

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.

```
In [28]: 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 3.2: Introduction to Tensorflow and Keras

TensorFlow [\[Cite:GoogleTensorFlow\]](#) is an open-source software library for machine learning in various kinds of perceptual and language understanding tasks. It is currently used for research and production by different teams in many

commercial Google products, such as speech recognition, Gmail, Google Photos, and search, many of which had previously used its predecessor DistBelief. TensorFlow was originally developed by the Google Brain team for Google's research and production purposes and later released under the Apache 2.0 open source license on November 9, 2015.

- [TensorFlow Homepage](#)
- [TensorFlow GitHub](#)
- [TensorFlow Google Groups Support](#)
- [TensorFlow Google Groups Developer Discussion](#)
- [TensorFlow FAQ](#)

Why TensorFlow

- Supported by Google
- Works well on Windows, Linux, and Mac
- Excellent GPU support
- Python is an easy to learn programming language
- Python is extremely popular in the data science community

Deep Learning Tools

TensorFlow is not the only game in town. The biggest competitor to TensorFlow/Keras is PyTorch. Listed below are some of the deep learning toolkits actively being supported:

- **TensorFlow** - Google's deep learning API. The focus of this class, along with Keras.
- **Keras** - Acts as a higher-level to Tensorflow.
- **PyTorch** - PyTorch is an open-source machine learning library based on the Torch library, used for computer vision and natural language applications processing. Facebook's AI Research lab primarily develops PyTorch.

Other deep learning tools:

- **Deeplearning4J** - Java-based. Supports all major platforms. GPU support in Java!
- **H2O** - Java-based.

In my opinion, the two primary Python libraries for deep learning are PyTorch and Keras. Generally, PyTorch requires more lines of code to perform the deep learning applications presented in this course. This trait of PyTorch gives Keras an easier learning curve than PyTorch. However, if you are creating entirely new

neural network structures in a research setting, PyTorch can make for easier access to some of the low-level internals of deep learning.

Using TensorFlow Directly

Most of the time in the course, we will communicate with TensorFlow using Keras [Cite:franccois2017deep], which allows you to specify the number of hidden layers and create the neural network. TensorFlow is a low-level mathematics API, similar to [Numpy](#). However, unlike Numpy, TensorFlow is built for deep learning. TensorFlow compiles these compute graphs into highly efficient C++/[CUDA](#) code.

TensorFlow Linear Algebra Examples

TensorFlow is a library for linear algebra. Keras is a higher-level abstraction for neural networks that you build upon TensorFlow. In this section, I will demonstrate some basic linear algebra that directly employs TensorFlow and does not use Keras. First, we will see how to multiply a row and column matrix.

```
In [29]: import tensorflow as tf

# Create a Constant op that produces a 1x2 matrix. The op is
# added as a node to the default graph.
#
# The value returned by the constructor represents the output
# of the Constant op.
matrix1 = tf.constant([[3., 3.]])

# Create another Constant that produces a 2x1 matrix.
matrix2 = tf.constant([[2.],[2.]])

# Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
# The returned value, 'product', represents the result of the matrix
# multiplication.
product = tf.matmul(matrix1, matrix2)

print(product)
print(float(product))
```

```
tf.Tensor([[12.]], shape=(1, 1), dtype=float32)
12.0
```

This example multiplied two TensorFlow constant tensors. Next, we will see how to subtract a constant from a variable.

```
In [30]: import tensorflow as tf

x = tf.Variable([1.0, 2.0])
a = tf.constant([3.0, 3.0])
```

```
# Add an op to subtract 'a' from 'x'. Run it and print the result
sub = tf.subtract(x, a)
print(sub)
print(sub.numpy())
# ==> [-2. -1.]
```

```
tf.Tensor([-2. -1.], shape=(2,), dtype=float32)
[-2. -1.]
```

Of course, variables are only useful if their values can be changed. The program can accomplish this change in value by calling the assign function.

```
In [31]: x.assign([4.0, 6.0])
```

```
Out[31]: <tf.Variable 'UnreadVariable' shape=(2,) dtype=float32, numpy=array([4.,
6.], dtype=float32)>
```

The program can now perform the subtraction with this new value.

```
In [32]: sub = tf.subtract(x, a)
print(sub)
print(sub.numpy())
```

```
tf.Tensor([1. 3.], shape=(2,), dtype=float32)
[1. 3.]
```

In the next section, we will see a TensorFlow example that has nothing to do with neural networks.

TensorFlow Mandelbrot Set Example

Next, we examine another example where we use TensorFlow directly. To demonstrate that TensorFlow is mathematical and does not only provide neural networks, we will also first use it for a non-machine learning rendering task. The code presented here can render a [Mandelbrot set](#).

```
In [33]: import tensorflow as tf
import numpy as np

import PIL.Image
from io import BytesIO
from IPython.display import Image, display

def render(a):
    a_cyclic = (a*0.3).reshape(list(a.shape)+[1])
    img = np.concatenate([10+20*np.cos(a_cyclic),
                          30+50*np.sin(a_cyclic),
                          155-80*np.cos(a_cyclic)], 2)

    img[a==a.max()] = 0
    a = img
    a = np.uint8(np.clip(a, 0, 255))
    f = BytesIO()
    return PIL.Image.fromarray(a)
```

```

#@tf.function
def mandelbrot_helper(grid_c, current_values, counts,cycles):

    for i in range(cycles):
        temp = current_values*current_values + grid_c
        not_diverged = tf.abs(temp) < 4
        current_values.assign(temp),
        counts.assign_add(tf.cast(not_diverged, tf.float32))

def mandelbrot(render_size,center,zoom,cycles):
    f = zoom/render_size[0]
    real_start = center[0]-(render_size[0]/2)*f
    real_end = real_start + render_size[0]*f
    imag_start = center[1]-(render_size[1]/2)*f
    imag_end = imag_start + render_size[1]*f

    real_range = tf.range(real_start,real_end,f,dtype=tf.float64)
    imag_range = tf.range(imag_start,imag_end,f,dtype=tf.float64)
    real, imag = tf.meshgrid(real_range,imag_range)
    grid_c = tf.constant(tf.complex(real, imag))
    current_values = tf.Variable(grid_c)
    counts = tf.Variable(tf.zeros_like(grid_c, tf.float32))



    mandelbrot_helper(grid_c, current_values,counts,cycles)
    return counts.numpy()

```

With the above code defined, we can now calculate and render a Mandelbrot plot.

```

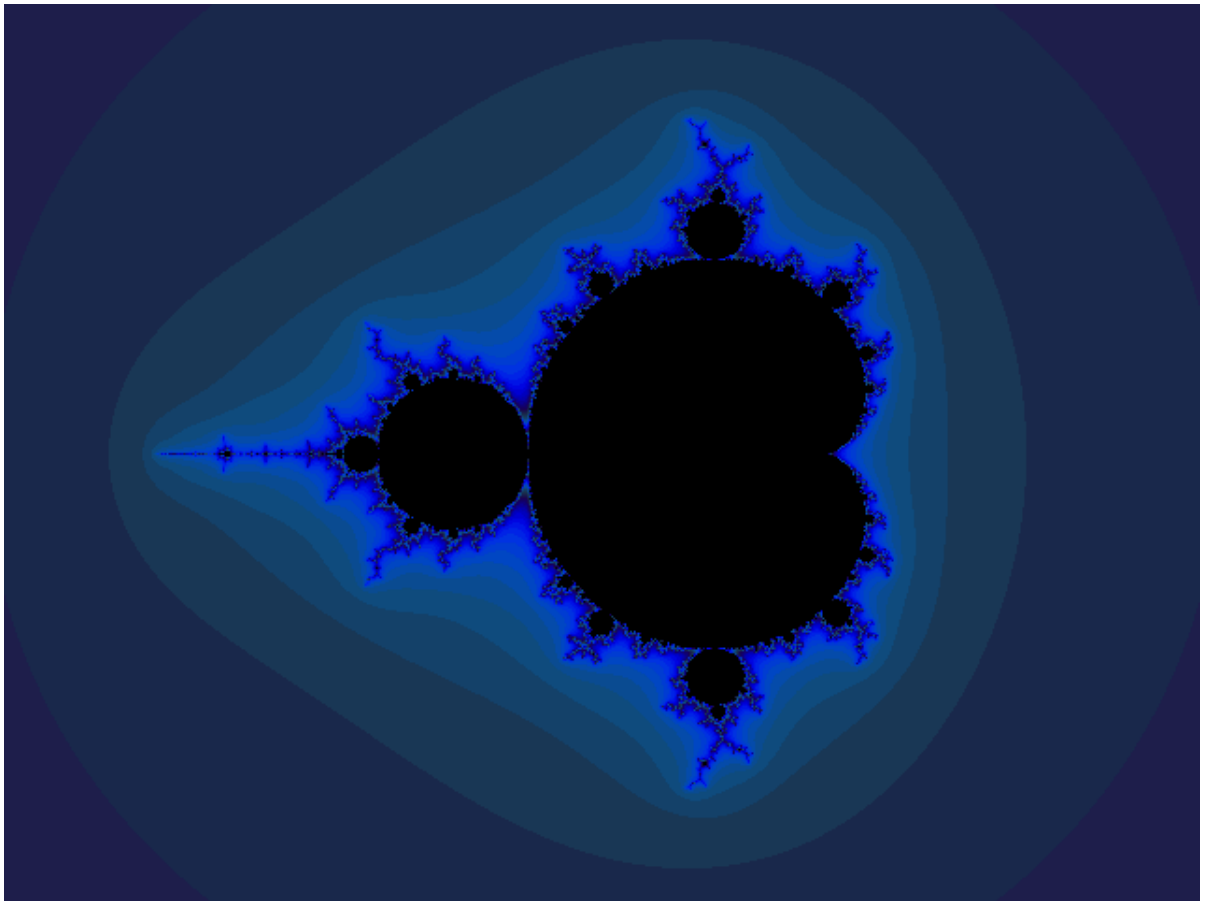
In [34]: counts = mandelbrot(
    #render_size=(3840,2160), # 4K
    #render_size=(1920,1080), # HD
    render_size=(640,480),
    center=(-0.5,0),
    zoom=4,
    cycles=200
)
img = render(counts)
print(img.size)
img

```

(640, 480)

Out[34]:



Mandelbrot rendering programs are both simple and infinitely complex at the same time. This view shows the entire Mandelbrot universe simultaneously, as a view completely zoomed out. However, if you zoom in on any non-black portion of the plot, you will find infinite hidden complexity.

Introduction to Keras

[Keras](#) is a layer on top of Tensorflow that makes it much easier to create neural networks. Rather than define the graphs, as you see above, you set the individual layers of the network with a much more high-level API. Unless you are researching entirely new structures of deep neural networks, it is unlikely that you need to program TensorFlow directly.

For this class, we will usually use TensorFlow through Keras, rather than direct TensorFlow

Simple TensorFlow Regression: MPG

This example shows how to encode the MPG dataset for regression and predict values. We will see if we can predict the miles per gallon (MPG) for a car based on the car's weight, cylinders, engine size, and other features.

```
In [35]: 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

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)
```

Train on 398 samples
Epoch 1/100
398/398 - 0s - loss: 359076.9421
Epoch 2/100
398/398 - 0s - loss: 74176.6153
Epoch 3/100
398/398 - 0s - loss: 9767.8802
Epoch 4/100
398/398 - 0s - loss: 1427.8838
Epoch 5/100
398/398 - 0s - loss: 2057.3394
Epoch 6/100
398/398 - 0s - loss: 1513.2297
Epoch 7/100
398/398 - 0s - loss: 1295.3801
Epoch 8/100
398/398 - 0s - loss: 1272.4230
Epoch 9/100
398/398 - 0s - loss: 1239.1273
Epoch 10/100
398/398 - 0s - loss: 1207.5261
Epoch 11/100
398/398 - 0s - loss: 1183.4229
Epoch 12/100
398/398 - 0s - loss: 1154.0081
Epoch 13/100
398/398 - 0s - loss: 1124.9607
Epoch 14/100
398/398 - 0s - loss: 1099.9529
Epoch 15/100
398/398 - 0s - loss: 1076.2990
Epoch 16/100
398/398 - 0s - loss: 1060.9813
Epoch 17/100
398/398 - 0s - loss: 1047.1127
Epoch 18/100
398/398 - 0s - loss: 1035.2276
Epoch 19/100
398/398 - 0s - loss: 1023.7794
Epoch 20/100
398/398 - 0s - loss: 1012.4624
Epoch 21/100
398/398 - 0s - loss: 1001.8168
Epoch 22/100
398/398 - 0s - loss: 992.2781
Epoch 23/100
398/398 - 0s - loss: 979.8584
Epoch 24/100
398/398 - 0s - loss: 969.5121
Epoch 25/100
398/398 - 0s - loss: 958.6515
Epoch 26/100
398/398 - 0s - loss: 950.3542
Epoch 27/100
398/398 - 0s - loss: 941.1478
Epoch 28/100

398/398 - 0s - loss: 926.8202
Epoch 29/100
398/398 - 0s - loss: 917.2789
Epoch 30/100
398/398 - 0s - loss: 905.2999
Epoch 31/100
398/398 - 0s - loss: 892.8540
Epoch 32/100
398/398 - 0s - loss: 884.2463
Epoch 33/100
398/398 - 0s - loss: 871.6899
Epoch 34/100
398/398 - 0s - loss: 864.9260
Epoch 35/100
398/398 - 0s - loss: 849.8424
Epoch 36/100
398/398 - 0s - loss: 840.7186
Epoch 37/100
398/398 - 0s - loss: 836.8501
Epoch 38/100
398/398 - 0s - loss: 824.8931
Epoch 39/100
398/398 - 0s - loss: 810.3899
Epoch 40/100
398/398 - 0s - loss: 800.1687
Epoch 41/100
398/398 - 0s - loss: 784.4148
Epoch 42/100
398/398 - 0s - loss: 774.2326
Epoch 43/100
398/398 - 0s - loss: 762.1020
Epoch 44/100
398/398 - 0s - loss: 751.2068
Epoch 45/100
398/398 - 0s - loss: 740.0900
Epoch 46/100
398/398 - 0s - loss: 728.2087
Epoch 47/100
398/398 - 0s - loss: 719.4002
Epoch 48/100
398/398 - 0s - loss: 710.2463
Epoch 49/100
398/398 - 0s - loss: 695.5551
Epoch 50/100
398/398 - 0s - loss: 686.5956
Epoch 51/100
398/398 - 0s - loss: 676.1623
Epoch 52/100
398/398 - 0s - loss: 661.9155
Epoch 53/100
398/398 - 0s - loss: 652.7001
Epoch 54/100
398/398 - 0s - loss: 649.1527
Epoch 55/100
398/398 - 0s - loss: 626.8102
Epoch 56/100

398/398 - 0s - loss: 617.8689
Epoch 57/100
398/398 - 0s - loss: 610.6475
Epoch 58/100
398/398 - 0s - loss: 592.5715
Epoch 59/100
398/398 - 0s - loss: 582.9929
Epoch 60/100
398/398 - 0s - loss: 571.0975
Epoch 61/100
398/398 - 0s - loss: 560.9386
Epoch 62/100
398/398 - 0s - loss: 549.7743
Epoch 63/100
398/398 - 0s - loss: 542.2976
Epoch 64/100
398/398 - 0s - loss: 524.7271
Epoch 65/100
398/398 - 0s - loss: 512.6835
Epoch 66/100
398/398 - 0s - loss: 501.3199
Epoch 67/100
398/398 - 0s - loss: 489.1912
Epoch 68/100
398/398 - 0s - loss: 476.9358
Epoch 69/100
398/398 - 0s - loss: 465.7011
Epoch 70/100
398/398 - 0s - loss: 454.4821
Epoch 71/100
398/398 - 0s - loss: 444.3988
Epoch 72/100
398/398 - 0s - loss: 432.2656
Epoch 73/100
398/398 - 0s - loss: 420.4270
Epoch 74/100
398/398 - 0s - loss: 408.0120
Epoch 75/100
398/398 - 0s - loss: 400.3801
Epoch 76/100
398/398 - 0s - loss: 384.8745
Epoch 77/100
398/398 - 0s - loss: 373.1397
Epoch 78/100
398/398 - 0s - loss: 364.7359
Epoch 79/100
398/398 - 0s - loss: 349.1083
Epoch 80/100
398/398 - 0s - loss: 336.7873
Epoch 81/100
398/398 - 0s - loss: 329.1152
Epoch 82/100
398/398 - 0s - loss: 315.6647
Epoch 83/100
398/398 - 0s - loss: 302.7605
Epoch 84/100

```
398/398 - 0s - loss: 292.5489
Epoch 85/100
398/398 - 0s - loss: 280.1445
Epoch 86/100
398/398 - 0s - loss: 268.6837
Epoch 87/100
398/398 - 0s - loss: 260.4212
Epoch 88/100
398/398 - 0s - loss: 250.5896
Epoch 89/100
398/398 - 0s - loss: 247.6127
Epoch 90/100
398/398 - 0s - loss: 230.2208
Epoch 91/100
398/398 - 0s - loss: 218.4122
Epoch 92/100
398/398 - 0s - loss: 208.7282
Epoch 93/100
398/398 - 0s - loss: 200.0094
Epoch 94/100
398/398 - 0s - loss: 189.9866
Epoch 95/100
398/398 - 0s - loss: 182.8754
Epoch 96/100
398/398 - 0s - loss: 175.6958
Epoch 97/100
398/398 - 0s - loss: 172.0255
Epoch 98/100
398/398 - 0s - loss: 156.4483
Epoch 99/100
398/398 - 0s - loss: 149.1817
Epoch 100/100
398/398 - 0s - loss: 142.9415
```

```
Out[35]: <tensorflow.python.keras.callbacks.History at 0x7f3ee44468d0>
```

Introduction to Neural Network Hyperparameters

If you look at the above code, you will see that the neural network contains four layers. The first layer is the input layer because it contains the **input_dim** parameter that the programmer sets to be the number of inputs the dataset has. The network needs one input neuron for every column in the data set (including dummy variables).

There are also several hidden layers, with 25 and 10 neurons each. You might be wondering how the programmer chose these numbers. Selecting a hidden neuron structure is one of the most common questions about neural networks. Unfortunately, there is no right answer. These are hyperparameters. They are settings that can affect neural network performance, yet there are no clearly defined means of setting them.

In general, more hidden neurons mean more capability to fit complex problems. However, too many neurons can lead to overfitting and lengthy training times. Too few can lead to underfitting the problem and will sacrifice accuracy. Also, how many layers you have is another hyperparameter. In general, more layers allow the neural network to perform more of its feature engineering and data preprocessing. But this also comes at the expense of training times and the risk of overfitting. In general, you will see that neuron counts start larger near the input layer and tend to shrink towards the output layer in a triangular fashion.

Some techniques use machine learning to optimize these values. These will be discussed in [Module 8.3](#).

Controlling the Amount of Output

The program produces one line of output for each training epoch. You can eliminate this output by setting the verbose setting of the fit command:

- **verbose=0** - No progress output (use with Jupyter if you do not want output).
- **verbose=1** - Display progress bar, does not work well with Jupyter.
- **verbose=2** - Summary progress output (use with Jupyter if you want to know the loss at each epoch).

Regression Prediction

Next, we will perform actual predictions. The program assigns these predictions to the **pred** variable. These are all MPG predictions from the neural network. Notice that this is a 2D array? You can always see the dimensions of what Keras returns by printing out **pred.shape**. Neural networks can return multiple values, so the result is always an array. Here the neural network only returns one value per prediction (there are 398 cars, so 398 predictions). However, a 2D range is needed because the neural network has the potential of returning more than one value.

```
In [36]: pred = model.predict(x)
print(f"Shape: {pred.shape}")
print(pred[0:10])
```

```
Shape: (398, 1)
[[22.639828]
 [20.882801]
 [19.801853]
 [20.337807]
 [21.1946  ]
 [23.72337 ]
 [21.285397]
 [21.545208]
 [21.873882]
 [19.303974]]
```

We would like to see how good these predictions are. We know the correct MPG for each car so we can measure how close the neural network was.

```
In [37]: # Measure RMSE error. RMSE is common for regression.
score = np.sqrt(metrics.mean_squared_error(pred,y))
print(f"Final score (RMSE): {score}")
```

Final score (RMSE): 11.862759295790802

The number printed above is the average number of predictions above or below the expected output. We can also print out the first ten cars with predictions and actual MPG.

```
In [38]: # Sample predictions
for i in range(10):
    print(f"{i+1}. Car name: {cars[i]}, MPG: {y[i]}, "
          + f"predicted MPG: {pred[i]}")
```

```
1. Car name: chevrolet chevelle malibu, MPG: 18.0, predicted MPG: [22.63982
8]
2. Car name: buick skylark 320, MPG: 15.0, predicted MPG: [20.882801]
3. Car name: plymouth satellite, MPG: 18.0, predicted MPG: [19.801853]
4. Car name: amc rebel sst, MPG: 16.0, predicted MPG: [20.337807]
5. Car name: ford torino, MPG: 17.0, predicted MPG: [21.1946]
6. Car name: ford galaxie 500, MPG: 15.0, predicted MPG: [23.72337]
7. Car name: chevrolet impala, MPG: 14.0, predicted MPG: [21.285397]
8. Car name: plymouth fury iii, MPG: 14.0, predicted MPG: [21.545208]
9. Car name: pontiac catalina, MPG: 14.0, predicted MPG: [21.873882]
10. Car name: amc ambassador dpl, MPG: 15.0, predicted MPG: [19.303974]
```

Simple TensorFlow Classification: Iris

Classification is how a neural network attempts to classify the input into one or more classes. The simplest way of evaluating a classification network is to track the percentage of training set items classified incorrectly. We typically score human results in this manner. For example, you might have taken multiple-choice exams in school in which you had to shade in a bubble for choices A, B, C, or D. If you chose the wrong letter on a 10-question exam, you would earn a 90%. In the same way, we can grade computers; however, most classification algorithms do not merely choose A, B, C, or D. Computers typically report a

classification as their percent confidence in each class. Figure 3.EXAM shows how a computer and a human might respond to question number 1 on an exam.

Figure 3.EXAM: Classification Neural Network Output

 Classification Neural Network Output

As you can see, the human test taker marked the first question as "B." However, the computer test taker had an 80% (0.8) confidence in "B" and was also somewhat sure with 10% (0.1) on "A." The computer then distributed the remaining points to the other two. In the simplest sense, the machine would get 80% of the score for this question if the correct answer were "B." The computer would get only 5% (0.05) of the points if the correct answer were "D."

We previously saw how to train a neural network to predict the MPG of a car. Based on four measurements, we will now see how to predict a class, such as the type of iris flower. The code to classify iris flowers is similar to MPG; however, there are several important differences:

- The output neuron count matches the number of classes (in the case of Iris, 3).
- The Softmax transfer function is utilized by the output layer.* The loss function is cross entropy.

```
In [39]: 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

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

# Build neural network
model = Sequential()
model.add(Dense(50, input_dim=x.shape[1], activation='relu')) # Hidden 1
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,y,verbose=2,epochs=100)
```

Train on 150 samples
Epoch 1/100
150/150 - 0s - loss: 1.4756
Epoch 2/100
150/150 - 0s - loss: 1.3098
Epoch 3/100
150/150 - 0s - loss: 1.1715
Epoch 4/100
150/150 - 0s - loss: 1.0646
Epoch 5/100
150/150 - 0s - loss: 0.9959
Epoch 6/100
150/150 - 0s - loss: 0.9439
Epoch 7/100
150/150 - 0s - loss: 0.8898
Epoch 8/100
150/150 - 0s - loss: 0.8446
Epoch 9/100
150/150 - 0s - loss: 0.8176
Epoch 10/100
150/150 - 0s - loss: 0.7961
Epoch 11/100
150/150 - 0s - loss: 0.7736
Epoch 12/100
150/150 - 0s - loss: 0.7527
Epoch 13/100
150/150 - 0s - loss: 0.7327
Epoch 14/100
150/150 - 0s - loss: 0.7109
Epoch 15/100
150/150 - 0s - loss: 0.6913
Epoch 16/100
150/150 - 0s - loss: 0.6720
Epoch 17/100
150/150 - 0s - loss: 0.6527
Epoch 18/100
150/150 - 0s - loss: 0.6311
Epoch 19/100
150/150 - 0s - loss: 0.6117
Epoch 20/100
150/150 - 0s - loss: 0.5933
Epoch 21/100
150/150 - 0s - loss: 0.5761
Epoch 22/100
150/150 - 0s - loss: 0.5568
Epoch 23/100
150/150 - 0s - loss: 0.5384
Epoch 24/100
150/150 - 0s - loss: 0.5222
Epoch 25/100
150/150 - 0s - loss: 0.5063
Epoch 26/100
150/150 - 0s - loss: 0.4945
Epoch 27/100
150/150 - 0s - loss: 0.4753
Epoch 28/100

150/150 - 0s - loss: 0.4608
Epoch 29/100
150/150 - 0s - loss: 0.4478
Epoch 30/100
150/150 - 0s - loss: 0.4333
Epoch 31/100
150/150 - 0s - loss: 0.4209
Epoch 32/100
150/150 - 0s - loss: 0.4114
Epoch 33/100
150/150 - 0s - loss: 0.3964
Epoch 34/100
150/150 - 0s - loss: 0.3886
Epoch 35/100
150/150 - 0s - loss: 0.3799
Epoch 36/100
150/150 - 0s - loss: 0.3674
Epoch 37/100
150/150 - 0s - loss: 0.3569
Epoch 38/100
150/150 - 0s - loss: 0.3477
Epoch 39/100
150/150 - 0s - loss: 0.3386
Epoch 40/100
150/150 - 0s - loss: 0.3297
Epoch 41/100
150/150 - 0s - loss: 0.3243
Epoch 42/100
150/150 - 0s - loss: 0.3121
Epoch 43/100
150/150 - 0s - loss: 0.3051
Epoch 44/100
150/150 - 0s - loss: 0.2995
Epoch 45/100
150/150 - 0s - loss: 0.2886
Epoch 46/100
150/150 - 0s - loss: 0.2836
Epoch 47/100
150/150 - 0s - loss: 0.2746
Epoch 48/100
150/150 - 0s - loss: 0.2683
Epoch 49/100
150/150 - 0s - loss: 0.2608
Epoch 50/100
150/150 - 0s - loss: 0.2545
Epoch 51/100
150/150 - 0s - loss: 0.2483
Epoch 52/100
150/150 - 0s - loss: 0.2426
Epoch 53/100
150/150 - 0s - loss: 0.2349
Epoch 54/100
150/150 - 0s - loss: 0.2310
Epoch 55/100
150/150 - 0s - loss: 0.2238
Epoch 56/100

150/150 - 0s - loss: 0.2193
Epoch 57/100
150/150 - 0s - loss: 0.2144
Epoch 58/100
150/150 - 0s - loss: 0.2101
Epoch 59/100
150/150 - 0s - loss: 0.2043
Epoch 60/100
150/150 - 0s - loss: 0.1996
Epoch 61/100
150/150 - 0s - loss: 0.1954
Epoch 62/100
150/150 - 0s - loss: 0.1909
Epoch 63/100
150/150 - 0s - loss: 0.1858
Epoch 64/100
150/150 - 0s - loss: 0.1823
Epoch 65/100
150/150 - 0s - loss: 0.1789
Epoch 66/100
150/150 - 0s - loss: 0.1747
Epoch 67/100
150/150 - 0s - loss: 0.1722
Epoch 68/100
150/150 - 0s - loss: 0.1684
Epoch 69/100
150/150 - 0s - loss: 0.1633
Epoch 70/100
150/150 - 0s - loss: 0.1623
Epoch 71/100
150/150 - 0s - loss: 0.1579
Epoch 72/100
150/150 - 0s - loss: 0.1563
Epoch 73/100
150/150 - 0s - loss: 0.1551
Epoch 74/100
150/150 - 0s - loss: 0.1513
Epoch 75/100
150/150 - 0s - loss: 0.1484
Epoch 76/100
150/150 - 0s - loss: 0.1435
Epoch 77/100
150/150 - 0s - loss: 0.1425
Epoch 78/100
150/150 - 0s - loss: 0.1396
Epoch 79/100
150/150 - 0s - loss: 0.1378
Epoch 80/100
150/150 - 0s - loss: 0.1351
Epoch 81/100
150/150 - 0s - loss: 0.1357
Epoch 82/100
150/150 - 0s - loss: 0.1308
Epoch 83/100
150/150 - 0s - loss: 0.1284
Epoch 84/100

```
150/150 - 0s - loss: 0.1278
Epoch 85/100
150/150 - 0s - loss: 0.1248
Epoch 86/100
150/150 - 0s - loss: 0.1237
Epoch 87/100
150/150 - 0s - loss: 0.1212
Epoch 88/100
150/150 - 0s - loss: 0.1214
Epoch 89/100
150/150 - 0s - loss: 0.1183
Epoch 90/100
150/150 - 0s - loss: 0.1199
Epoch 91/100
150/150 - 0s - loss: 0.1155
Epoch 92/100
150/150 - 0s - loss: 0.1159
Epoch 93/100
150/150 - 0s - loss: 0.1112
Epoch 94/100
150/150 - 0s - loss: 0.1141
Epoch 95/100
150/150 - 0s - loss: 0.1104
Epoch 96/100
150/150 - 0s - loss: 0.1089
Epoch 97/100
150/150 - 0s - loss: 0.1089
Epoch 98/100
150/150 - 0s - loss: 0.1089
Epoch 99/100
150/150 - 0s - loss: 0.1197
Epoch 100/100
150/150 - 0s - loss: 0.1076
```

```
Out[39]: <tensorflow.python.keras.callbacks.History at 0x7f3ee4332410>
```

```
In [40]: # Print out number of species found:
```

```
print(species)
```

```
Index(['Iris-setosa', 'Iris-versicolor', 'Iris-virginica'], dtype='object')
```

Now that you have a neural network trained, we would like to be able to use it. The following code makes use of our neural network. Exactly like before, we will generate predictions. Notice that three values come back for each of the 150 iris flowers. There were three types of iris (Iris-setosa, Iris-versicolor, and Iris-virginica).

```
In [41]: pred = model.predict(x)
print(f"Shape: {pred.shape}")
print(pred[0:10])
```

```
Shape: (150, 3)
[[0.99278367 0.00704507 0.00017127]
 [0.98646706 0.01317458 0.0003583 ]
 [0.98927665 0.01040124 0.00032212]
 [0.98444796 0.01508906 0.00046296]
 [0.99318063 0.00664989 0.0001695 ]
 [0.9944395  0.00544237 0.00011823]
 [0.99035525 0.00932539 0.0003193 ]
 [0.99134773 0.00843632 0.00021587]
 [0.980791   0.01855918 0.00064985]
 [0.98711026 0.01257468 0.00031499]]
```

If you would like to turn off scientific notation, the following line can be used:

```
In [42]: np.set_printoptions(suppress=True)
```

Now we see these values rounded up.

```
In [43]: print(y[0:10])
```

[illegible]

Usually, the program considers the column with the highest prediction to be the prediction of the neural network. It is easy to convert the predictions to the expected iris species. The `argmax` function finds the index of the maximum prediction for each row.

```
In [44]: predict_classes = np.argmax(pred,axis=1)
expected_classes = np.argmax(y,axis=1)
print(f"Predictions: {predict_classes}")
print(f"Expected: {expected_classes}")
```

[illegible]

Of course, it is straightforward to turn these indexes back into iris species. We use the species list that we created earlier.

```
In [45]: print(species[predict_classes[1:10]])
Index(['Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
      'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
      'Iris-setosa'],
      dtype='object')
```

Accuracy might be a more easily understood error metric. It is essentially a test score. For all of the iris predictions, what percent were correct? The downside is it does not consider how confident the neural network was in each prediction.

```
In [46]: from sklearn.metrics import accuracy_score

correct = accuracy_score(expected_classes, predict_classes)
print(f"Accuracy: {correct}")
```

Accuracy: 0.9733333333333334

The code below performs two ad hoc predictions. The first prediction is a single iris flower, and the second predicts two iris flowers. Notice that the **argmax** in the second prediction requires **axis=1**? Since we have a 2D array now, we must specify which axis to take the **argmax** over. The value **axis=1** specifies we want the max column index for each row.

```
In [47]: sample_flower = np.array( [[5.0,3.0,4.0,2.0]], dtype=float)
pred = model.predict(sample_flower)
print(pred)
pred = np.argmax(pred)
print(f"Predict that {sample_flower} is: {species[pred]}")
```

```
[[0.00402835 0.25205988 0.74391174]]
Predict that [[5. 3. 4. 2.]] is: Iris-virginica
```

You can also predict two sample flowers.

```
In [48]: sample_flower = np.array( [[5.0,3.0,4.0,2.0],[5.2,3.5,1.5,0.8]],\
      dtype=float)
pred = model.predict(sample_flower)
print(pred)
pred = np.argmax(pred,axis=1)
print(f"Predict that these two flowers {sample_flower} ")
print(f"are: {species[pred]}")
```

```
[[0.00402835 0.25205988 0.74391174]
 [0.98642194 0.01315794 0.0004201 ]]
Predict that these two flowers [[5. 3. 4. 2. ]
 [5.2 3.5 1.5 0.8]]
are: Index(['Iris-virginica', 'Iris-setosa'], dtype='object')
```