In [2]:
```
!pip3 install tensorflow keras
```

```
Collecting tensorflow
  Downloading tensorflow-2.7.4-cp39-cp39-macosx_10_11_x86_64.whl (21
3.2 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 213.2/213.2 MB 6.4 MB/s
eta 0:00:0000:0100:01
Requirement already satisfied: keras in /opt/anaconda3/lib/python3.9/
site-packages (2.11.0)
Collecting flatbuffers<3.0,>=1.12
  Downloading flatbuffers-2.0.7-py2.py3-none-any.whl (26 kB)
Collecting libclang>=9.0.1
  Downloading libclang-15.0.6.1-py2.py3-none-macosx_10_9_x86_64.whl
(25.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 25.0/25.0 MB 9.6 MB/s e
ta 0:00:00:00:0100:01
Collecting keras
  Downloading keras-2.7.0-py2.py3-none-any.whl (1.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 13.6 MB/s et
a 0:00:0000:0100:01
Requirement already satisfied: h5py>=2.9.0 in /opt/anaconda3/lib/pyth
```

Neural Networks - intro

Part 1 - XOR

1. Using the XOR dataset below, train (400 epochs) a neural network (NN) using 1, 2, 3, 4, and 5 hidden layers (where each layer has only 2 neurons). For each n layers, store the resulting loss score along with n. Plot the results to find what the optimal number of layers is.
2. Repeat the above with 3 neurons in each Hidden layers. How do these results compare to the 2 neuron layers?
3. Repeat the above with 4 neurons in each Hidden layers. How do these results compare to the 2 and 3 neuron layers?
4. Using the most optimal configuraion (n-layers, k-neurons per layer), compare how `tanh`, `sigmoid`, `softplus` and `relu` effect the loss after 400 epochs. Try other Activation functions as well ([https://keras.io/activations/ (https://keras.io/activations/)](https://keras.io/activations/))
5. Again with the most optimal setup, try other optimizers (instead of `SGD`) and report on the loss score. ([https://keras.io/optimizers/ (https://keras.io/optimizers/)](https://keras.io/optimizers/))

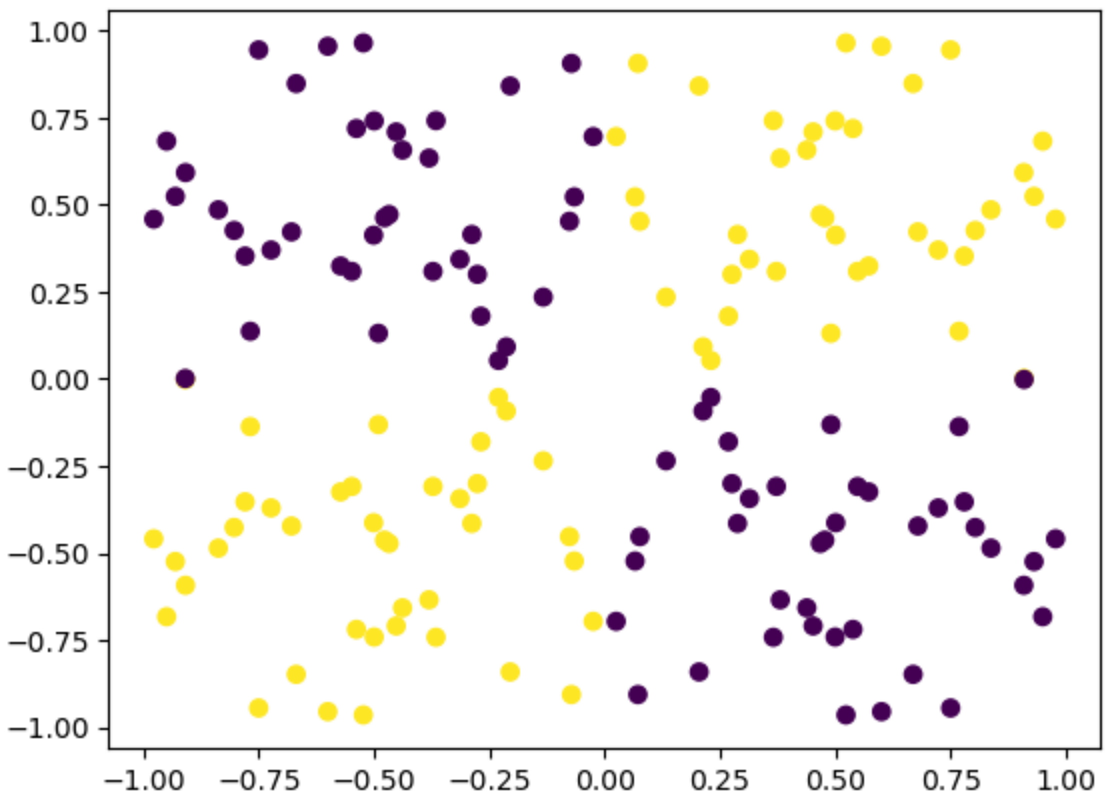
In [1]:
```
from keras.models import Sequential
from keras.layers import Dense
from tensorflow.keras.optimizers import SGD  #Stochastic Gradient Desc

import numpy as np
# fix random seed for reproducibility
np.random.seed(7)

import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
n = 40
xx = np.random.random((n,1))
yy = np.random.random((n,1))
```

In [3]:
```
X = np.array([np.array([xx,-xx,-xx,xx]),np.array([yy,-yy,yy,-yy])]).re
y = np.array([np.ones([2*n]),np.zeros([2*n])]).reshape(4*n)
```

In [4]:
```
plt.scatter(*zip(*X), c=y)
```

Out[4]: <matplotlib.collections.PathCollection at 0x161962880>



In [5]:
```
num_layers = [1,2,3,4,5]
scores_2 = []
for num_layer in num_layers:

    # build model and evaluate
    model = Sequential()

    i = 0
    while i < num_layer:
        model.add(Dense(2, input_dim=2, activation='tanh'))
        print(num_layer)
        i = i + 1
        print(i)

    sgd = SGD(learning_rate=0.1)
    model.compile(loss='binary_crossentropy', optimizer='sgd')

    model.fit(X, y, batch_size=2, epochs=400) #160/4 = 40 per epoch
   # print(model.predict(X).reshape(4*n))

    score = model.evaluate(X, y)
    scores_2.append(score)
```

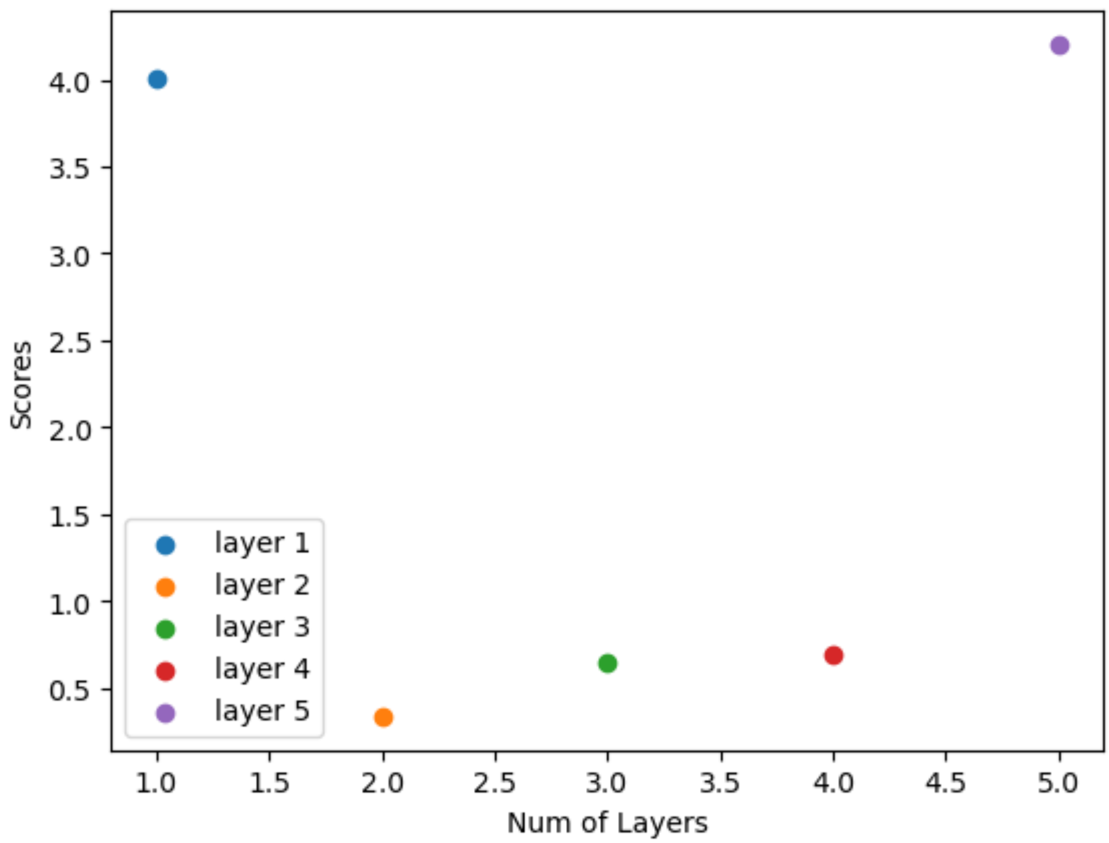
```
2023-04-07 22:56:21.666996: I tensorflow/core/platform/cpu_feature_gu
ard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neur
al Network Library (oneDNN) to use the following CPU instructions in
performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appro
priate compiler flags.

1
1

2023-04-07 22:56:22.248435: I tensorflow/compiler/mlir/mlir_graph_opt
imization_pass.cc:185] None of the MLIR Optimization Passes are enabl
ed (registered 2)

Epoch 1/400
80/80 [==============================] - 0s 622us/step - loss: 4.1128
Epoch 2/400
80/80 [==============================] - 0s 589us/step - loss: 4.0889
Epoch 3/400
80/80 [==============================] - 0s 584us/step - loss: 4.0729
```

In [6]:
```
# plot scores
scores_2
```

Out[6]:
```
[4.006331443786621,
 0.3380340039730072,
 0.6530135869979858,
 0.6938508749008179,
 4.202830314636230 5]
```

In [7]:
```
plt.figure()
plt.scatter(num_layers[0], scores_2[0], label='layer 1')
plt.scatter(num_layers[1], scores_2[1], label='layer 2')
plt.scatter(num_layers[2], scores_2[2], label='layer 3')
plt.scatter(num_layers[3], scores_2[3], label='layer 4')
plt.scatter(num_layers[4], scores_2[4], label='layer 5')
plt.xlabel("Num of Layers")
plt.ylabel("Scores")
plt.legend()
```

Out[7]:
```
<matplotlib.legend.Legend at 0x16457c7c0>
```



In [8]:
```
# 3 neurons
num_layers = [1,2,3,4,5]
scores_3 = []
for num_layer in num_layers:

    # build model and evaluate
    model = Sequential()

    i = 0
    while i < num_layer:
        model.add(Dense(3, input_dim=2, activation='tanh'))
        print(num_layer)
        i = i + 1
        print(i)

    sgd = SGD(learning_rate=0.1)
    model.compile(loss='binary_crossentropy', optimizer='sgd')

    model.fit(X, y, batch_size=2, epochs=400) #160/4 = 40 per epoch
   # print(model.predict(X).reshape(4*n))

    score = model.evaluate(X, y)
    scores_3.append(score)
```

```
1
1
Epoch 1/400
80/80 [==============================] - 0s 625us/step - loss: 1.7182
Epoch 2/400
80/80 [==============================] - 0s 619us/step - loss: 0.8244
Epoch 3/400
80/80 [==============================] - 0s 602us/step - loss: 0.7796
Epoch 4/400
80/80 [==============================] - 0s 614us/step - loss: 0.7526
Epoch 5/400
80/80 [==============================] - 0s 586us/step - loss: 0.7350
Epoch 6/400
80/80 [==============================] - 0s 586us/step - loss: 0.7229
Epoch 7/400
80/80 [==============================] - 0s 577us/step - loss: 0.7141
Epoch 8/400
80/80 [==============================] - 0s 590us/step - loss: 0.7084
Epoch 9/400
```

In [9]:
```
scores_3
```

Out[9]:
```
[0.6931473016738892,
 2.6472299098968506,
 0.09375002235174179,
 0.03623267635703087,
 0.6944209337234497]
```

In [10]:
```
# 4 neurons
num_layers = [1,2,3,4,5]
scores_4 = []
for num_layer in num_layers:

    # build model and evaluate
    model = Sequential()

    i = 0
    while i < num_layer:
        model.add(Dense(4, input_dim=2, activation='tanh'))
        print(num_layer)
        i = i + 1
        print(i)

    sgd = SGD(learning_rate=0.1)
    model.compile(loss='binary_crossentropy', optimizer='sgd')

    model.fit(X, y, batch_size=2, epochs=400) #160/4 = 40 per epoch
   # print(model.predict(X).reshape(4*n))

    score = model.evaluate(X, y)
    scores_4.append(score)
```

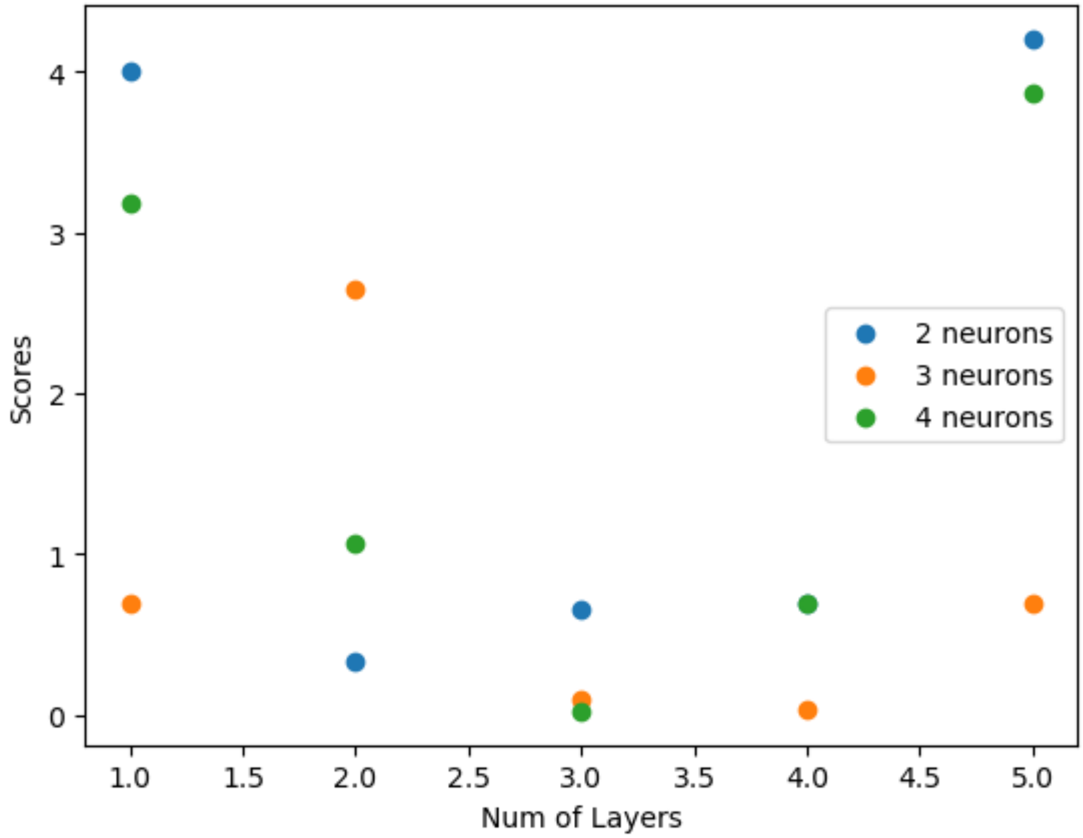
```
1
1
Epoch 1/400
80/80 [==============================] - 0s 608us/step - loss: 3.9267
Epoch 2/400
80/80 [==============================] - 0s 621us/step - loss: 3.3481
Epoch 3/400
80/80 [==============================] - 0s 591us/step - loss: 3.2724
Epoch 4/400
80/80 [==============================] - 0s 609us/step - loss: 3.2543
Epoch 5/400
80/80 [==============================] - 0s 586us/step - loss: 3.2464
Epoch 6/400
80/80 [==============================] - 0s 599us/step - loss: 3.2399
Epoch 7/400
80/80 [==============================] - 0s 606us/step - loss: 3.2344
Epoch 8/400
80/80 [==============================] - 0s 609us/step - loss: 3.2299
Epoch 9/400
80/80 [==============================] - 0s 611us/step - loss: 3.2265
```

In [11]:
```
scores_4
```

Out[11]:
```
[3.1783578395843506,
 1.0658525228500366,
 0.024626335129141808,
 0.6932481527328491,
 3.867734909057617]
```

In [12]:
```
plt.figure()
plt.scatter(num_layers, scores_2, label='2 neurons')
plt.scatter(num_layers, scores_3, label='3 neurons')
plt.scatter(num_layers, scores_4, label='4 neurons')
plt.xlabel("Num of Layers")
plt.ylabel("Scores")
plt.legend()
```

Out[12]:
```
<matplotlib.legend.Legend at 0x1641bd2e0>
```



In [13]:
```
model = Sequential()

model.add(Dense(4, input_dim=2, activation='tanh'))
model.add(Dense(4, activation='tanh'))
model.add(Dense(4, activation='tanh'))
model.add(Dense(1, activation='tanh'))

sgd = SGD(learning_rate=0.1)
model.compile(loss='binary_crossentropy', optimizer='sgd')

model.fit(X, y, batch_size=2, epochs=400) #160/4 = 40 per epoch
#print(model.predict(X).reshape(4*n))

# evaluate the model
scores = model.evaluate(X, y)
```

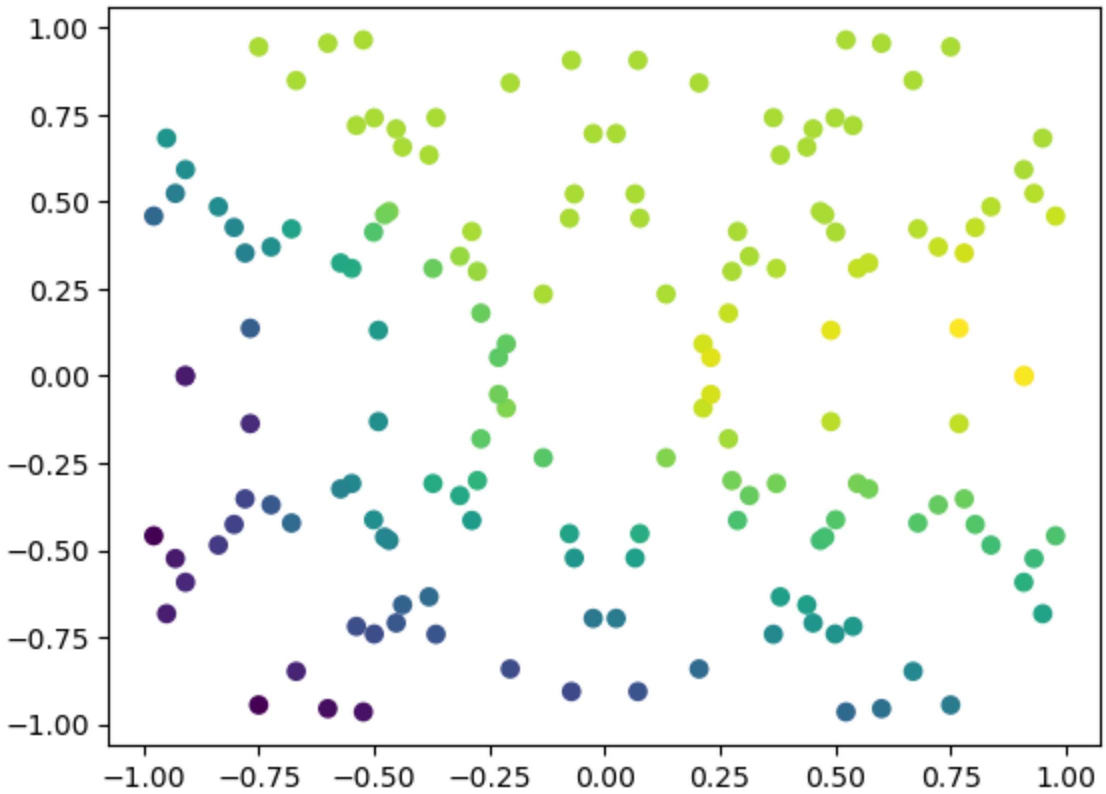
```
Epoch 1/400
80/80 [==============================] - 0s 711us/step - loss: 1.7001
Epoch 2/400
80/80 [==============================] - 0s 663us/step - loss: 0.7020
Epoch 3/400
80/80 [==============================] - 0s 659us/step - loss: 0.6921
Epoch 4/400
80/80 [==============================] - 0s 675us/step - loss: 0.6863
Epoch 5/400
80/80 [==============================] - 0s 662us/step - loss: 0.6676
Epoch 6/400
80/80 [==============================] - 0s 686us/step - loss: 0.6674
Epoch 7/400
80/80 [==============================] - 0s 680us/step - loss: 0.6454
Epoch 8/400
80/80 [==============================] - 0s 629us/step - loss: 0.6346
Epoch 9/400
80/80 [==============================] - 0s 683us/step - loss: 0.6221
Epoch 10/400
```

In [14]:
```
# relu comparison
model = Sequential()

model.add(Dense(4, input_dim=2, activation='relu'))
model.add(Dense(4, activation='relu'))
model.add(Dense(4, activation='relu'))
model.add(Dense(1, activation='relu'))

sgd = SGD(learning_rate=0.1)
model.compile(loss='binary_crossentropy', optimizer='sgd')

model.fit(X, y, batch_size=2, epochs=400) #160/4 = 40 per epoch
#print(model.predict(X).reshape(4*n))

# evaluate the model
scores = model.evaluate(X, y)
```

```
Epoch 1/400
80/80 [==============================] - 0s 704us/step - loss: 8.5464
Epoch 2/400
80/80 [==============================] - 0s 684us/step - loss: 7.6246
Epoch 3/400
80/80 [==============================] - 0s 656us/step - loss: 7.6246
Epoch 4/400
80/80 [==============================] - 0s 646us/step - loss: 7.6246
Epoch 5/400
80/80 [==============================] - 0s 753us/step - loss: 7.6246
Epoch 6/400
80/80 [==============================] - 0s 645us/step - loss: 7.6246
Epoch 7/400
80/80 [==============================] - 0s 676us/step - loss: 7.6246
Epoch 8/400
80/80 [==============================] - 0s 715us/step - loss: 7.6246
Epoch 9/400
80/80 [==============================] - 0s 681us/step - loss: 7.6246
Epoch 10/400
```

In [15]:
```
plt.scatter(*zip(*X), c=model.predict(X))
```

Out[15]: <matplotlib.collections.PathCollection at 0x164cc2670>



In [16]:
```
# sigmoid comparison
model = Sequential()

model.add(Dense(4, input_dim=2, activation='sigmoid'))
model.add(Dense(4, activation='sigmoid'))
model.add(Dense(4, activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))

sgd = SGD(learning_rate=0.1)
model.compile(loss='binary_crossentropy', optimizer='sgd')

model.fit(X, y, batch_size=2, epochs=400) #160/4 = 40 per epoch
#print(model.predict(X).reshape(4*n))

# evaluate the model
scores = model.evaluate(X, y)
```

```
Epoch 1/400
80/80 [==============================] - 0s 753us/step - loss: 0.7007
Epoch 2/400
80/80 [==============================] - 0s 675us/step - loss: 0.6967
Epoch 3/400
80/80 [==============================] - 0s 680us/step - loss: 0.6952
Epoch 4/400
80/80 [==============================] - 0s 656us/step - loss: 0.6948
Epoch 5/400
80/80 [==============================] - 0s 671us/step - loss: 0.6944
Epoch 6/400
80/80 [==============================] - 0s 674us/step - loss: 0.6944
Epoch 7/400
80/80 [==============================] - 0s 683us/step - loss: 0.6945
Epoch 8/400
80/80 [==============================] - 0s 687us/step - loss: 0.6946
Epoch 9/400
80/80 [==============================] - 0s 671us/step - loss: 0.6942
Epoch 10/400
80/80 [==============================] - 0s 685us/step - loss: 0.6941
```

In [17]:
```
print(model.predict(X).reshape(4*n))
```

```
[0.4977923  0.49882632 0.4976428  0.49872363 0.49876535 0.4975962
 0.49837252 0.49656928 0.49875602 0.49749246 0.4985417  0.49866426
 0.4976378  0.49758667 0.498132   0.49979523 0.49894056 0.49751994
 0.4985548  0.4970694  0.49705958 0.49818152 0.49906424 0.4987017
 0.49835983 0.49846172 0.49694005 0.49727488 0.49741837 0.4981755
 0.49688312 0.4991132  0.49851522 0.4982115  0.49733198 0.4985501
 0.49935746 0.4983568  0.49868575 0.49843433 0.50010467 0.49911216
 0.50026727 0.49922118 0.49916923 0.5003164  0.4995771  0.5011535
 0.49919456 0.50040823 0.49940854 0.4992808  0.50026786 0.50028825
 0.49980322 0.49802545 0.49900895 0.50038    0.4993918  0.5007326
 0.5008004  0.4997719  0.49888235 0.4992472  0.4995921  0.49948198
 0.5008967  0.5006217  0.50048685 0.49976832 0.5009115  0.49882215
 0.4994347  0.49973384 0.5005412  0.49939814 0.4985396  0.49958864
 0.49926314 0.49951237 0.49762622 0.49720162 0.49670845 0.49720642
 0.49676588 0.49646235 0.49730036 0.4964198  0.4981856  0.49644017
 0.49710843 0.4969887  0.496822   0.4974433  0.4975084  0.4980178
 0.49849275 0.4965621  0.49663943 0.4970159  0.49584714 0.49624458
 0.49858484 0.4975385  0.4964883  0.4981757  0.4958812  0.49577114
 0.49604654 0.49717182 0.49645352 0.49809024 0.4977166  0.49718773
```

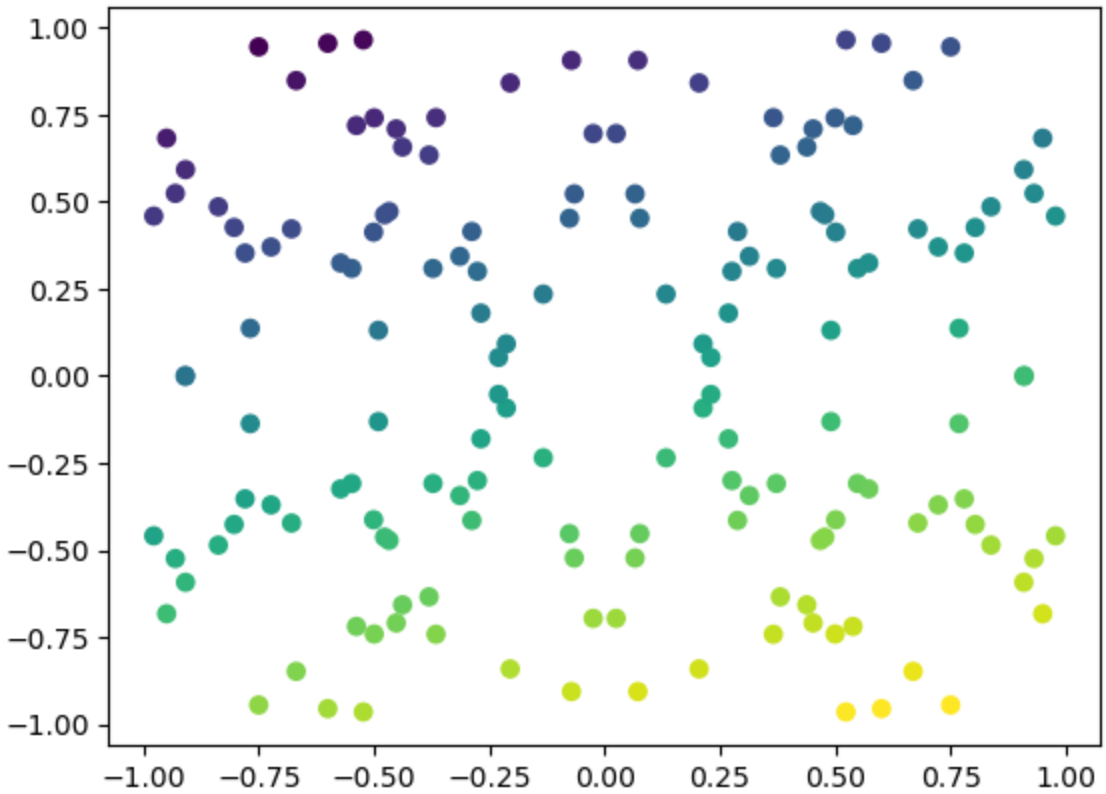
In [18]:
```
scores = model.evaluate(X, y)
scores, model.metrics_names
```

```
5/5 [==============================] - 0s 857us/step - loss: 0.6931
```

Out[18]:
```
(0.6930533647537231, ['loss'])
```

In [19]:
```
plt.scatter(*zip(*X), c=model.predict(X))
```

Out[19]: <matplotlib.collections.PathCollection at 0x164e965b0>



In [20]:
```
# adam comparison
model = Sequential()

model.add(Dense(4, input_dim=2, activation='tanh'))
model.add(Dense(4, activation='relu'))
model.add(Dense(4, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam')

model.fit(X, y, batch_size=2, epochs=400) #160/4 = 40 per epoch

# evaluate the model
scores = model.evaluate(X, y)
```

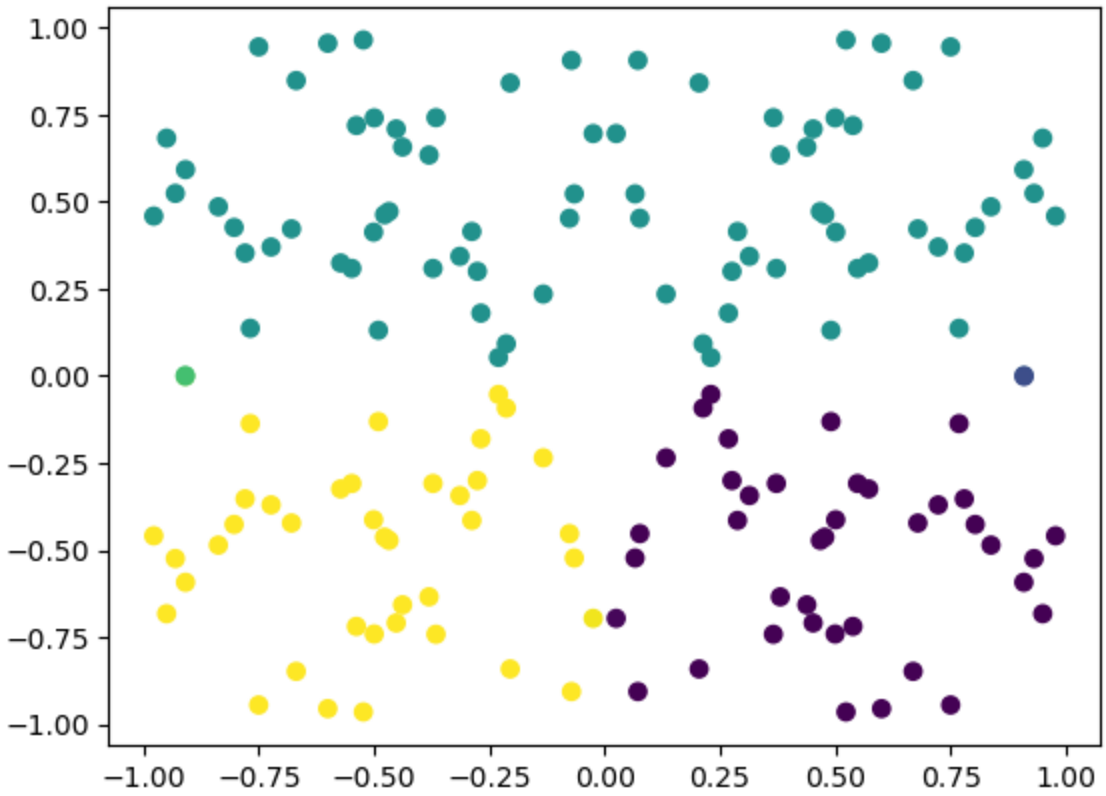
```
Epoch 1/400
80/80 [==============================] - 0s 824us/step - loss: 0.7003
Epoch 2/400
80/80 [==============================] - 0s 749us/step - loss: 0.6881
Epoch 3/400
80/80 [==============================] - 0s 752us/step - loss: 0.6757
Epoch 4/400
80/80 [==============================] - 0s 722us/step - loss: 0.6623
Epoch 5/400
80/80 [==============================] - 0s 766us/step - loss: 0.6461
Epoch 6/400
80/80 [==============================] - 0s 746us/step - loss: 0.6280
Epoch 7/400
80/80 [==============================] - 0s 736us/step - loss: 0.6093
Epoch 8/400
80/80 [==============================] - 0s 739us/step - loss: 0.5891
Epoch 9/400
80/80 [==============================] - 0s 738us/step - loss: 0.5675
Epoch 10/400
```

In [21]:
```
# rmsprop comparison
model = Sequential()

model.add(Dense(4, input_dim=2, activation='tanh'))
model.add(Dense(4, activation='relu'))
model.add(Dense(4, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='RMSprop')

model.fit(X, y, batch_size=2, epochs=400) #160/4 = 40 per epoch

# evaluate the model
scores = model.evaluate(X, y)
```

```
Epoch 1/400
80/80 [==============================] - 1s 806us/step - loss: 0.6682
Epoch 2/400
80/80 [==============================] - 0s 755us/step - loss: 0.6439
Epoch 3/400
80/80 [==============================] - 0s 732us/step - loss: 0.6188
Epoch 4/400
80/80 [==============================] - 0s 729us/step - loss: 0.5967
Epoch 5/400
80/80 [==============================] - 0s 749us/step - loss: 0.5775
Epoch 6/400
80/80 [==============================] - 0s 746us/step - loss: 0.5601
Epoch 7/400
80/80 [==============================] - 0s 714us/step - loss: 0.5432
Epoch 8/400
80/80 [==============================] - 0s 745us/step - loss: 0.5258
Epoch 9/400
80/80 [==============================] - 0s 720us/step - loss: 0.5110
Epoch 10/400
```

In [22]:
```
plt.scatter(*zip(*X), c=model.predict(X))
```

Out[22]: <matplotlib.collections.PathCollection at 0x164bae6d0>



Using Diabetes data

[http://archive.ics.uci.edu/ml/machine-learning-databases/pima-indians-diabetes/pima-indians-diabetes.data (http://archive.ics.uci.edu/ml/machine-learning-databases/pima-indians-diabetes/pima-indians-diabetes.data)](http://archive.ics.uci.edu/ml/machine-learning-databases/pima-indians-diabetes/pima-indians-diabetes.data)

1. Number of times pregnant
2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test
3. Diastolic blood pressure (mm Hg)
4. Triceps skin fold thickness (mm)
5. 2-Hour serum insulin (mu U/ml)
6. Body mass index (weight in kg/(height in m)^2)
7. Diabetes pedigree function
8. Age (years)
9. Class variable (0 or 1)

In [23]:
```
# load pima indians dataset
dataset = np.loadtxt("pima-indians-diabetes.data", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
```

In [24]:
```
dataset
```

Out[24]:
```
array([[  6.   , 148.   ,  72.   , ...,   0.627,  50.   ,   1.   ],
       [  1.   ,  85.   ,  66.   , ...,   0.351,  31.   ,   0.   ],
       [  8.   , 183.   ,  64.   , ...,   0.672,  32.   ,   1.   ],
       ...,
       [  5.   , 121.   ,  72.   , ...,   0.245,  30.   ,   0.   ],
       [  1.   , 126.   ,  60.   , ...,   0.349,  47.   ,   1.   ],
       [  1.   ,  93.   ,  70.   , ...,   0.315,  23.   ,   0.   ]])
```

In [25]:
```
X, Y
```

Out[25]:
```
(array([[  6.   , 148.   ,  72.   , ...,  33.6  ,   0.627,  50.   ],
        [  1.   ,  85.   ,  66.   , ...,  26.6  ,   0.351,  31.   ],
        [  8.   , 183.   ,  64.   , ...,  23.3  ,   0.672,  32.   ],
        ...,
        [  5.   , 121.   ,  72.   , ...,  26.2  ,   0.245,  30.   ],
        [  1.   , 126.   ,  60.   , ...,  30.1  ,   0.349,  47.   ],
        [  1.   ,  93.   ,  70.   , ...,  30.4  ,   0.315,  23.   
]]),
 array([1., 0., 1., 0., 1., 0., 1., 0., 1., 1., 0., 1., 0., 1., 1.,
1., 1.,
        1., 0., 1., 0., 0., 1., 1., 1., 1., 1., 0., 0., 0., 0., 1.,
0., 0.,
        0., 0., 0., 1., 1., 1., 0., 0., 0., 1., 0., 1., 0., 0., 1.,
0., 0.,
        0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 1., 0.,
1., 0.,
        0., 0., 1., 0., 1., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
0., 1.,
        0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1.,
1., 0.,
        0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 1., 1., 1.,
0., 0.,
        0., 1., 0., 0., 0., 1., 1., 0., 0., 1., 1., 1., 1., 1., 0.,
0., 0.,
        0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
0., 1.,
        0., 1., 1., 0., 0., 0., 1., 0., 0., 0., 0., 1., 1., 0., 0.,
0., 0.,
        1., 1., 0., 0., 0., 1., 0., 1., 0., 1., 0., 0., 0., 0., 0.,
1., 1.,
        1., 1., 1., 0., 0., 1., 1., 0., 1., 0., 1., 1., 1., 0., 0.,
0., 0.,
        0., 0., 1., 1., 0., 1., 0., 0., 0., 1., 1., 1., 1., 0., 1.,
1., 1.,
        1., 0., 0., 0., 0., 0., 1., 0., 0., 1., 1., 0., 0., 0., 1.,
1., 1.,
        1., 0., 0., 0., 1., 1., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
0., 1.,
        1., 0., 0., 0., 1., 0., 1., 0., 0., 1., 0., 1., 0., 0., 1.,
1., 0.,
        0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 1., 1., 0., 0.,
1., 0.,
        0., 0., 1., 1., 1., 0., 0., 1., 0., 1., 0., 1., 1., 0., 1.,
0., 0.,
        1., 0., 1., 1., 0., 0., 1., 0., 1., 0., 0., 1., 0., 1., 0.,
1., 1.,
        1., 0., 0., 1., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 1.,
1., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.,
1., 1.,
        1., 0., 1., 1., 0., 0., 1., 0., 0., 1., 0., 0., 1., 1., 0.,
0., 0.,
        0., 1., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1.,
0., 0.,
```

```
        1., 0., 0., 1., 0., 0., 1., 0., 1., 1., 0., 1., 0., 1., 0.,
1., 0.,
        1., 1., 0., 0., 0., 0., 1., 1., 0., 1., 0., 1., 0., 0., 0.,
0., 1.,
        1., 0., 1., 0., 1., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
1., 0.,
        0., 1., 1., 1., 0., 0., 1., 0., 0., 1., 0., 0., 0., 1., 0.,
0., 1.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.,
0., 0.,
        1., 0., 0., 0., 1., 0., 0., 0., 1., 1., 0., 0., 0., 0., 0.,
0., 0.,
        1., 0., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 1., 0.,
0., 0.,
        1., 0., 0., 0., 0., 1., 1., 0., 0., 0., 0., 0., 0., 1., 0.,
0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1., 1., 1.,
1., 0.,
        0., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 1.,
        1., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.,
0., 1.,
        0., 1., 1., 0., 0., 0., 1., 0., 1., 0., 1., 0., 1., 0., 1.,
0., 0.,
        1., 0., 0., 1., 0., 0., 0., 0., 1., 1., 0., 1., 0., 0., 0.,
0., 1.,
        1., 0., 1., 0., 0., 0., 1., 1., 0., 0., 0., 0., 0., 0., 0.,
0., 0.,
        0., 1., 0., 0., 0., 0., 1., 0., 0., 1., 0., 0., 0., 1., 0.,
0., 0.,
        1., 1., 1., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 1., 0.,
1., 1.,
        1., 1., 0., 1., 1., 0., 0., 0., 0., 0., 0., 0., 1., 1., 0.,
1., 0.,
        0., 1., 0., 1., 0., 0., 0., 0., 0., 1., 0., 1., 0., 1., 0.,
1., 1.,
        0., 0., 0., 0., 1., 1., 0., 0., 0., 1., 0., 1., 1., 0., 0.,
1., 0.,
        0., 1., 1., 0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 0., 0.,
0., 1.,
        1., 1., 0., 0., 0., 0., 0., 0., 1., 1., 0., 0., 1., 0., 0.,
1., 0.,
        1., 1., 1., 0., 0., 1., 1., 1., 0., 1., 0., 1., 0., 1., 0.,
0., 0.,
        0., 1., 0.]))
```

In [26]:
```
# create model
model = Sequential()
model.add(Dense(16, input_dim=8, activation='tanh'))
model.add(Dense(16, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['
# Fit the model
model.fit(X, Y, epochs=1000, batch_size=10)
# evaluate the model
scores = model.evaluate(X, Y)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

```
Epoch 1/1000
77/77 [==============================] - 0s 842us/step - loss: 0.6761
- accuracy: 0.6432
Epoch 2/1000
77/77 [==============================] - 0s 838us/step - loss: 0.6295
- accuracy: 0.6549
Epoch 3/1000
77/77 [==============================] - 0s 830us/step - loss: 0.6166
- accuracy: 0.6628
Epoch 4/1000
77/77 [==============================] - 0s 775us/step - loss: 0.6054
- accuracy: 0.6667
Epoch 5/1000
77/77 [==============================] - 0s 786us/step - loss: 0.5973
- accuracy: 0.6706
Epoch 6/1000
77/77 [==============================] - 0s 808us/step - loss: 0.5998
- accuracy: 0.6693
Epoch 7/1000
```

In [27]:
```
scores = model.evaluate(X, Y)
scores, model.metrics_names
```

```
24/24 [==============================] - 0s 688us/step - loss: 0.3841
- accuracy: 0.8203
```

Out[27]:
```
([0.38405176997184753, 0.8203125], ['loss', 'accuracy'])
```

In [28]:
```
print(model.predict(X))
```

```
[[6.00913048e-01]
 [2.52083242e-02]
 [9.74543452e-01]
 [1.11680478e-01]
 [6.42308176e-01]
 [4.64568138e-02]
 [3.78234029e-01]
 [3.92463773e-01]
 [7.37599671e-01]
 [4.61595237e-01]
 [2.53132284e-01]
 [9.72659528e-01]
 [6.90394640e-03]
 [9.87788558e-01]
 [6.42308176e-01]
 [9.76546884e-01]
 [9.89922643e-01]
 [2.64266253e-01]
 [7.31278956e-02]
```

In [29]:
```
#plt.scatter(*zip(*X), c=model.predict(X))
#plt.scatter(*zip(*X))
list(zip(*X))
```

Out[29]:
```
[(6.0,
  1.0,
  8.0,
  1.0,
  0.0,
  5.0,
  3.0,
  10.0,
  2.0,
  8.0,
  4.0,
  10.0,
  10.0,
  1.0,
  5.0,
  7.0,
  0.0,
  7.0,
  1.0,
```

In [30]:
```
X
```

Out[30]:
```
array([[  6.   , 148.   ,  72.   , ...,  33.6  ,   0.627,  50.   ],
       [  1.   ,  85.   ,  66.   , ...,  26.6  ,   0.351,  31.   ],
       [  8.   , 183.   ,  64.   , ...,  23.3  ,   0.672,  32.   ],
       ...,
       [  5.   , 121.   ,  72.   , ...,  26.2  ,   0.245,  30.   ],
       [  1.   , 126.   ,  60.   , ...,  30.1  ,   0.349,  47.   ],
       [  1.   ,  93.   ,  70.   , ...,  30.4  ,   0.315,  23.   ]])
```

Part 2 - BYOD (Bring your own Dataset)

Using your own dataset, experiment and find the best Neural Network configuration. You may use any resource to improve results, just reference it.

While you may use any dataset, I'd prefer you didn't use the diabetes dataset used in the lesson.

https://stackoverflow.com/questions/34673164/how-to-train-and-tune-an-artificial-multilayer-perceptron-neural-network-using-k (https://stackoverflow.com/questions/34673164/how-to-train-and-tune-an-artificial-multilayer-perceptron-neural-network-using-k)

https://keras.io/ (https://keras.io/)

In [31]:
```
import pandas as pd

beans = pd.read_excel('Dry_Bean_Dataset/Dry_Bean_Dataset.xlsx')

beans.head()
```

Out[31]:

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	Eccentricity	ConvexAre
0	28395	610.291	208.178117	173.888747	1.197191	0.549812	2871
1	28734	638.018	200.524796	182.734419	1.097356	0.411785	2917
2	29380	624.110	212.826130	175.931143	1.209713	0.562727	2969
3	30008	645.884	210.557999	182.516516	1.153638	0.498616	3072
4	30140	620.134	201.847882	190.279279	1.060798	0.333680	3041

In [32]:
```
from sklearn import preprocessing

enc = preprocessing.OneHotEncoder(sparse = False)
```

In [33]:
```
enc.fit(beans[["Class"]])
enc.categories_
```

```
/opt/anaconda3/lib/python3.9/site-packages/sklearn/preprocessing/_enc
oders.py:828: FutureWarning: `sparse` was renamed to `sparse_output`
in version 1.2 and will be removed in 1.4. `sparse_output` is ignored
unless you leave `sparse` to its default value.
  warnings.warn(
```

Out[33]:
```
[array(['BARBUNYA', 'BOMBAY', 'CALI', 'DERMASON', 'HOROZ', 'SEKER', '
SIRA'],
       dtype=object)]
```

In [34]:
```
y = enc.fit_transform(beans[["Class"]])
y
```

```
/opt/anaconda3/lib/python3.9/site-packages/sklearn/preprocessing/_enc
oders.py:828: FutureWarning: `sparse` was renamed to `sparse_output`
in version 1.2 and will be removed in 1.4. `sparse_output` is ignored
unless you leave `sparse` to its default value.
  warnings.warn(
```

Out[34]:
```
array([[0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

In [35]:
```
beans.head()
```

Out[35]:

	Area	Perimeter	MajorAxisLength	MinorAxisLength	AspectRation	Eccentricity	ConvexAre
0	28395	610.291	208.178117	173.888747	1.197191	0.549812	2871
1	28734	638.018	200.524796	182.734419	1.097356	0.411785	2917
2	29380	624.110	212.826130	175.931143	1.209713	0.562727	2969
3	30008	645.884	210.557999	182.516516	1.153638	0.498616	3072
4	30140	620.134	201.847882	190.279279	1.060798	0.333680	3041

In [37]:
```
x = preprocessing.StandardScaler().fit_transform(beans.drop(['Class'],
x
```

Out[37]:
```
array([[-0.84074853, -1.1433189 , -1.30659814, ...,  2.40217287,
         1.92572347,  0.83837103],
       [-0.82918764, -1.01392388, -1.39591111, ...,  3.10089314,
         2.68970162,  0.77113842],
       [-0.80715717, -1.07882906, -1.25235661, ...,  2.23509147,
         1.84135576,  0.91675514],
       ...,
       [-0.37203825, -0.44783294, -0.45047814, ...,  0.28920441,
         0.33632829,  0.39025114],
       [-0.37176543, -0.42702856, -0.42897404, ...,  0.22837538,
         0.2489734 ,  0.03644001],
       [-0.37135619, -0.38755718, -0.2917356 , ..., -0.12777587,
        -0.2764814 ,  0.71371948]])
```

In [38]:
```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=.2
#beans.shape, train.shape, test.shape
x_train
```

Out[38]:
```
array([[-0.16046385,  0.18960797,  0.74191759, ..., -1.44380004,
        -2.27843226,  0.67999998],
       [-0.09447474,  0.12887953,  0.48799531, ..., -1.10046477,
        -1.56548678, -1.13181319],
       [-0.32033825, -0.43514404, -0.70665668, ...,  1.30852819,
         1.83106171,  0.94191871],
       ...,
       [-0.64755559, -0.9013245 , -1.16352296, ...,  2.45652854,
         2.5138521 ,  0.95198236],
       [-0.47492671, -0.67033898, -0.91607438, ...,  1.77349309,
         2.11611477,  0.77423579],
       [ 1.08879362,  1.50696189,  1.26371968, ..., -1.06635481,
        -0.54702293,  0.74907011]])
```

In [39]:
```
x_train.shape
```

Out[39]:
```
(10208, 16)
```

In [40]:
```
# create model
model = Sequential()
model.add(Dense(32*4, input_dim=16, activation='relu'))
model.add(Dense(32*4, activation='relu'))
model.add(Dense(32*4*2, activation='relu'))
model.add(Dense(32*4*4, activation='relu'))
model.add(Dense(32*4*4, activation='relu'))
model.add(Dense(32*4, activation='relu'))
model.add(Dense(7, activation='softmax'))
# Compile model
sgd = SGD(learning_rate=0.0001)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=
# Fit the model
model.fit(x_train, y_train, epochs=500, batch_size=75)
# evaluate the model
scores = model.evaluate(x_train, y_train)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

```
Epoch 1/500
137/137 [==============================] - 1s 3ms/step - loss: 1.9514
- accuracy: 0.1153
Epoch 2/500
137/137 [==============================] - 0s 3ms/step - loss: 1.9443
- accuracy: 0.1234
Epoch 3/500
137/137 [==============================] - 0s 3ms/step - loss: 1.9374
- accuracy: 0.1294
Epoch 4/500
137/137 [==============================] - 0s 3ms/step - loss: 1.9307
- accuracy: 0.1331
Epoch 5/500
137/137 [==============================] - 0s 3ms/step - loss: 1.9242
- accuracy: 0.1357
Epoch 6/500
137/137 [==============================] - 0s 3ms/step - loss: 1.9179
- accuracy: 0.1391
Epoch 7/500
137/137 [==============================] - 0s 3ms/step - loss: 1.9118
```

In [41]:
```
model.predict(x_train)
```

Out[41]:
```
array([[2.0989815e-05, 1.4930310e-09, 6.8191739e-06, ..., 9.9823415e-
01,
         6.1378152e-08, 1.7376345e-03],
        [1.5321679e-03, 6.1044661e-06, 2.1603652e-03, ..., 9.5996135e-
01,
         3.1019728e-05, 3.6222715e-02],
        [1.1129740e-03, 9.5645337e-05, 5.1572365e-06, ..., 1.5958047e-
05,
         9.9715269e-01, 1.1865281e-03],
        ...,
        [8.5209740e-06, 6.8891751e-07, 8.2414973e-09, ..., 6.2024220e-
08,
         9.9989355e-01, 3.6236128e-05],
        [1.1960589e-04, 1.0910860e-05, 3.0854500e-07, ..., 1.3035688e-
06,
         9.9943334e-01, 2.5588687e-04],
        [9.5249474e-01, 6.4030094e-03, 4.0540095e-02, ..., 3.8620483e-
04,
         7.5502176e-05, 1.0013752e-04]], dtype=float32)
```

In [63]:
```
#random forest approach
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(criterion='entropy')
```

In [64]:
```
model.fit(x_train, y_train)
```

Out[64]:
```
▾ RandomForestClassifier
RandomForestClassifier(criterion='entropy')
```

In [65]:
```
predictions = model.predict(x_train)
```

In [66]:
```
from sklearn.metrics import (
    accuracy_score,
    classification_report,
    confusion_matrix, auc, roc_curve
)
```

In [67]:
```
accuracy_score(y_train, predictions)
```

Out[67]:
```
1.0
```

In [68]:
```
confusion_matrix(y_train.argmax(axis=1), predictions.argmax(axis=1))
```

Out[68]:
```
array([[ 977,    0,    0,    0,    0,    0,    0],
       [   0,  398,    0,    0,    0,    0,    0],
       [   0,    0, 1217,    0,    0,    0,    0],
       [   0,    0,    0, 2676,    0,    0,    0],
       [   0,    0,    0,    0, 1426,    0,    0],
       [   0,    0,    0,    0,    0, 1534,    0],
       [   0,    0,    0,    0,    0,    0, 1980]])
```

In [69]:
```
print(classification_report(y_train.argmax(axis=1), predictions.argmax
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       977
           1       1.00      1.00      1.00       398
           2       1.00      1.00      1.00      1217
           3       1.00      1.00      1.00      2676
           4       1.00      1.00      1.00      1426
           5       1.00      1.00      1.00      1534
           6       1.00      1.00      1.00      1980

    accuracy                           1.00     10208
   macro avg       1.00      1.00      1.00     10208
weighted avg       1.00      1.00      1.00     10208
```

In [70]:
```
test_predictions = model.predict(x_test)
```

In [71]:
```
accuracy_score(y_test, test_predictions)
```

Out[71]:
```
0.9133117837202468
```

In [72]:
```
confusion_matrix(y_test.argmax(axis=1), test_predictions.argmax(axis=1
```

Out[72]:
```
array([[321,   0,  15,   0,   2,   0,   7],
       [  0, 124,   0,   0,   0,   0,   0],
       [ 22,   0, 382,   0,   5,   0,   4],
       [ 15,   0,   0, 803,   0,  11,  41],
       [ 10,   0,   4,   3, 477,   0,   8],
       [ 13,   0,   0,  16,   0, 451,  13],
       [ 18,   0,   0,  68,  10,   3, 557]])
```

In [73]:
```
print(classification_report(y_test.argmax(axis=1), test_predictions.ar
```

```
              precision    recall  f1-score   support

           0       0.80      0.93      0.86       345
           1       1.00      1.00      1.00       124
           2       0.95      0.92      0.94       413
           3       0.90      0.92      0.91       870
           4       0.97      0.95      0.96       502
           5       0.97      0.91      0.94       493
           6       0.88      0.85      0.87       656

    accuracy                           0.92      3403
   macro avg       0.93      0.93      0.93      3403
weighted avg       0.92      0.92      0.92      3403
```

In [ ]: