch 9/1000
1500/1500 - 0s - loss: 122.9873 - val_loss: 115.0928
Epoch 10/1000
1500/1500 - 0s - loss: 104.7249 - val_loss: 98.7375
Epoch 11/1000
1500/1500 - 0s - loss: 89.8292 - val_loss: 86.2749
Epoch 12/1000
1500/1500 - 0s - loss: 77.3071 - val_loss: 75.0022
Epoch 13/1000
1500/1500 - 0s - loss: 67.0604 - val_loss: 66.1396
Epoch 14/1000
1500/1500 - 0s - loss: 58.9584 - val_loss: 58.4367
Epoch 15/1000
1500/1500 - 0s - loss: 51.2491 - val_loss: 52.7136
Epoch 16/1000
1500/1500 - 0s - loss: 45.1765 - val_loss: 46.5179
Epoch 17/1000
1500/1500 - 0s - loss: 39.8843 - val_loss: 41.3721
Epoch 18/1000
1500/1500 - 0s - loss: 35.1468 - val_loss: 37.2132
Epoch 19/1000
1500/1500 - 0s - loss: 31.1755 - val_loss: 33.0697
Epoch 20/1000
1500/1500 - 0s - loss: 27.6307 - val_loss: 30.3131
Epoch 21/1000
1500/1500 - 0s - loss: 24.8457 - val_loss: 26.9474
Epoch 22/1000
1500/1500 - 0s - loss: 22.4056 - val_loss: 24.3656
Epoch 23/1000
1500/1500 - 0s - loss: 20.3071 - val_loss: 22.1642
Epoch 24/1000
1500/1500 - 0s - loss: 18.5446 - val_loss: 20.4782
Epoch 25/1000
1500/1500 - 0s - loss: 17.1571 - val_loss: 18.8670
Epoch 26/1000
1500/1500 - 0s - loss: 15.9407 - val_loss: 17.6862
Epoch 27/1000
1500/1500 - 0s - loss: 14.9866 - val_loss: 16.5275
Epoch 28/1000
```

```
1500/1500 - 0s - loss: 14.1251 - val_loss: 15.6342
Epoch 29/1000
1500/1500 - 0s - loss: 13.4655 - val_loss: 14.8625
Epoch 30/1000
1500/1500 - 0s - loss: 12.8994 - val_loss: 14.2826
Epoch 31/1000
1500/1500 - 0s - loss: 12.5566 - val_loss: 13.6121
Epoch 32/1000
1500/1500 - 0s - loss: 12.0077 - val_loss: 13.3087
Epoch 33/1000
1500/1500 - 0s - loss: 11.5357 - val_loss: 12.6593
Epoch 34/1000
1500/1500 - 0s - loss: 11.2365 - val_loss: 12.1849
Epoch 35/1000
1500/1500 - 0s - loss: 10.8074 - val_loss: 11.9388
Epoch 36/1000
1500/1500 - 0s - loss: 10.5593 - val_loss: 11.4006
Epoch 37/1000
1500/1500 - 0s - loss: 10.2093 - val_loss: 10.9751
Epoch 38/1000
1500/1500 - 0s - loss: 9.8386 - val_loss: 10.8651
Epoch 39/1000
1500/1500 - 0s - loss: 9.5938 - val_loss: 10.5728
Epoch 40/1000
1500/1500 - 0s - loss: 9.1488 - val_loss: 9.8661
Epoch 41/1000
1500/1500 - 0s - loss: 8.8920 - val_loss: 9.5228
Epoch 42/1000
1500/1500 - 0s - loss: 8.5156 - val_loss: 9.1506
Epoch 43/1000
1500/1500 - 0s - loss: 8.2628 - val_loss: 8.9486
Epoch 44/1000
1500/1500 - 0s - loss: 7.9219 - val_loss: 8.5034
Epoch 45/1000
1500/1500 - 0s - loss: 7.7077 - val_loss: 8.0760
Epoch 46/1000
1500/1500 - 0s - loss: 7.3165 - val_loss: 7.6620
Epoch 47/1000
1500/1500 - 0s - loss: 7.0259 - val_loss: 7.4933
Epoch 48/1000
1500/1500 - 0s - loss: 6.7422 - val_loss: 7.0583
Epoch 49/1000
1500/1500 - 0s - loss: 6.5163 - val_loss: 6.8024
Epoch 50/1000
1500/1500 - 0s - loss: 6.2633 - val_loss: 7.3045
Epoch 51/1000
1500/1500 - 0s - loss: 6.0029 - val_loss: 6.2712
Epoch 52/1000
1500/1500 - 0s - loss: 5.6791 - val_loss: 5.9342
Epoch 53/1000
1500/1500 - 0s - loss: 5.4798 - val_loss: 6.0110
Epoch 54/1000
1500/1500 - 0s - loss: 5.2115 - val_loss: 5.3928
Epoch 55/1000
1500/1500 - 0s - loss: 4.9592 - val_loss: 5.2215
Epoch 56/1000
```

```
1500/1500 - 0s - loss: 4.7189 - val_loss: 5.0103
Epoch 57/1000
1500/1500 - 0s - loss: 4.4683 - val_loss: 4.7098
Epoch 58/1000
1500/1500 - 0s - loss: 4.2650 - val_loss: 4.5259
Epoch 59/1000
1500/1500 - 0s - loss: 4.0953 - val_loss: 4.4263
Epoch 60/1000
1500/1500 - 0s - loss: 3.8027 - val_loss: 4.1103
Epoch 61/1000
1500/1500 - 0s - loss: 3.5759 - val_loss: 3.7770
Epoch 62/1000
1500/1500 - 0s - loss: 3.3755 - val_loss: 3.5737
Epoch 63/1000
1500/1500 - 0s - loss: 3.1781 - val_loss: 3.4833
Epoch 64/1000
1500/1500 - 0s - loss: 3.0001 - val_loss: 3.2246
Epoch 65/1000
1500/1500 - 0s - loss: 2.7691 - val_loss: 3.1021
Epoch 66/1000
1500/1500 - 0s - loss: 2.6227 - val_loss: 2.8215
Epoch 67/1000
1500/1500 - 0s - loss: 2.4682 - val_loss: 2.7528
Epoch 68/1000
1500/1500 - 0s - loss: 2.3243 - val_loss: 2.5394
Epoch 69/1000
1500/1500 - 0s - loss: 2.1664 - val_loss: 2.3886
Epoch 70/1000
1500/1500 - 0s - loss: 2.0377 - val_loss: 2.2536
Epoch 71/1000
1500/1500 - 0s - loss: 1.8845 - val_loss: 2.2354
Epoch 72/1000
1500/1500 - 0s - loss: 1.7931 - val_loss: 2.0831
Epoch 73/1000
1500/1500 - 0s - loss: 1.6889 - val_loss: 1.8866
Epoch 74/1000
1500/1500 - 0s - loss: 1.5820 - val_loss: 1.7964
Epoch 75/1000
1500/1500 - 0s - loss: 1.5085 - val_loss: 1.7138
Epoch 76/1000
1500/1500 - 0s - loss: 1.4159 - val_loss: 1.6468
Epoch 77/1000
1500/1500 - 0s - loss: 1.3606 - val_loss: 1.5906
Epoch 78/1000
1500/1500 - 0s - loss: 1.2652 - val_loss: 1.5063
Epoch 79/1000
1500/1500 - 0s - loss: 1.1937 - val_loss: 1.4506
Epoch 80/1000
1500/1500 - 0s - loss: 1.1180 - val_loss: 1.4817
Epoch 81/1000
1500/1500 - 0s - loss: 1.1412 - val_loss: 1.2800
Epoch 82/1000
1500/1500 - 0s - loss: 1.0385 - val_loss: 1.2412
Epoch 83/1000
1500/1500 - 0s - loss: 0.9846 - val_loss: 1.1891
Epoch 84/1000
```

```
1500/1500 - 0s - loss: 0.9937 - val_loss: 1.1322
Epoch 85/1000
1500/1500 - 0s - loss: 0.8915 - val_loss: 1.0847
Epoch 86/1000
1500/1500 - 0s - loss: 0.8562 - val_loss: 1.1110
Epoch 87/1000
1500/1500 - 0s - loss: 0.8468 - val_loss: 1.0686
Epoch 88/1000
1500/1500 - 0s - loss: 0.7947 - val_loss: 0.9805
Epoch 89/1000
1500/1500 - 0s - loss: 0.7807 - val_loss: 0.9463
Epoch 90/1000
1500/1500 - 0s - loss: 0.7502 - val_loss: 0.9965
Epoch 91/1000
1500/1500 - 0s - loss: 0.7529 - val_loss: 0.9532
Epoch 92/1000
1500/1500 - 0s - loss: 0.6857 - val_loss: 0.8712
Epoch 93/1000
1500/1500 - 0s - loss: 0.6717 - val_loss: 0.8498
Epoch 94/1000
1500/1500 - 0s - loss: 0.6869 - val_loss: 0.8518
Epoch 95/1000
1500/1500 - 0s - loss: 0.6626 - val_loss: 0.8275
Epoch 96/1000
1500/1500 - 0s - loss: 0.6308 - val_loss: 0.7850
Epoch 97/1000
1500/1500 - 0s - loss: 0.6056 - val_loss: 0.7708
Epoch 98/1000
1500/1500 - 0s - loss: 0.5991 - val_loss: 0.7643
Epoch 99/1000
1500/1500 - 0s - loss: 0.6102 - val_loss: 0.8104
Epoch 100/1000
1500/1500 - 0s - loss: 0.5647 - val_loss: 0.7227
Epoch 101/1000
1500/1500 - 0s - loss: 0.5474 - val_loss: 0.7107
Epoch 102/1000
1500/1500 - 0s - loss: 0.5395 - val_loss: 0.6847
Epoch 103/1000
1500/1500 - 0s - loss: 0.5350 - val_loss: 0.7383
Epoch 104/1000
1500/1500 - 0s - loss: 0.5551 - val_loss: 0.6698
Epoch 105/1000
1500/1500 - 0s - loss: 0.5032 - val_loss: 0.6520
Epoch 106/1000
1500/1500 - 0s - loss: 0.5418 - val_loss: 0.7518
Epoch 107/1000
1500/1500 - 0s - loss: 0.4949 - val_loss: 0.6307
Epoch 108/1000
1500/1500 - 0s - loss: 0.5166 - val_loss: 0.6741
Epoch 109/1000
1500/1500 - 0s - loss: 0.4992 - val_loss: 0.6195
Epoch 110/1000
1500/1500 - 0s - loss: 0.4610 - val_loss: 0.6268
Epoch 111/1000
1500/1500 - 0s - loss: 0.4554 - val_loss: 0.5956
Epoch 112/1000
```

```
1500/1500 - 0s - loss: 0.4704 - val_loss: 0.5977
Epoch 113/1000
1500/1500 - 0s - loss: 0.4687 - val_loss: 0.5736
Epoch 114/1000
1500/1500 - 0s - loss: 0.4497 - val_loss: 0.5817
Epoch 115/1000
1500/1500 - 0s - loss: 0.4326 - val_loss: 0.5833
Epoch 116/1000
1500/1500 - 0s - loss: 0.4181 - val_loss: 0.5738
Epoch 117/1000
1500/1500 - 0s - loss: 0.4252 - val_loss: 0.5688
Epoch 118/1000
1500/1500 - 0s - loss: 0.4675 - val_loss: 0.5680
Epoch 119/1000
1500/1500 - 0s - loss: 0.4328 - val_loss: 0.5463
Epoch 120/1000
1500/1500 - 0s - loss: 0.4091 - val_loss: 0.5912
Epoch 121/1000
1500/1500 - 0s - loss: 0.4047 - val_loss: 0.5459
Epoch 122/1000
1500/1500 - 0s - loss: 0.4456 - val_loss: 0.5509
Epoch 123/1000
1500/1500 - 0s - loss: 0.4081 - val_loss: 0.5540
Epoch 124/1000
Restoring model weights from the end of the best epoch.
1500/1500 - 0s - loss: 0.4353 - val_loss: 0.5538
Epoch 00124: early stopping
```

Out[3]:
```
<tensorflow.python.keras.callbacks.History at 0x1a40e6b0d0>
```

Mean Square Error

The mean square error (MSE) is the sum of the squared differences between the prediction ($\hat{y}$) and the expected (y). MSE values are not of a particular unit. If an MSE value has decreased for a model, that is good. However, beyond this, there is not much more you can determine. We seek to achieve low MSE values. The following equation demonstrates how to calculate MSE.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} \left(\hat{y}_i - y_i\right)^2$$

The following code calculates the MSE on the predictions from the neural network.

In [4]:
```python
from sklearn import metrics

# Predict
pred = model.predict(x_test)

# Measure MSE error.
score = metrics.mean_squared_error(pred,y_test)
print("Final score (MSE): {}".format(score))
```

```
Final score (MSE): 0.5463447829677607
```

Root Mean Square Error

The root mean square (RMSE) is essentially the square root of the MSE. Because of this, the RMSE error is in the same units as the training data outcome. We desire Low RMSE values. The following equation calculates RMSE.

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right)^2}$$

In [5]:
```
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): 0.7391513938076291
```

Lift Chart

We often visualize the results of regression with a lift chart. To generate a lift chart, perform the following activities:

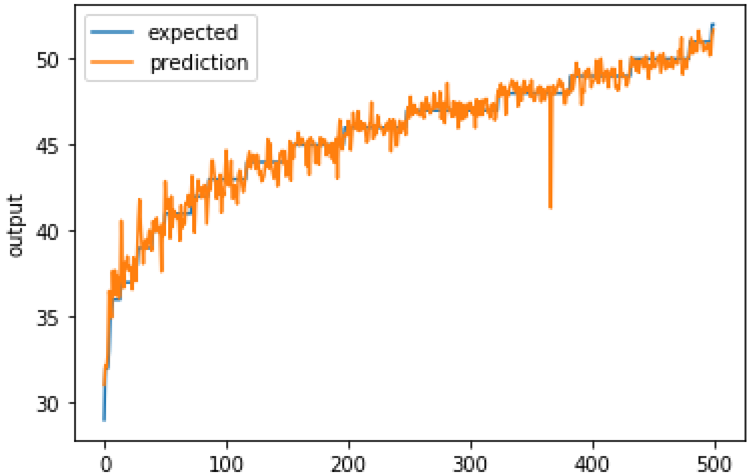
- Sort the data by expected output and plot these values.
- For every point on the x-axis, plot that same data point's predicted value in another color.
- The x-axis is just 0 to 100% of the dataset. The expected always starts low and ends high.
- The y-axis is ranged according to the values predicted.

You can interpret the lift chart as follows:

- The expected and predict lines should be close. Notice where one is above the other.
- The below chart is the most accurate for lower ages.

In [7]:
```
# Regression chart.
def chart_regression(pred, y, sort=True):
    t = pd.DataFrame({'pred': pred, 'y': y.flatten()})
    if sort:
        t.sort_values(by=['y'], inplace=True)
    plt.plot(t['y'].tolist(), label='expected')
    plt.plot(t['pred'].tolist(), label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()
```

```python
# Plot the chart
chart_regression(pred.flatten(),y_test)
```



In [ ]: