Neural Networks - intro

Part 1 - XOR

1. Using the XOR dataset below, train (400 epochs) a neural network (NN) using 2, 3, 4, and 5 hidden layers (where each layer has only 2 neurons). For each n layers, store the resulting accuracy along with n. Plot the results to find what the optimal number of layers is.

In [48]:
```
import tensorflow
```

In [49]:
```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD  #Stochastic Gradient Descent
from keras.optimizers import Adam

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

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

XOR Datset

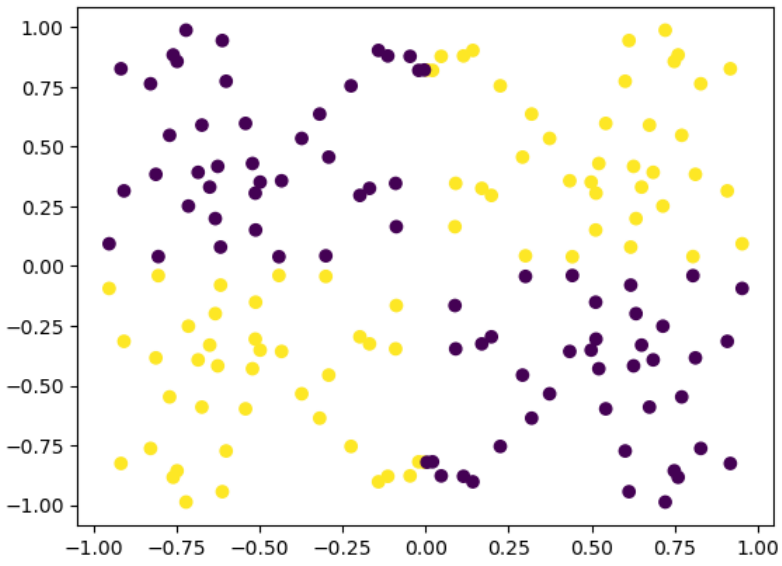
In [65]:
```
np.random.seed(seed=10)
n = 40
xx = np.random.random((n,1))
yy = np.random.random((n,1))
```

In [66]:
```
X = np.array([np.array([xx,-xx,-xx,xx]),np.array([yy,-yy,yy,-yy])]).reshape(2,4*n).T
y = np.array([np.ones([2*n]),np.zeros([2*n])]).reshape(4*n)
```

In [52]:
```
model = Sequential()
```

In [53]:
```
plt.scatter(*zip(*X), c=y)
```

Out[53]:
```
<matplotlib.collections.PathCollection at 0x1f3e973dbe0>
```

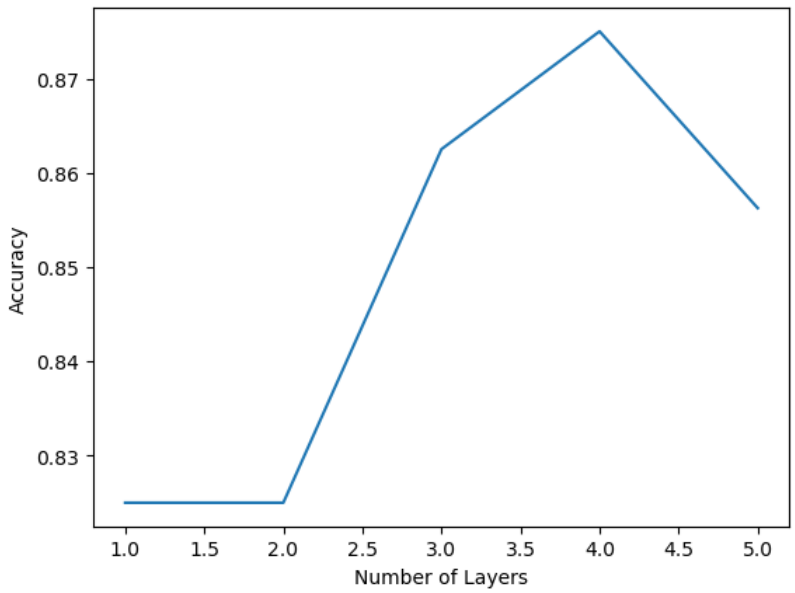


In [54]:
```
num_layers = [1,2,3,4,5]
scores = []
for num_layer in num_layers:
    # iterate over model.add the number of times dictated by num_layers
    for _ in range(num_layer):
        model.add(Dense(2, input_dim=2, activation='tanh'))
    model.add(Dense(1, activation='sigmoid'))
    adam = Adam(learning_rate=0.1)
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    model.fit(X, y, batch_size=2, epochs=400)

    score = model.evaluate(X, y)
    scores.append(score[1])
```

```
Epoch 377/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3971 - accuracy: 0.8562
Epoch 378/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3972 - accuracy: 0.8562
Epoch 379/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3971 - accuracy: 0.8562
Epoch 380/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3974 - accuracy: 0.8562
Epoch 381/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3972 - accuracy: 0.8562
Epoch 382/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3974 - accuracy: 0.8562
Epoch 383/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3974 - accuracy: 0.8562
Epoch 384/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3972 - accuracy: 0.8562
Epoch 385/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3971 - accuracy: 0.8562
Epoch 386/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3974 - accuracy: 0.8562
Epoch 387/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3973 - accuracy: 0.8562
Epoch 388/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3972 - accuracy: 0.8562
Epoch 389/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3972 - accuracy: 0.8562
Epoch 390/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3970 - accuracy: 0.8562
Epoch 391/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3971 - accuracy: 0.8562
Epoch 392/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3973 - accuracy: 0.8562
Epoch 393/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3971 - accuracy: 0.8562
Epoch 394/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3975 - accuracy: 0.8562
Epoch 395/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3972 - accuracy: 0.8562
Epoch 396/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3974 - accuracy: 0.8562
Epoch 397/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3975 - accuracy: 0.8562
Epoch 398/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3972 - accuracy: 0.8562
Epoch 399/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3972 - accuracy: 0.8562
Epoch 400/400
80/80 [==============================] - 0s 3ms/step - loss: 0.3980 - accuracy: 0.8562
5/5 [==============================] - 0s 3ms/step - loss: 0.3968 - accuracy: 0.8562
```

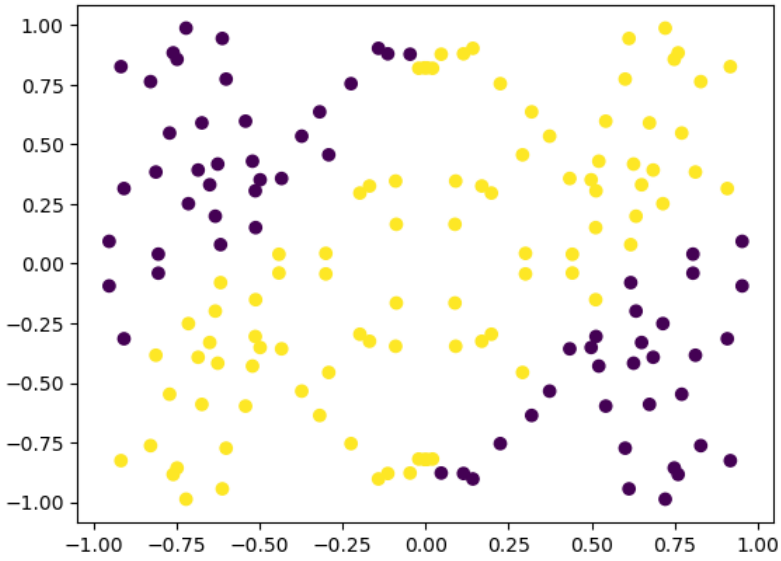
In [55]:
```
plt.figure()
plot = plt.plot(num_layers, scores)
plt.xlabel('Number of Layers')
plt.ylabel('Accuracy')
plt.show()
```



In [56]:
```
plt.scatter(*zip(*X), c=model.predict(X))
```

```
5/5 [==============================] - 0s 2ms/step
```

Out[56]:
```
<matplotlib.collections.PathCollection at 0x1f3e3e022b0>
```



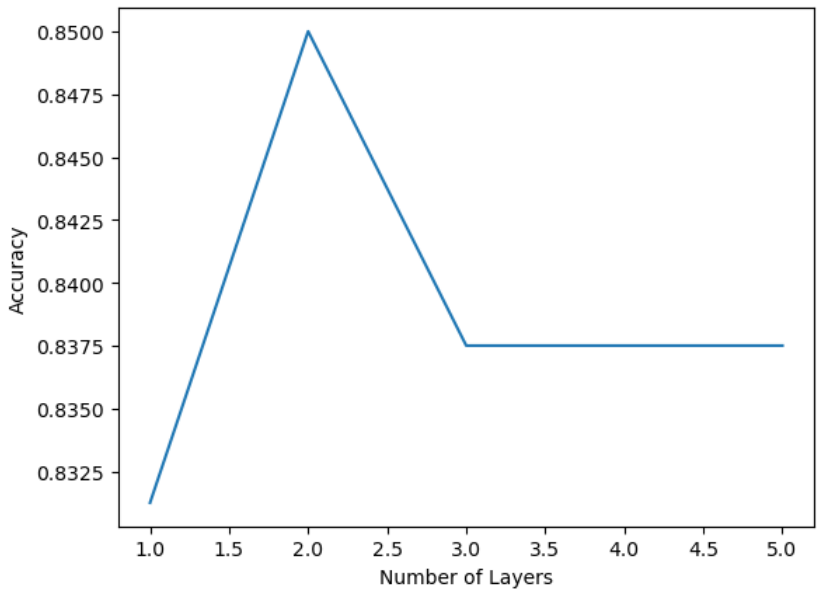
1. Repeat the above with 3 neurons in each Hidden layers. How do these results compare to the 2 neuron layers?

In [57]:
```
num_layers = [1,2,3,4,5]
scores = []
for num_layer in num_layers:
    # iterate over model.add the number of times dictated by num_layers
    for _ in range(num_layer):
        model.add(Dense(3, input_dim=2, activation='tanh'))
    model.add(Dense(1, activation='sigmoid'))
    adam = Adam(learning_rate=0.1)
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    model.fit(X, y, batch_size=2, epochs=400)

    score = model.evaluate(X, y)
    scores.append(score[1])
```

```
Epoch 377/400
80/80 [==============================] - 0s 3ms/step - loss: 0.4447 - accuracy: 0.8375
Epoch 378/400
80/80 [==============================] - 0s 3ms/step - loss: 0.4444 - accuracy: 0.8375
Epoch 379/400
80/80 [==============================] - 0s 4ms/step - loss: 0.4445 - accuracy: 0.8375
Epoch 380/400
80/80 [==============================] - 0s 3ms/step - loss: 0.4444 - accuracy: 0.8375
Epoch 381/400
80/80 [==============================] - 0s 4ms/step - loss: 0.4445 - accuracy: 0.8375
Epoch 382/400
80/80 [==============================] - 0s 3ms/step - loss: 0.4442 - accuracy: 0.8375
Epoch 383/400
80/80 [==============================] - 0s 4ms/step - loss: 0.4448 - accuracy: 0.8375
Epoch 384/400
80/80 [==============================] - 0s 3ms/step - loss: 0.4444 - accuracy: 0.8375
Epoch 385/400
80/80 [==============================] - 0s 4ms/step - loss: 0.4441 - accuracy: 0.8375
Epoch 386/400
80/80 [==============================] - 0s 4ms/step - loss: 0.4442 - accuracy: 0.8375
Epoch 387/400
80/80 [==============================] - 0s 3ms/step - loss: 0.4442 - accuracy: 0.8375
Epoch 388/400
80/80 [==============================] - 0s 3ms/step - loss: 0.4442 - accuracy: 0.8375
Epoch 389/400
80/80 [==============================] - 0s 3ms/step - loss: 0.4443 - accuracy: 0.8375
Epoch 390/400
80/80 [==============================] - 0s 3ms/step - loss: 0.4446 - accuracy: 0.8375
Epoch 391/400
80/80 [==============================] - 0s 3ms/step - loss: 0.4450 - accuracy: 0.8375
Epoch 392/400
80/80 [==============================] - 0s 3ms/step - loss: 0.4443 - accuracy: 0.8375
Epoch 393/400
80/80 [==============================] - 0s 3ms/step - loss: 0.4444 - accuracy: 0.8375
Epoch 394/400
80/80 [==============================] - 0s 4ms/step - loss: 0.4440 - accuracy: 0.8375
Epoch 395/400
80/80 [==============================] - 0s 4ms/step - loss: 0.4443 - accuracy: 0.8375
Epoch 396/400
80/80 [==============================] - 0s 4ms/step - loss: 0.4443 - accuracy: 0.8375
Epoch 397/400
80/80 [==============================] - 0s 4ms/step - loss: 0.4446 - accuracy: 0.8375
Epoch 398/400
80/80 [==============================] - 0s 4ms/step - loss: 0.4448 - accuracy: 0.8375
Epoch 399/400
80/80 [==============================] - 0s 4ms/step - loss: 0.4445 - accuracy: 0.8375
Epoch 400/400
80/80 [==============================] - 0s 4ms/step - loss: 0.4440 - accuracy: 0.8375
5/5 [==============================] - 1s 3ms/step - loss: 0.4435 - accuracy: 0.8375
```

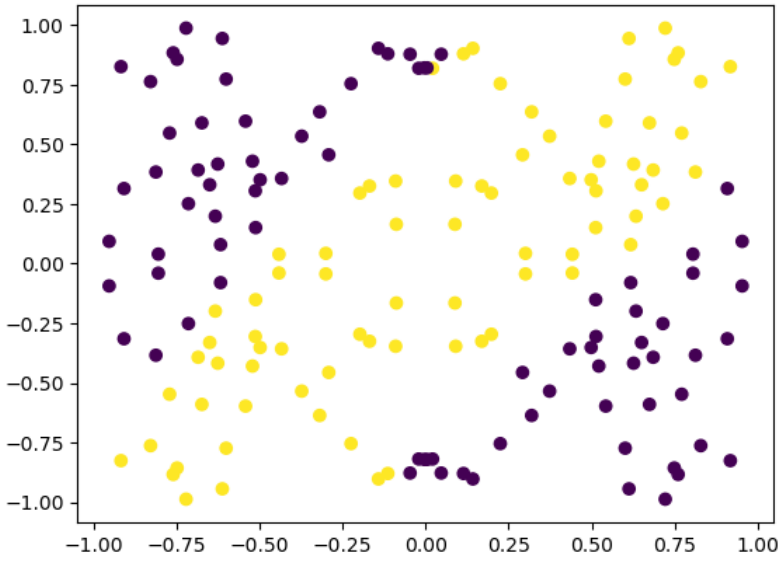
In [58]:
```
plt.figure()
plot = plt.plot(num_layers, scores)
plt.xlabel('Number of Layers')
plt.ylabel('Accuracy')
plt.show()
```



In [59]:
```
plt.scatter(*zip(*X), c=model.predict(X))
```

```
5/5 [==============================] - 0s 4ms/step
```

Out[59]:
```
<matplotlib.collections.PathCollection at 0x1f3e828aa90>
```



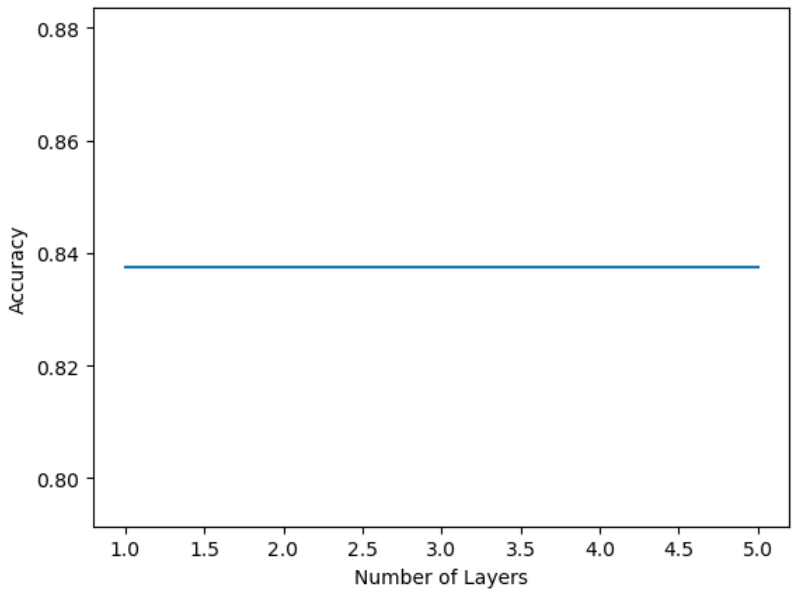
1. Repeat the above with 4 neurons in each Hidden layers. How do these results compare to the 2 and 3 neuron layers?

In [60]:
```
num_layers = [1,2,3,4,5]
scores = []
for num_layer in num_layers:
    # iterate over model.add the number of times dictated by num_layers
    for _ in range(num_layer):
        model.add(Dense(4, input_dim=2, activation='tanh'))
    model.add(Dense(1, activation='sigmoid'))
    adam = Adam(learning_rate=0.1)
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    model.fit(X, y, batch_size=2, epochs=400)

    score = model.evaluate(X, y)
    scores.append(score[1])
```

```
Epoch 377/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4450 - accuracy: 0.8375
Epoch 378/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4457 - accuracy: 0.8375
Epoch 379/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4445 - accuracy: 0.8375
Epoch 380/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4442 - accuracy: 0.8375
Epoch 381/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4445 - accuracy: 0.8375
Epoch 382/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4450 - accuracy: 0.8375
Epoch 383/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4452 - accuracy: 0.8375
Epoch 384/400
80/80 [==============================] - 0s 4ms/step - loss: 0.4442 - accuracy: 0.8375
Epoch 385/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4450 - accuracy: 0.8375
Epoch 386/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4442 - accuracy: 0.8375
Epoch 387/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4448 - accuracy: 0.8375
Epoch 388/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4445 - accuracy: 0.8375
Epoch 389/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4446 - accuracy: 0.8375
Epoch 390/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4445 - accuracy: 0.8375
Epoch 391/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4444 - accuracy: 0.8375
Epoch 392/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4461 - accuracy: 0.8375
Epoch 393/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4446 - accuracy: 0.8375
Epoch 394/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4450 - accuracy: 0.8375
Epoch 395/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4442 - accuracy: 0.8375
Epoch 396/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4449 - accuracy: 0.8375
Epoch 397/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4443 - accuracy: 0.8375
Epoch 398/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4443 - accuracy: 0.8375
Epoch 399/400
80/80 [==============================] - 0s 5ms/step - loss: 0.4444 - accuracy: 0.8375
Epoch 400/400
80/80 [==============================] - 0s 6ms/step - loss: 0.4444 - accuracy: 0.8375
5/5 [==============================] - 1s 4ms/step - loss: 0.4435 - accuracy: 0.8375
```

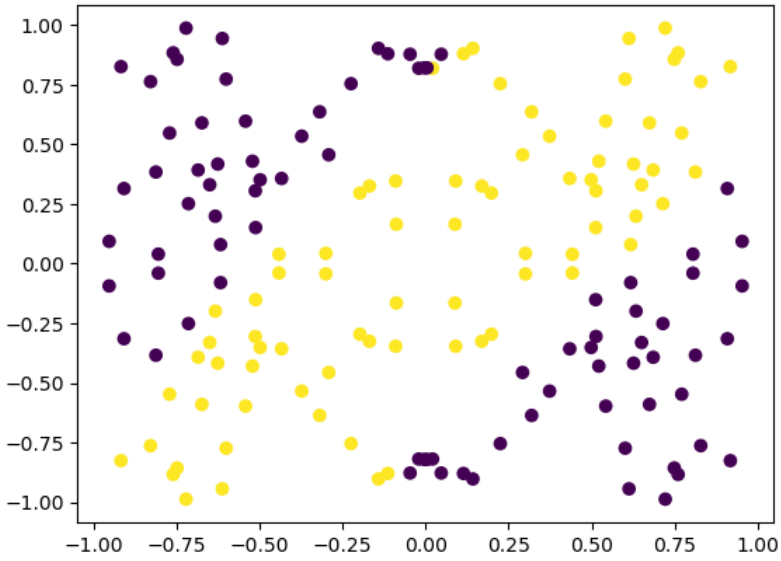
In [62]:
```
plt.figure()
plot = plt.plot(num_layers, scores)
plt.xlabel('Number of Layers')
plt.ylabel('Accuracy')
plt.show()
```



In [63]:
```
plt.scatter(*zip(*X), c=model.predict(X))
```

```
5/5 [==============================] - 1s 5ms/step
```

Out[63]:
```
<matplotlib.collections.PathCollection at 0x1f3c7ea5df0>
```



1. 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/)

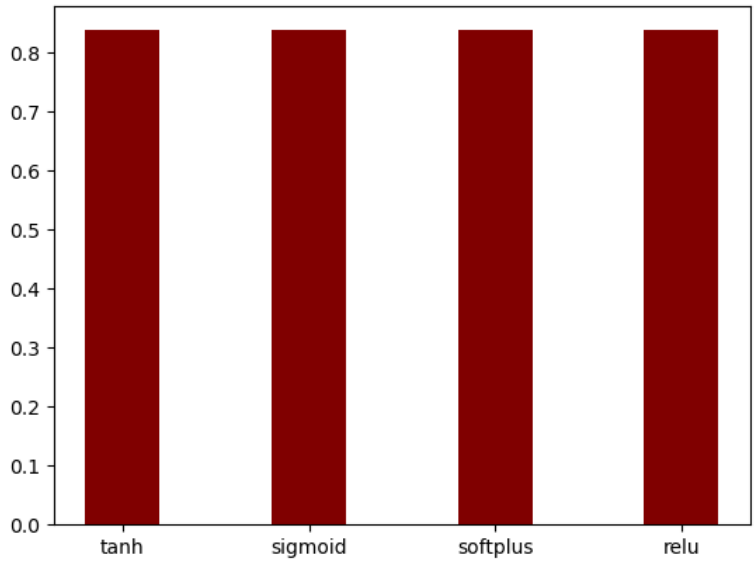
In [71]:
```
ideal_n = 4
ideal_k = 2
activations = ["tanh", "sigmoid", "softplus", "relu"]
scores = []
    # iterate over model.add the number of times dictated by num_layers
for activation in activations:
    for _ in range(ideal_n):
        model.add(Dense(ideal_k, input_dim=2, activation=activation))
    model.add(Dense(1, activation="sigmoid"))
    adam = Adam(learning_rate=0.01)
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    model.fit(X, y, batch_size=2, epochs=400)

    score = model.evaluate(X, y)
    scores.append(score[1])
    print(str(score[1]) + " accuracy for " + str(activation) + " model")
```

```
Epoch 389/400
80/80 [==============================] - 1s 7ms/step - loss: 0.6933 - accuracy: 0.4750
Epoch 390/400
80/80 [==============================] - 1s 7ms/step - loss: 0.6933 - accuracy: 0.4625
Epoch 391/400
80/80 [==============================] - 1s 7ms/step - loss: 0.6933 - accuracy: 0.4125
Epoch 392/400
80/80 [==============================] - 1s 7ms/step - loss: 0.6933 - accuracy: 0.4375
Epoch 393/400
80/80 [==============================] - 1s 7ms/step - loss: 0.6933 - accuracy: 0.5000
Epoch 394/400
80/80 [==============================] - 1s 7ms/step - loss: 0.6933 - accuracy: 0.4625
Epoch 395/400
80/80 [==============================] - 1s 7ms/step - loss: 0.6933 - accuracy: 0.4375
Epoch 396/400
80/80 [==============================] - 1s 7ms/step - loss: 0.6932 - accuracy: 0.4875
Epoch 397/400
80/80 [==============================] - 1s 7ms/step - loss: 0.6932 - accuracy: 0.5000
Epoch 398/400
80/80 [==============================] - 1s 7ms/step - loss: 0.6932 - accuracy: 0.4625
Epoch 399/400
80/80 [==============================] - 1s 7ms/step - loss: 0.6932 - accuracy: 0.5000
Epoch 400/400
80/80 [==============================] - 1s 8ms/step - loss: 0.6934 - accuracy: 0.4875
5/5 [==============================] - 1s 6ms/step - loss: 0.6931 - accuracy: 0.5000
0.5 accuracy for relu model
```

In [68]:
```
plt.bar(activations, scores, color ='maroon', width = 0.4)
```

Out[68]:
```
<BarContainer object of 4 artists>
```



1. gain with the most optimal setup, try other optimizers (instead of `SGD`) and report on the loss score. (https://keras.io/optimizers/)

In [72]:
```
model.add(Dense(2, input_dim=2, activation="tanh"))
model.add(Dense(2, input_dim=2, activation="tanh"))
model.add(Dense(2, input_dim=2, activation="tanh"))
model.add(Dense(2, input_dim=2, activation="tanh"))
model.add(Dense(1, activation="sigmoid"))
opt = SGD(learning_rate=0.01)
model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
model.fit(X, y, batch_size=2, epochs=400)

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

```
Epoch 388/400
80/80 [==============================] - 1s 10ms/step - loss: 0.6951 - accuracy: 0.4125
Epoch 389/400
80/80 [==============================] - 1s 9ms/step - loss: 0.6948 - accuracy: 0.4000
Epoch 390/400
80/80 [==============================] - 1s 9ms/step - loss: 0.6948 - accuracy: 0.4625
Epoch 391/400
80/80 [==============================] - 1s 9ms/step - loss: 0.6947 - accuracy: 0.4625
Epoch 392/400
80/80 [==============================] - 1s 9ms/step - loss: 0.6951 - accuracy: 0.4125
Epoch 393/400
80/80 [==============================] - 1s 9ms/step - loss: 0.6947 - accuracy: 0.4375
Epoch 394/400
80/80 [==============================] - 1s 9ms/step - loss: 0.6948 - accuracy: 0.4750
Epoch 395/400
80/80 [==============================] - 1s 9ms/step - loss: 0.6950 - accuracy: 0.4375
Epoch 396/400
80/80 [==============================] - 1s 9ms/step - loss: 0.6948 - accuracy: 0.4500
Epoch 397/400
80/80 [==============================] - 1s 9ms/step - loss: 0.6949 - accuracy: 0.4500
Epoch 398/400
80/80 [==============================] - 1s 9ms/step - loss: 0.6952 - accuracy: 0.5000
Epoch 399/400
80/80 [==============================] - 1s 9ms/step - loss: 0.6950 - accuracy: 0.4375
Epoch 400/400
80/80 [==============================] - 1s 8ms/step - loss: 0.6946 - accuracy: 0.5000
5/5 [==============================] - 1s 5ms/step - loss: 0.6932 - accuracy: 0.5000
```

It looks like 4 layers with 2 neurons gives the highest accuracy after 400 epochs. The activation model doesn't actually seem to matter, but something is off about the results because the first activation in the list should be identical to the model run in question 2 and it got very different results. The test for optimizers suffers from the same problem.

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://keras.io/

Using 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 [74]:
```
# load pima indians dataset
dataset = np.loadtxt("../data/pima-indians-diabetes.data", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
```

In [75]:
```
# create model
model = Sequential()
model.add(Dense(2, input_dim=8, activation='tanh'))
model.add(Dense(4, activation='tanh'))
model.add(Dense(1, activation='tanh'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, epochs=500, batch_size=15)
# evaluate the model
scores = model.evaluate(X, Y)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

```
Epoch 474/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 475/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 476/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 477/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 478/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 479/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 480/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 481/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 482/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 483/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 484/500
52/52 [==============================] - 0s 2ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 485/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 486/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 487/500
52/52 [==============================] - 0s 2ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 488/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 489/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 490/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 491/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 492/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 493/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 494/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 495/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 496/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 497/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 498/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 499/500
52/52 [==============================] - 0s 2ms/step - loss: 5.3827 - accuracy: 0.6510
Epoch 500/500
52/52 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510
24/24 [==============================] - 0s 1ms/step - loss: 5.3827 - accuracy: 0.6510

accuracy: 65.10%
```

It doesn't seem to matter the number of neurons or the number of layers, what the optimizer is, or the batch size. The activations don't seem to make much of a difference with the exception of "softmax" which lowers the accuracy when used a majority of the time. Some looking around on StackOverflow for ideas seems to suggest that lower batch sizes and high learning rates may produce more unpredictable results, especially for smaller sample sizes, so I raised the batch size to 15 and kept the learning rate at it's default of 0.001.

In [ ]: