Instructions

This is more of a follow-along lab. We'll show you the steps to build the network. However, at the end of the lab you'll be given the opportunity to improve the model, and try to improve on its performance. Here are the main steps in this lab.

Studying the data

The dataset has the following columns:

- Student GPA (grades)
- Score on the GRE (test)
- Class rank (1-4)

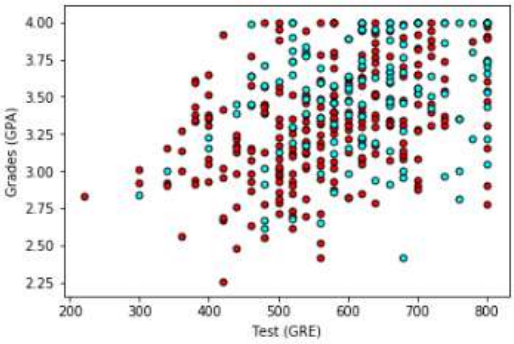
First, let's start by looking at the data. For that, we'll use the read_csv function in pandas.

```
import pandas as pd
data = pd.read_csv('student_data.csv')
print(data)
```

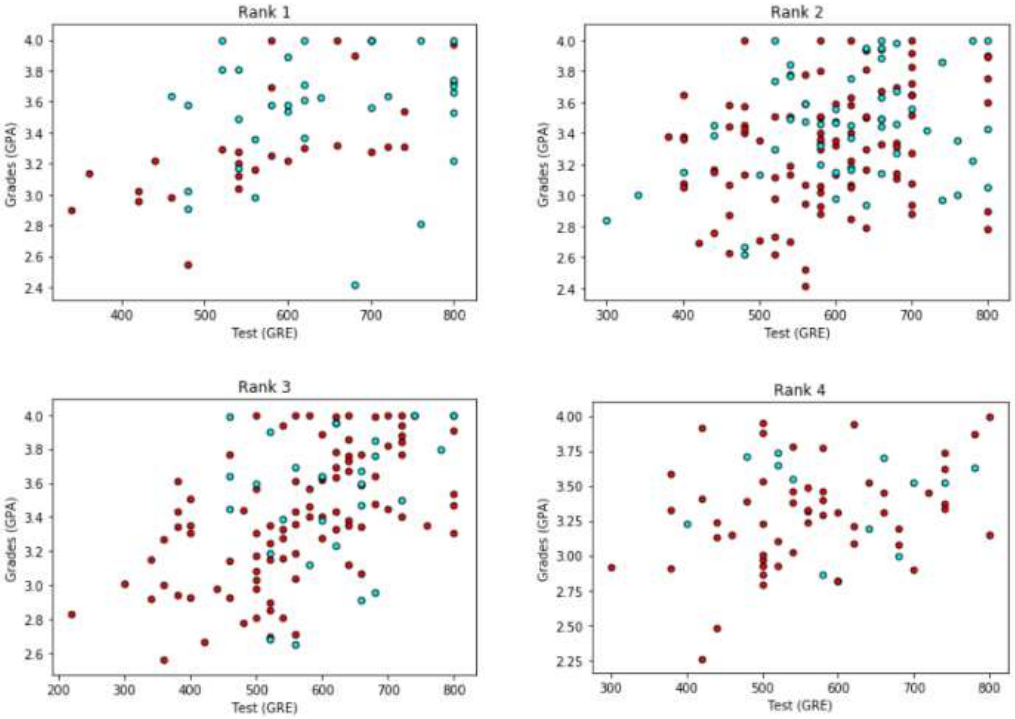
	admit	gre	gpa	rank
0	0	380	3.61	3
1	1	660	3.67	3
2	1	800	4.00	1
3	1	640	3.19	4
4	0	520	2.93	4
5	1	760	3.00	2
6	1	560	2.98	1
7	0	400	3.08	2
8	1	540	3.39	3
9	0	700	3.92	2

Here we can see that the first column is the label `y`, which corresponds to acceptance/rejection. Namely, a label of `1` means the student got accepted, and a label of `0` means the student got rejected.

When we plot the data, we get the following graphs, which show that unfortunately, the data is not as nicely separable as we'd hope:



So one thing we can do is make one graph for each of the 4 ranks. In that case, we get this:



Pre-processing the data

Ok, there's a bit more hope here. It seems like the better grades and test scores the student has, the more likely they are to be accepted. And the rank has something to do with it. So what we'll do is, we'll one-hot encode the rank, and our 6 input variables will be:

- Test (GPA)
- Grades (GRE)
- Rank 1
- Rank 2
- Rank 3
- Rank 4.

The last 4 inputs will be binary variables that have a value of 1 if the student has that rank, or 0 otherwise.

So, first things first, let's notice that the test scores have a range of 800, while the grades have a range of 4. This is a huge discrepancy, and it will affect our training. Normally, the best thing to do is to normalize the scores so they are between 0 and 1. We can do this as follows:

```
data["gre"] = data["gre"]/800
data["gpa"] = data["gpa"]/4.0
```

Now, we split our data input into X, and the labels y , and one-hot encode the output, so it appears as two classes (accepted and not accepted).

```
X = np.array(data)[:,1:]
y = keras.utils.to_categorical(np.array(data["admit"]))
```

Building the model architecture

And finally, we define the model architecture. We can use different architectures, but here's an example:

```
model = Sequential()
model.add(Dense(128, input_dim=6))
model.add(Activation('sigmoid'))
model.add(Dense(32))
model.add(Activation('sigmoid'))
model.add(Dense(2))
model.add(Activation('sigmoid'))
model.compile(loss = 'categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

The error function is given by `categorical_crossentropy`, which is the one we've been using, but there are other options. There are several optimizers which you can choose from in order to improve your training. Here we use *adam*, but there are others that are useful, such as *rmsprop*. These use a variety of techniques that we'll outline in upcoming pages in this lesson.

The model summary will tell us the following:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_57 (Dense)             (None, 128)               896
_________________________________________________________________
activation_50 (Activation)   (None, 128)               0
_________________________________________________________________
dense_58 (Dense)             (None, 32)                4128
_________________________________________________________________
activation_51 (Activation)   (None, 32)                0
_________________________________________________________________
dense_59 (Dense)             (None, 2)                 66
_________________________________________________________________
activation_52 (Activation)   (None, 2)                 0
=================================================================
Total params: 5,090.0
Trainable params: 5,090.0
Non-trainable params: 0.0
_________________________________________________________________
```

Training the model

Now, we train the model, with 1000 epochs. Don't worry about the batch_size, we'll learn about it soon.

```
model.fit(X_train, y_train, epochs=1000, batch_size=100, verbose=0)
```

Evaluating the model

And finally, we can evaluate our model.

```
score = model.evaluate(X_train, y_train)
```

Results may vary, but you should get somewhere over 70% accuracy.