Applied Deep Learning - Part 1: Artificial Neural Networks

Arden Dertat Aug 8, 2017 · 23 min read

Overview

Welcome to the Applied Deep Learning tutorial series. We will do a detailed analysis of several deep learning techniques starting with Artificial Neural Networks (ANN), in particular Feedforward Neural Networks. What separates this tutorial from the rest you can find online is that we'll take a hands-on approach with plenty of code examples and visualization. I won't go into too much math and theory behind these models to keep the focus on application.

We will use the Keras deep learning framework, which is a high level API on top of Tensorflow. Keras is becoming super popular recently because of its simplicity. It's very easy to build complex models and iterate rapidly. I also used barebone Tensorflow, and actually struggled quite a bit. After trying out Keras I'm not going back.

Here's the table of contents. First an overview of ANN and the intuition behind these deep models. Then we will start simple with Logistic Regression, mainly to get familiar with Keras. Then we will train deep neural nets and demonstrate how they outperform linear models. We will compare the models on both binary and multiclass classification datasets.

The code for this article is available here as a Jupyter notebook, feel free to download and try it out yourself.

I think you'll learn a lot from this article. You don't need to have prior knowledge of deep learning, only some basic familiarity with general machine learning. So let's begin…

1. ANN Overview

1.1) Introduction

Artificial Neural Networks (ANN) are multi-layer fully-connected neural nets that look like the figure below. They consist of an input layer, multiple hidden layers, and an output layer. Every node in one layer is connected to

every other node in the next layer. We make the network deeper by increasing the number of hidden layers.

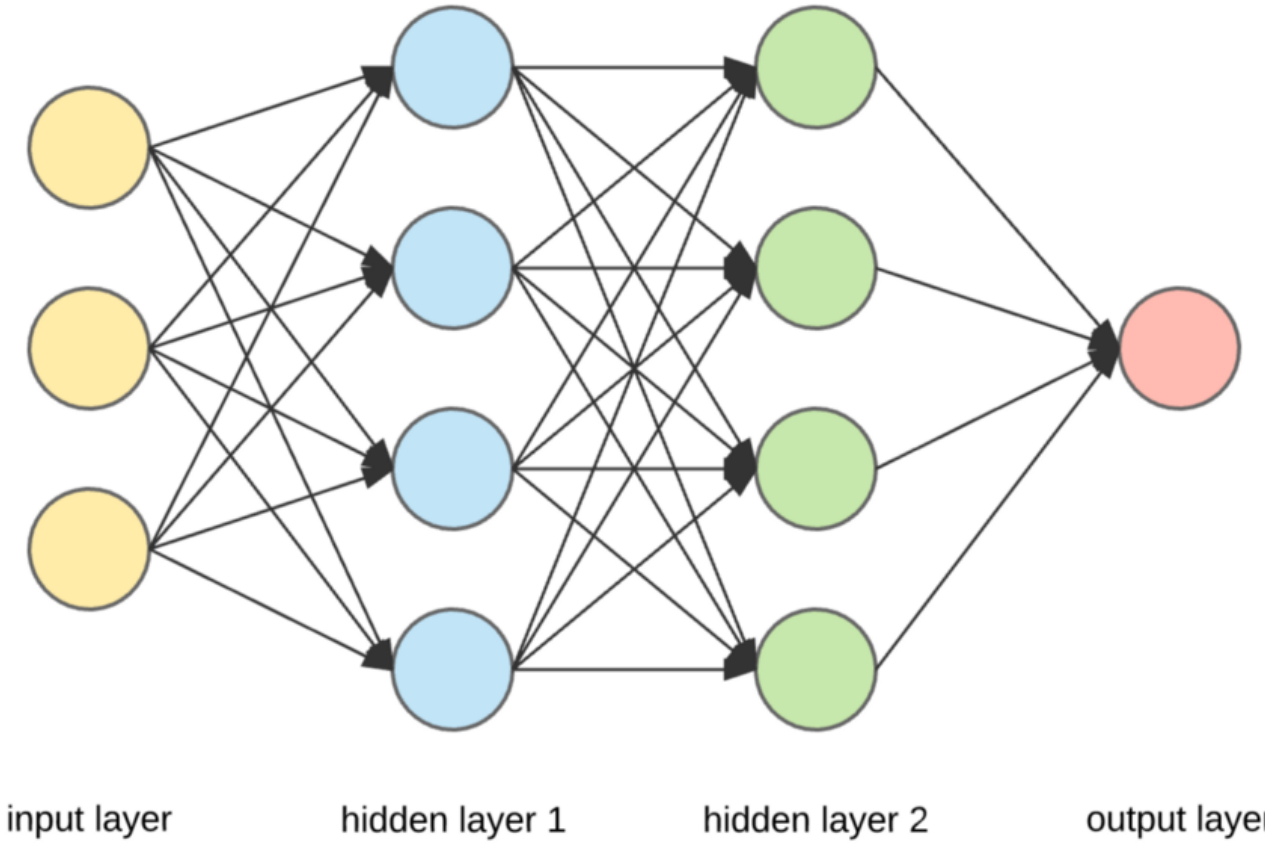


Figure 1

If we zoom in to one of the hidden or output nodes, what we will encounter is the figure below.

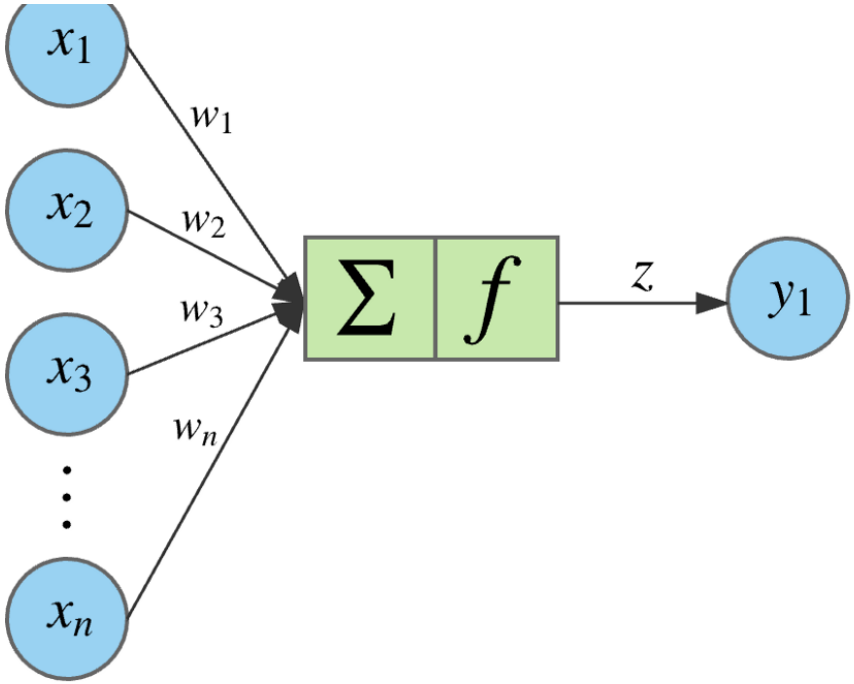


Figure 2

A given node takes the weighted sum of its inputs, and passes it through a non-linear activation function. This is the output of the node, which then becomes the input of another node in the next layer. The signal flows from left to right, and the final output is calculated by performing this procedure for all the nodes. Training this deep neural network means learning the weights associated with all the edges.

The equation for a given node looks as follows. The weighted sum of its inputs passed through a non-linear activation function. It can be

represented as a vector dot product, where *n* is the number of inputs for the node.

$$z = f(x \cdot w) = f\left(\sum_{i=1}^{n} x_i w_i\right)$$

$$x \in d_{1\times n},\ w \in d_{n\times 1},\ z \in d_{1\times 1}$$

I omitted the *bias* term for simplicity. Bias is an input to all the nodes and always has the value 1. It allows to shift the result of the activation function to the left or right. It also helps the model to train when all the input features are 0. If this sounds complicated right now you can safely ignore the bias terms. For completeness, the above equation looks as follows with the bias included.

$$z = f(b + x \cdot w) = f\left(b + \sum_{i=1}^{n} x_i w_i\right)$$

$$x \in d_{1\times n},\ w \in d_{n\times 1},\ b \in d_{1\times 1},\ z \in d_{1\times 1}$$

So far we have described the *forward pass*, meaning given an input and weights how the output is computed. After the training is complete, we only run the forward pass to make the predictions. But we first need to train our model to actually learn the weights, and the training procedure works as follows:

- Randomly initialize the weights for all the nodes. There are smart initialization methods which we will explore in another article.
- For every training example, perform a forward pass using the current weights, and calculate the output of each node going from left to right. The final output is the value of the last node.
- Compare the final output with the actual target in the training data, and measure the error using a *loss function*.
- Perform a *backwards pass* from right to left and propagate the error to every individual node using *backpropagation*. Calculate each weight's contribution to the error, and adjust the weights accordingly using *gradient descent*. Propagate the error gradients back starting from the last layer.

Backpropagation with gradient descent is literally the "magic" behind the deep learning models. It's a rather long topic and involves some calculus, so

we won't go into the specifics in this applied deep learning series. For a detailed explanation of gradient descent refer here. A basic overview of backpropagation is available here. For a detailed mathematical treatment refer here and here. And for more advanced optimization algorithms refer here.

In the standard ML world this feed forward architecture is known as the *multilayer perceptron*. The difference between the ANN and perceptron is that ANN uses a non-linear activation function such as *sigmoid* but the perceptron uses the step function. And that non-linearity gives the ANN its great power.

1.2) Intuition

There's a lot going on already, even with the basic forward pass. Now let's simplify this, and understand the intuition behind it.

> *Essentially what each layer of the ANN does is a non-linear transformation of the input from one vector space to another.*

Let's use the ANN in Figure 1 above as an example. We have a 3-dimensional input corresponding to a vector in 3D space. We then pass it

through two hidden layers with 4 nodes each. And the final output is a 1D vector or a scalar.

So if we visualize this as a sequence of vector transformations, we first map the 3D input to a 4D vector space, then we perform another transformation to a new 4D space, and the final transformation reduces it to 1D. This is just a chain of matrix multiplications. The forward pass performs these matrix dot products and applies the activation function element-wise to the result. The figure below only shows the weight matrices being used (not the activations).

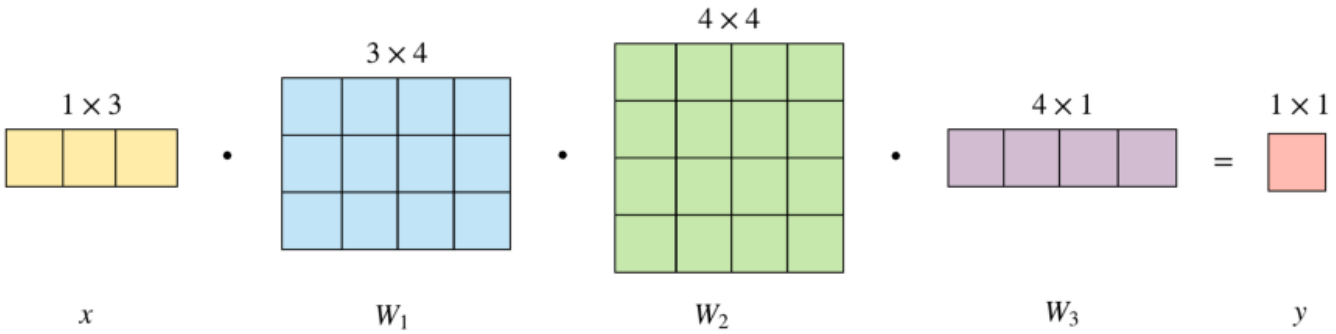


Figure 3

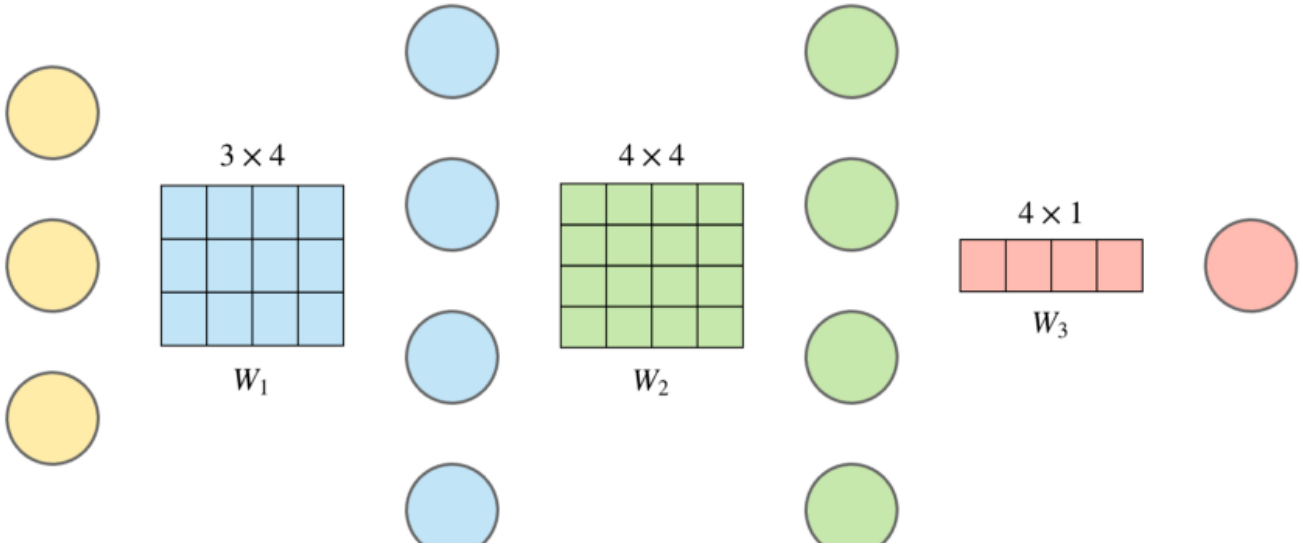
The input vector *x* has 1 row and 3 columns. To transform it into a 4D space, we need to multiply it with a *3x4* matrix. Then to another 4D space, we

multiply with a *4x4* matrix. And finally to reduce it to a 1D space, we use a *4x1* matrix.

Notice how the dimensions of the matrices represent the input and output dimensions of a layer. The connection between a layer with 3 nodes and 4 nodes is a matrix multiplication using a *3x4* matrix.

These matrices represent the weights that define the ANN. To make a prediction using the ANN on a given input, we only need to know these weights and the activation function (and the biases), nothing more. We train the ANN via backpropagation to "learn" these weights.

If we put everything together it looks like the figure below.



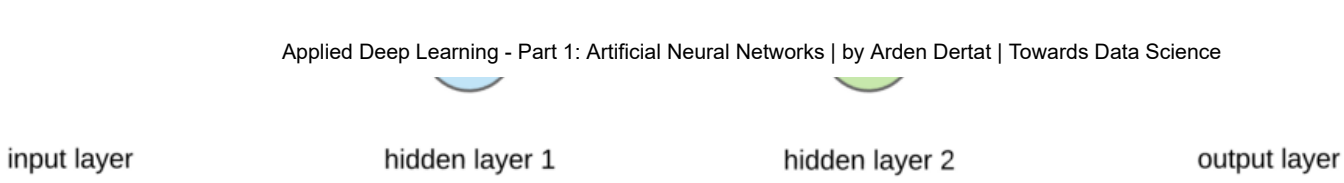


Figure 4

A fully connected layer between 3 nodes and 4 nodes is just a matrix multiplication of the *1x3* input vector (yellow nodes) with the *3x4* weight matrix *W1*. The result of this dot product is a *1x4* vector represented as the blue nodes. We then multiply this *1x4* vector with a *4x4* matrix *W2,* resulting in a *1x4* vector, the green nodes. And finally a using a *4x1* matrix *W3* we get the output.

We have omitted the activation function in the above figures for simplicity. In reality after every matrix multiplication, we apply the activation function to each element of the resulting matrix. More formally

$$a_1 = f(x \cdot W_1)$$

$$a_2 = f(a_1 \cdot W_2)$$

$$y = f(a_2 \cdot W_3)$$

$$\Downarrow$$

$$y = f(\ f(\ f(x \cdot W_1) \cdot W_2) \cdot W_3)$$

Equation 2

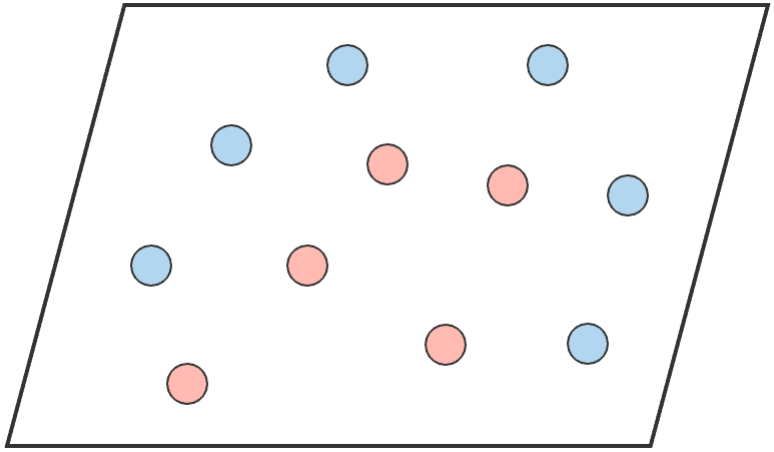
The output of the matrix multiplications go through the activation function f. In case of the sigmoid function, this means taking the sigmoid of each element in the matrix. We can see the chain of matrix multiplications more clearly in the equations.

1.3) Reasoning

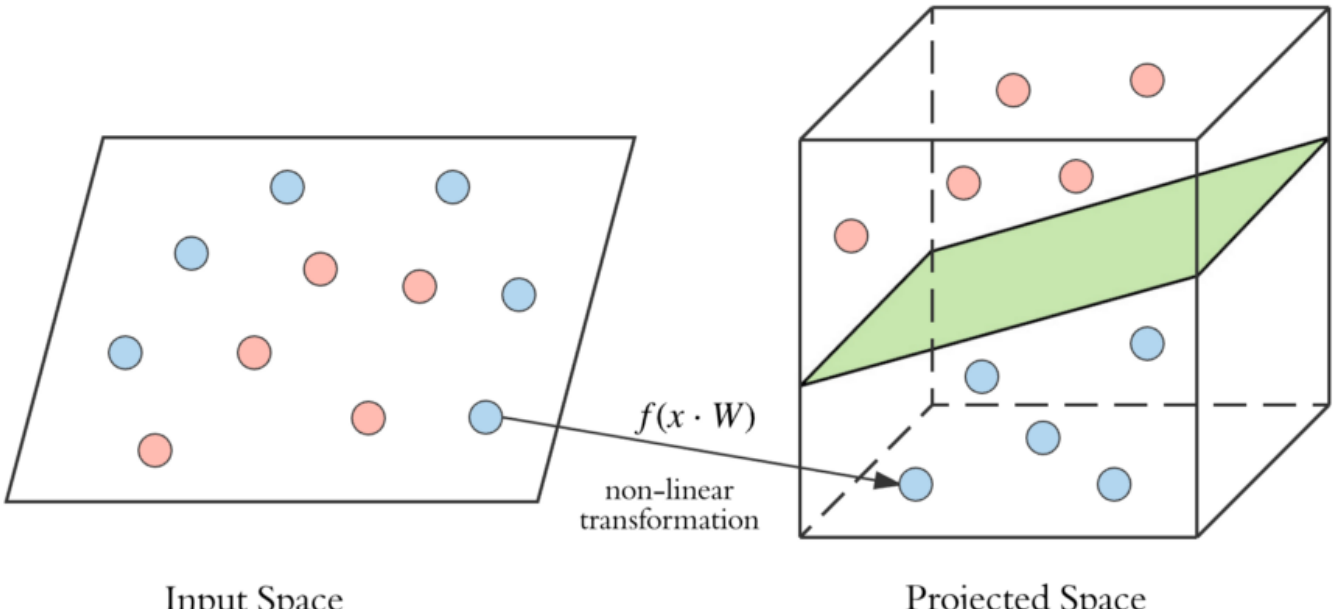
So far we talked about what deep models are and how they work, but why do we need to go deep in the first place?

We saw that a layer of ANN just performs a non-linear transformation of its inputs from one vector space to another. If we take a classification problem as an example, we want to separate out the classes by drawing a decision boundary. The input data in its given form is not separable. By performing non-linear transformations at each layer, we are able to project the input to a new vector space, and draw a complex decision boundary to separate the classes.

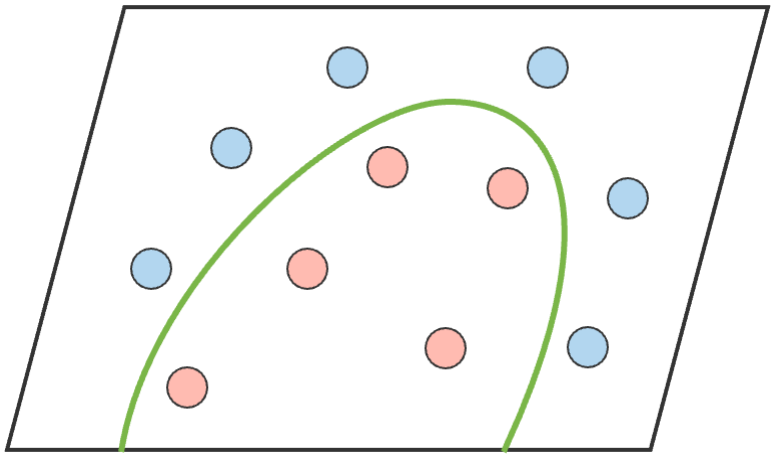
Let's visualize what we just described with a concrete example. Given the following data we can see that it isn't linearly separable.



So we project it to a higher dimensional space by performing a non-linear transformation, and then it becomes linearly separable. The green hyperplane is the decision boundary.



This is equivalent to drawing a complex decision boundary in the original input space.



> *So the main benefit of having a deeper model is being able to do more non-linear transformations of the input and drawing a more complex decision boundary.*

As a summary, ANNs are very flexible yet powerful deep learning models. They are universal function approximators, meaning they can model any complex function. There has been an incredible surge on their popularity recently due to a couple of reasons: clever tricks which made training these

models possible, huge increase in computational power especially GPUs and distributed training, and vast amount of training data. All these combined enabled deep learning to gain significant traction.

This was a brief introduction, there are tons of great tutorials online which cover deep neural nets. For reference, I highly recommend this paper. It's a fantastic overview of deep learning and Section 4 covers ANN. Another great reference is this book which is available online.

2. Logistic Regression

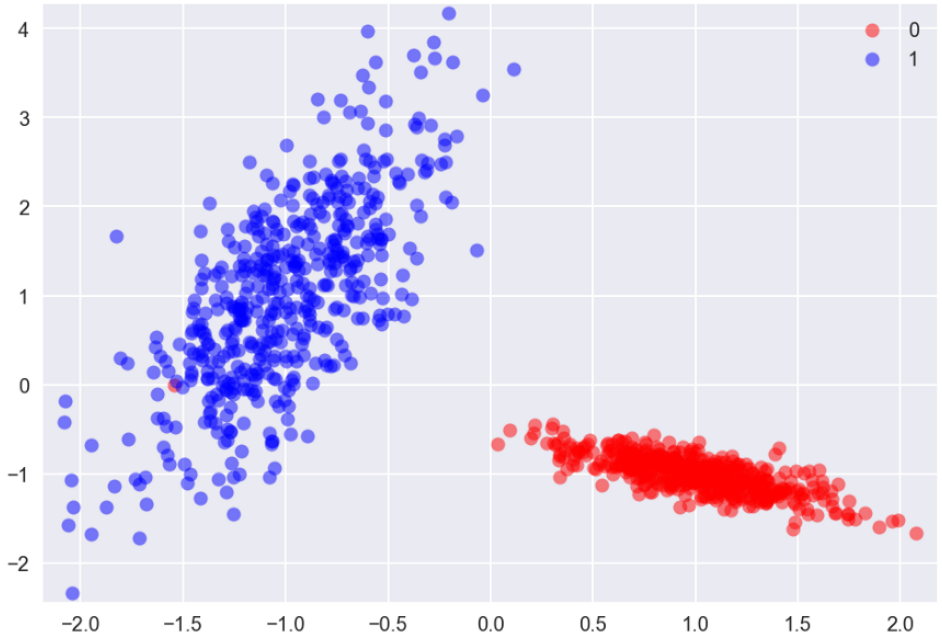
Despite its name, logistic regression (LR) is a binary *classification* algorithm. It's the most popular technique for 0/1 classification. On a 2 dimensional (2D) data LR will try to draw a straight line to separate the classes, that's where the term *linear model* comes from. LR works with any number of dimensions though, not just two. For 3D data it'll try to draw a 2D plane to separate the classes. This generalizes to N dimensional data and N-1 dimensional hyperplane separator. If you have a supervised binary classification problem, given an input data with multiple columns and a binary 0/1 outcome, LR is the first method to try. In this section we will focus on 2D data since it's easier to visualize, and in another tutorial we will focus on multidimensional input.

1.1) Linearly Separable Data

First let's start with an easy example. 2D linearly separable data. We are using the scikit-learn *make_classification* method to generate our data and using a helper function to visualize it.

```
X, y = make_classification(n_samples=1000, n_features=2, n_redundant=0,
                           n_informative=2, random_state=7, n_clusters_per_class=1)
plot_data(X, y)
```

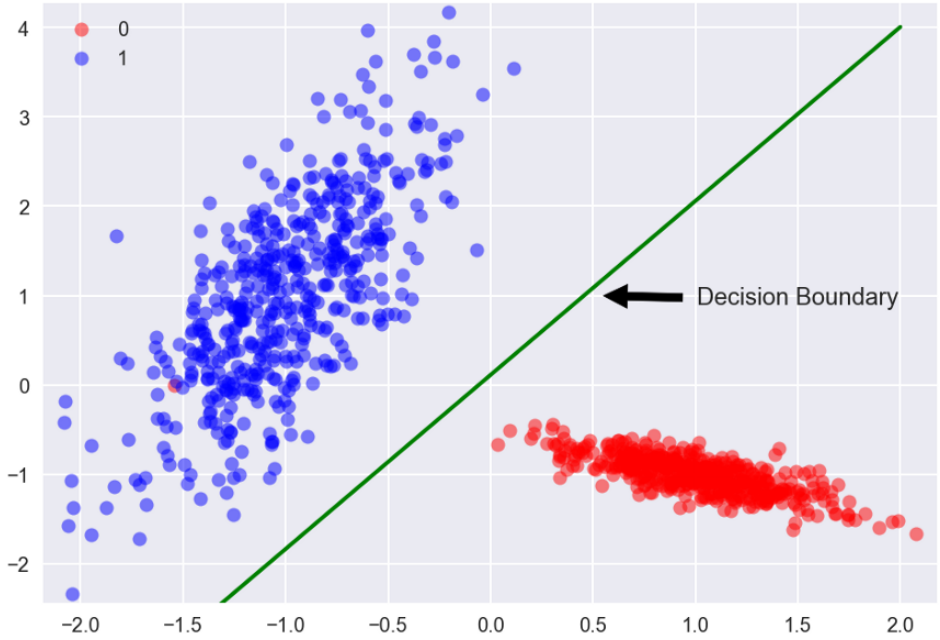
linearly_separable_data_1.py hosted with ♡ by GitHub view raw



There is a *LogisticRegression* classifier available in scikit-learn, I won't go into too much detail here since our goal is to learn building models with Keras. But here's how to train an LR model, using the *fit* function just like any other model in scikit-learn. We see the linear decision boundary as the green line.

```
lr = LogisticRegression()
lr.fit(X, y)
```

linearly_separable_data_2.py hosted with ♡ by GitHub view raw



As we can see the data is linearly separable. We will now train the same logistic regression model with Keras to predict the class membership of every input point. To keep things simple for now, we won't perform the standard practices of separating out the data to training and test sets, or performing k-fold cross-validation.

Keras has great documentation, check it out for a more detailed description of its API. Here's the code for training the model, let's go over it step by step below.

```
model = Sequential()
model.add(Dense(units=1, input_shape=(2,), activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

history = model.fit(x=X, y=y, verbose=0, epochs=50)
plot_loss_accuracy(history)
```

linearly_separable_data_3.py hosted with ♡ by GitHub view raw

We will use the *Sequential* model API available here. The Sequential model allows us to build deep neural networks by stacking layers one on top of another. Since we're now building a simple logistic regression model, we will have the input nodes directly connected to output node, without any hidden layers. Note that the LR model has the form $y=f(xW)$ where f is the

sigmoid function. Having a single output layer being directly connected to the input reflects this function.

Quick clarification to disambiguate the terms being used. In neural networks literature, it's common to talk about input nodes and output nodes. This may sound strange at first glance, what's an input "node" per se? When we say input nodes, we're talking about the features of a given training example. In our case we have 2 features, the x and y coordinates of the points we plotted above, so we have 2 input nodes. You can simply think of it as a vector of 2 numbers. What about the output node then? The output of the logistic regression model is a single number, the probability of an input data point belonging to class 1. In other words $P(class=1)$. The probability of an input point belonging to class 0 is then $P(class=0)=1-P(class=1)$. So you can simply think of the output node as a vector with a single number (or simply a scalar) between 0 and 1.

In Keras we don't add layers corresponding to input nodes, we only do for hidden and output nodes. In our current model, we don't have any hidden layers, the input nodes are directly connected to the output node. This means our neural network definition in Keras will just have one layer with one node, corresponding to the output node.

```
model = Sequential()
model.add(Dense(units=1, input_shape=(2,), activation='sigmoid'))
```

The *Dense* function in Keras constructs a fully connected neural network layer, automatically initializing the weights as biases. It's a super useful function that you will see being used everywhere. The function arguments are defined as follows:

- *units*: The first argument, representing number of nodes in this layer. Since we're constructing the output layer, and we said it has only one node, this value is 1.
- *input_shape*: The first layer in Keras models need to specify the input dimensions. The subsequent layers (which we don't have here but we will in later sections) don't need to specify this argument because Keras can infer the dimensions automatically. In this case our input dimensionality is 2, the x and y coordinates. The input_shape parameter expects a vector, so in our case it's a tuple with one number.
- *activation*: The activation function of a logistic regression model is the *logistic* function, or alternatively called the *sigmoid*. We will explore different activation functions, where to use them and why in another tutorial.

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=
['accuracy'])
```

We then compile the model with the *compile* function. This creates the neural network model by specifying the details of the learning process. The model hasn't been trained yet. Right now we're just declaring the optimizer to use and the loss function to minimize. The arguments for the compile function are defined as follows:

- *optimizer*: Which optimizer to use in order to minimize the loss function. There are a lot of different optimizers, most of them based on gradient descent. We will explore different optimizers in another tutorial. For now we will use the *adam* optimizer, which is the one people prefer to use by default.
- *loss*: The loss function to minimize. Since we're building a binary 0/1 classifier, the loss function to minimize is *binary_crossentropy*. We will see other examples of loss functions in later sections.
- *metrics*: Which metric to report statistics on, for classification problems we set this as *accuracy*.

```
history = model.fit(x=X, y=y, verbose=0, epochs=50)
```

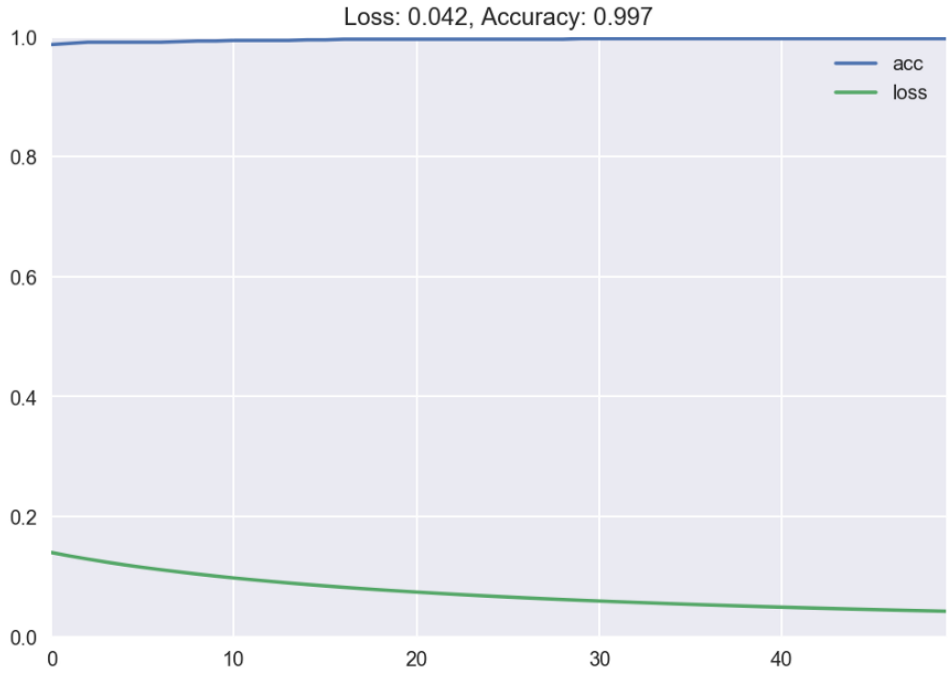
Now comes the fun part of actually training the model using the *fit* function. The arguments are as follows:

- *x*: The input data, we defined it as *X* above. It contains the x and y coordinates of the input points
- *y*: Not to be confused with the y coordinate of the input points. In all ML tutorials *y* refers to the labels, in our case the class we're trying to predict: 0 or 1.
- *verbose*: Prints out the loss and accuracy, set it to 1 to see the output.
- *epochs*: Number of times to go over the entire training data. When training models we pass through the training data not just once but multiple times.

```
plot_loss_accuracy(history)
```

The output of the fit method is the loss and accuracy at every epoch. We then plot it using our custom function, and see that the loss goes down to almost 0 over time, and the accuracy goes up to almost 1. Great! We have

successfully trained our first neural network model with Keras. I know this was a long explanation, but I wanted to explain what we're doing in detail the first time. Once you understand what's going on and practice a couple of times, all this becomes second nature.

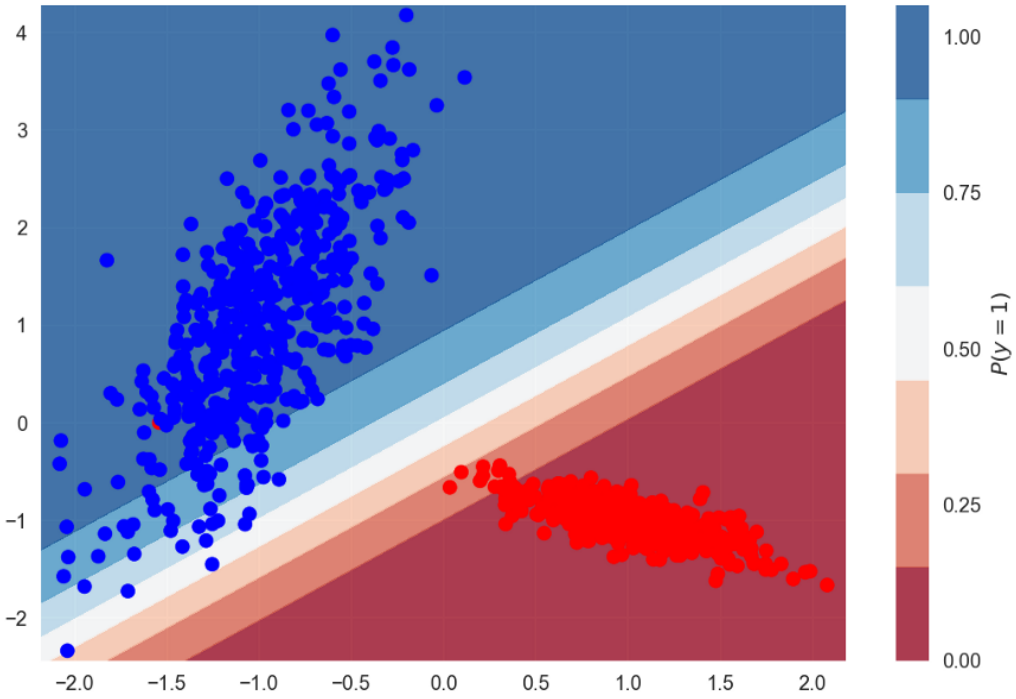


Below is a plot of the decision boundary. The various shades of blue and red represent the probability of a hypothetical point in that area belonging to class 1 or 0. The top left area is classified as class 1, with the color blue. The bottom right area is classified as class 0, colored as red. And there is a

transition around the decision boundary. This is a cool way to visualize the decision boundary the model is learning.

```
1  plot_decision_boundary(lambda x: model.predict(x), X, y)
```

linearly_separable_data_4.py hosted with ♡ by GitHub view raw



The *classification report* shows the precision and recall of our model. We get close to 100% accuracy. The value shown in the report should be 0.997 but it got rounded up to 1.0.

```
y_pred = model.predict_classes(X, verbose=0)
print(classification_report(y, y_pred))
```

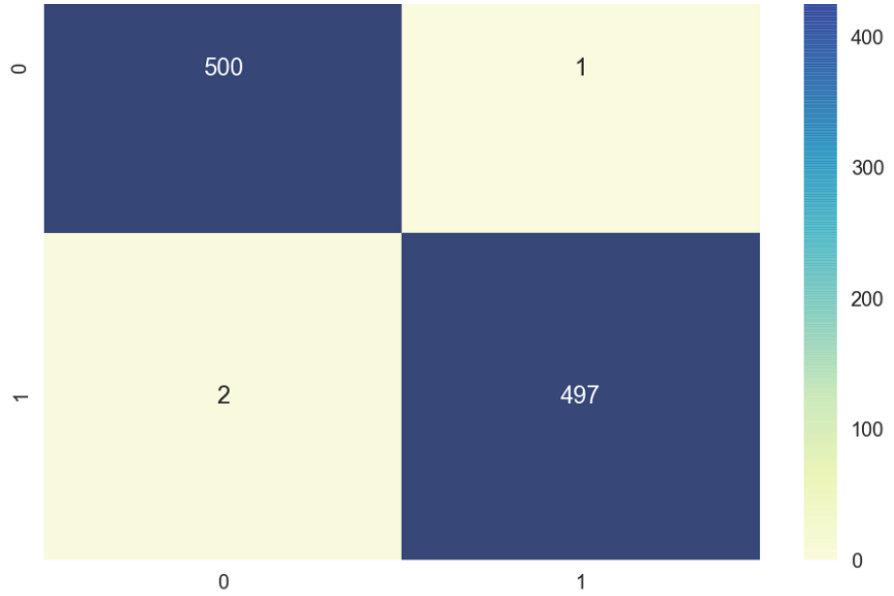
linearly_separable_data_5.py hosted with ♡ by GitHub view raw

	precision	recall	f1-score	support
0	1.00	1.00	1.00	501
1	1.00	1.00	1.00	499
avg / total	1.00	1.00	1.00	1000

The *confusion matrix* shows us how many classes were correctly classified vs misclassified. The numbers on the diagonal axis represent the number of correctly classified points, the rest are the misclassified ones. This particular matrix is not very interesting because the model only misclassifies 3 points. We can see one of the misclassified points at the top right part of the confusion matrix, the true value is class 0 but the predicted value is class 1.

```
plot_confusion_matrix(model, X, y)
```

linearly_separable_data_6.py hosted with ♡ by GitHub view raw

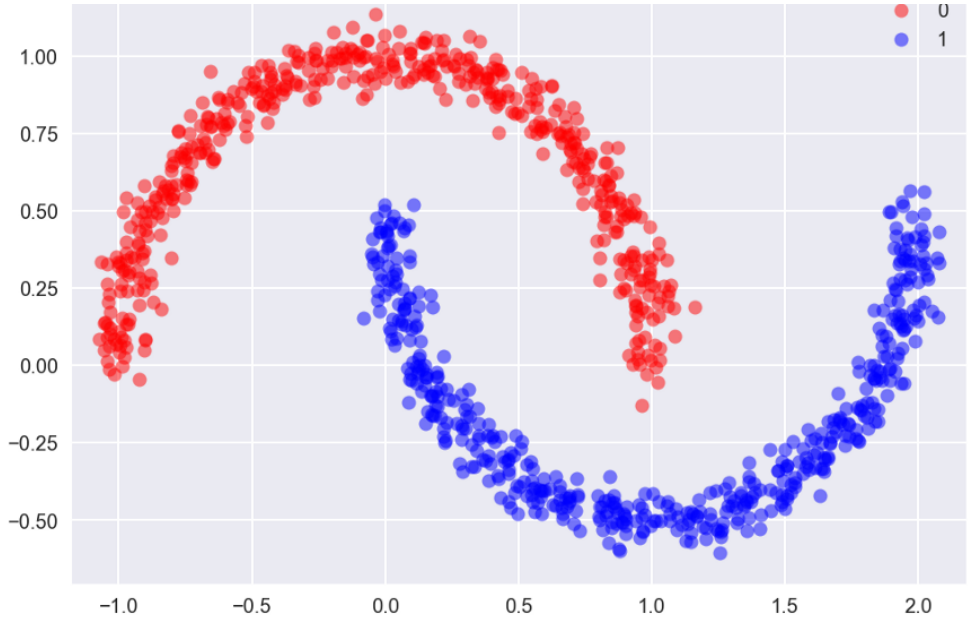


2.2) Complex Data - Moons

The previous dataset was linearly separable, so it was trivial for our logistic regression model to separate the classes. Here is a more complex dataset which isn't linearly separable. The simple logistic regression model won't be able to clearly distinguish between the classes. We're using the *make_moons* method of scikit-learn to generate the data.

```
1  X, y = make_moons(n_samples=1000, noise=0.05, random_state=0)
2  plot_data(X, y)
```

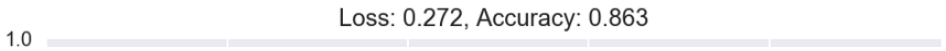
moons_1.py hosted with ♡ by GitHub view raw

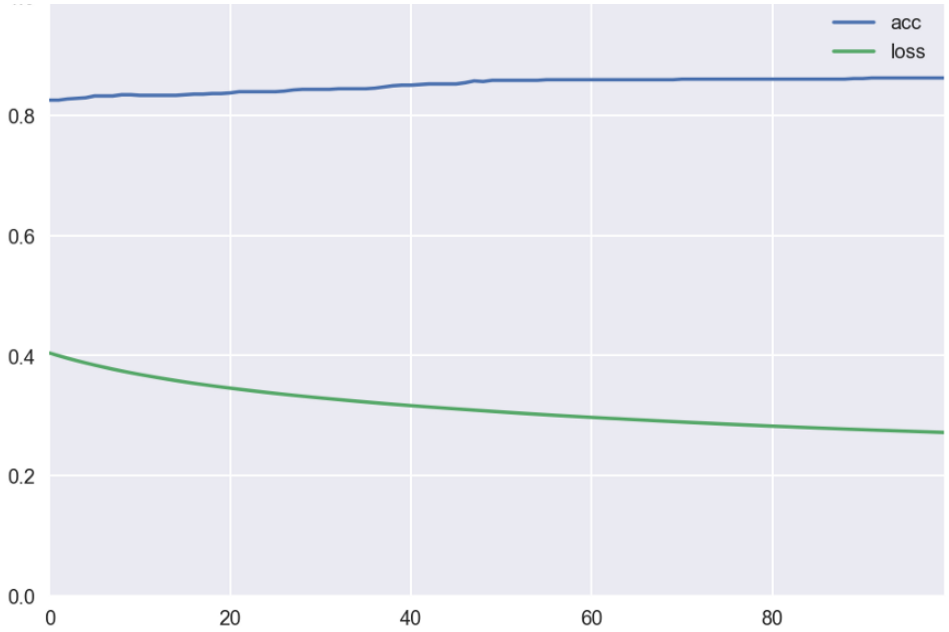


Let's build another logistic regression model with the same parameters as we did before. On this dataset we get 86% accuracy.

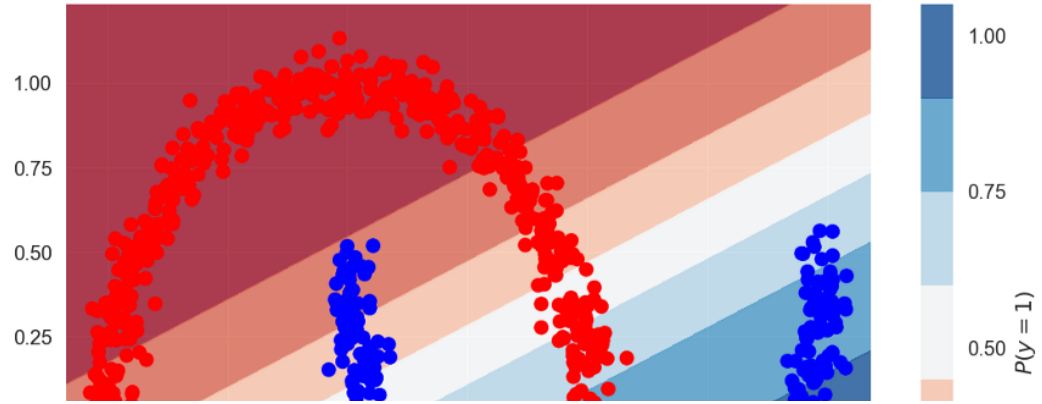
```
model = Sequential()
model.add(Dense(1, input_shape=(2,), activation='sigmoid'))

model.compile('adam', 'binary_crossentropy', metrics=['accuracy'])

history = model.fit(X, y, verbose=0, epochs=100)
plot_loss_accuracy(history)
```

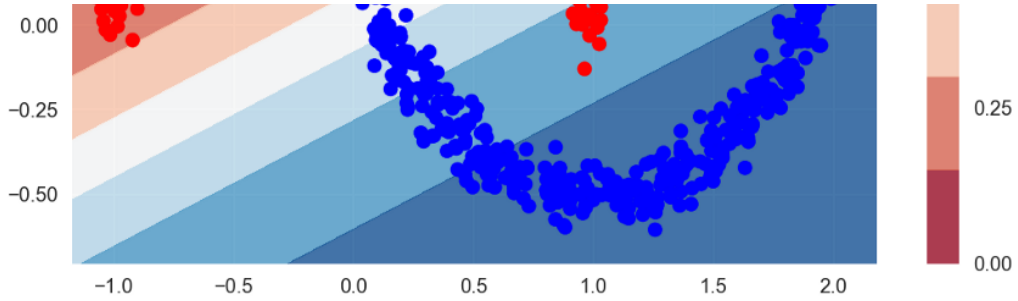
moons_2.py hosted with ♡ by GitHub view raw





The current decision boundary doesn't look as clean as the one before. The model tried to separate out the classes from the middle, but there are a lot of misclassified points. We need a more complex classifier with a non-linear decision boundary, and we will see an example of that soon.

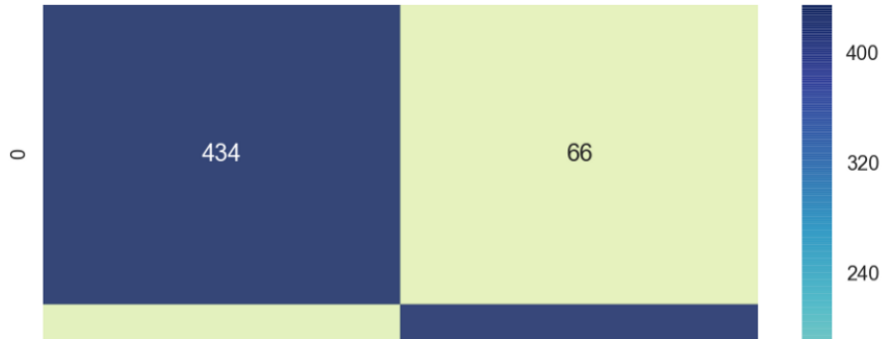


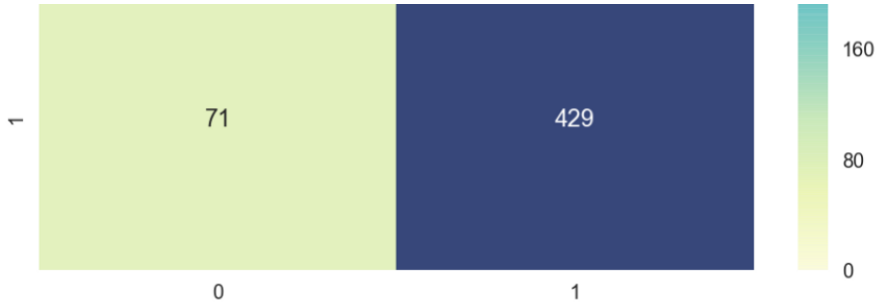


Precision of the model is 86%. It looks good on paper but we should easily be able to get 100% with a more complex model. You can imagine a curved decision boundary that will separate out the classes, and a complex model should be able to approximate that.

The classification report and the confusion matrix looks as follows.

	precision	recall	f1-score	support
0	0.86	0.87	0.86	500
1	0.87	0.86	0.86	500
avg / total	0.86	0.86	0.86	1000



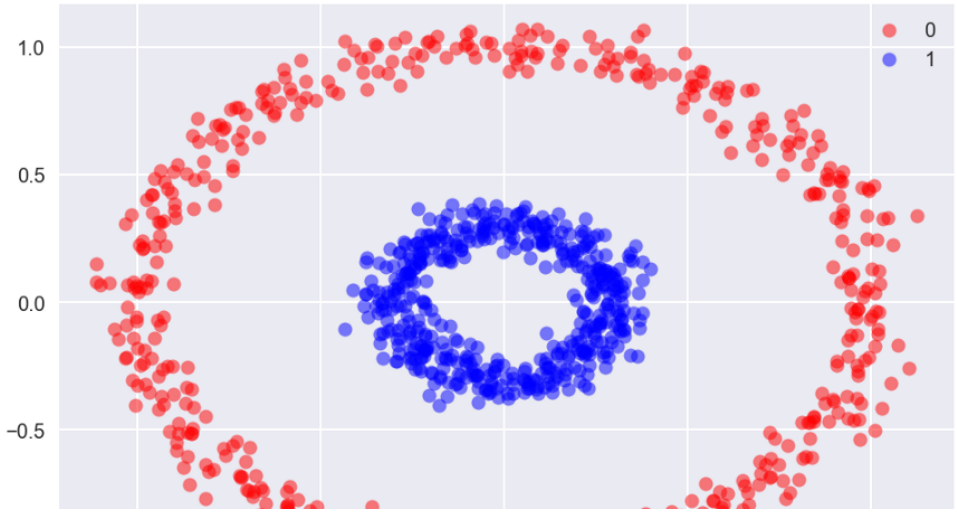


2.3 Complex Data - Circles

Let's look at one final example where the liner model will fail. This time using the *make_circles* function in scikit-learn.

```
1  X, y = make_circles(n_samples=1000, noise=0.05, factor=0.3, random_state=0)
2  plot_data(X, y)
```

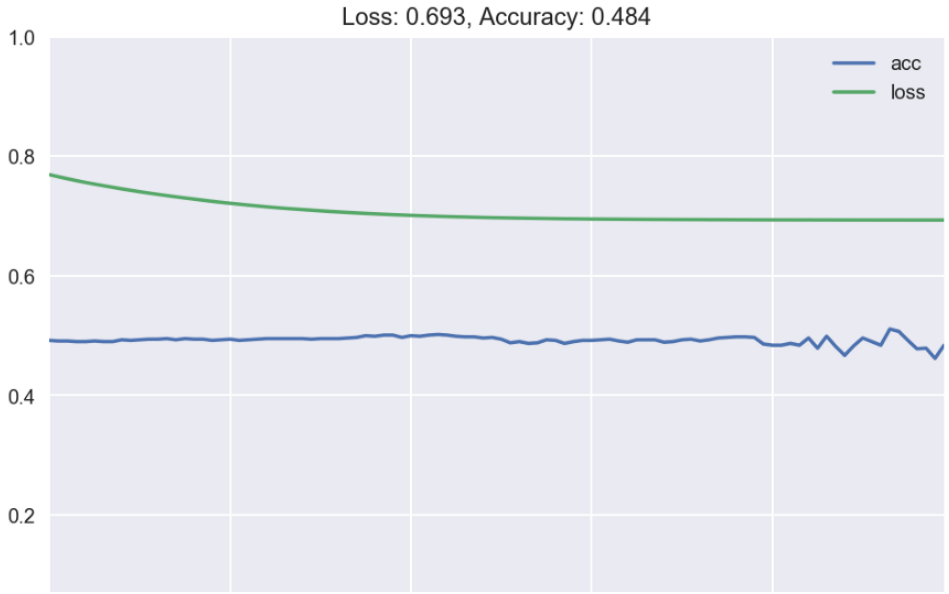
circles_1.py hosted with ♡ by GitHub view raw



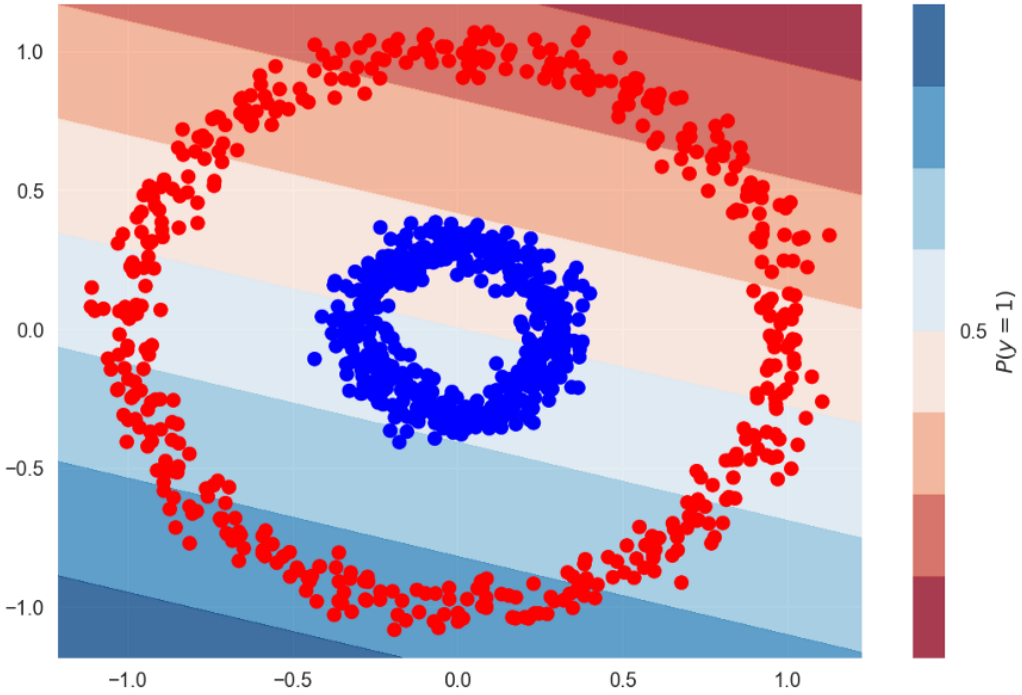
Building the model with same parameters.

```
model = Sequential()
model.add(Dense(1, input_shape=(2,), activation='sigmoid'))

model.compile('adam', 'binary_crossentropy', metrics=['accuracy'])

history = model.fit(X, y, verbose=0, epochs=100)
plot_loss_accuracy(history)
```

circles_2.py hosted with ♡ by GitHub view raw



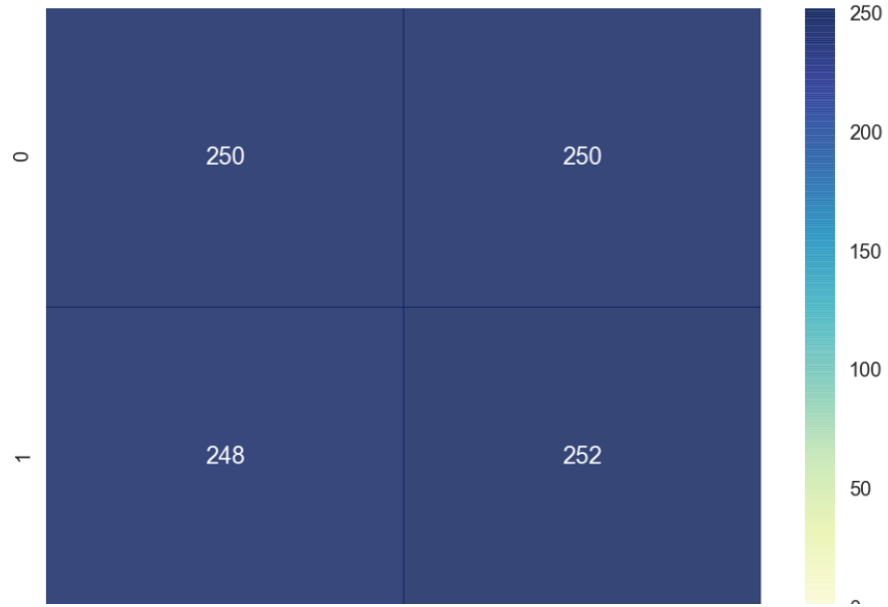
The decision boundary again passes from the middle of the data, but now we have much more misclassified points.



The accuracy is around 50%, shown below. No matter where the model draws the line, it will misclassify half of the points, due to the nature of the dataset.

The confusion matrix we see here is an example one belonging to a poor classifier. Ideally we prefer confusion matrices to look like the ones we saw above. High numbers along the diagonals meaning that the classifier was right, and low numbers everywhere else where the classifier was wrong. In our visualization, the color blue represents large numbers and yellow represents the smaller ones. So we would prefer to see blue on the diagonals and yellow everywhere else. Blue color everywhere is a bad sign meaning that our classifier is confused.

	precision	recall	f1-score	support
0	0.50	0.50	0.50	500
1	0.50	0.50	0.50	500
avg / total	0.50	0.50	0.50	1000



The most naive method which always predicts 1 no matter what the input is would get a 50% accuracy. Our model also got 50% accuracy, so it's not useful at all.

3. Artificial Neural Networks (ANN)

Now we will train a deep Artificial Neural Networks (ANN) to better classify the datasets which the logistic regression model struggled, Moons and Circles. We will also classify an even harder dataset of Sine Wave to demonstrate that ANN can form really complex decision boundaries.

3.1) Complex Data - Moons

While building Keras models for logistic regression above, we performed the following steps:

- Step 1: Define a Sequential model.
- Step 2: Add a Dense layer with sigmoid activation function. This was the only layer we needed.
- Step 3: Compile the model with an optimizer and loss function.

- Step 4: Fit the model to the dataset.
- Step 5: Analyze the results: plotting loss/accuracy curves, plotting the decision boundary, looking at the classification report, and understanding the confusion matrix.

While building a deep neural network, we only need to change step 2 such that, we will add several Dense layers one after another. The output of one layer becomes the input of the next. Keras again does most of the heavy lifting by initializing the weights and biases, and connecting the output of one layer to the input of the next. We only need to specify how many nodes we want in a given layer, and the activation function. It's as simple as that.

```
X, y = make_moons(n_samples=1000, noise=0.05, random_state=0)

model = Sequential()
model.add(Dense(4, input_shape=(2,), activation='tanh'))
model.add(Dense(2, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))

model.compile(Adam(lr=0.01), 'binary_crossentropy', metrics=['accuracy'])

history = model.fit(X, y, verbose=0, epochs=100)

plot_loss_accuracy(history)
```

moons_5.py hosted with ♡ by GitHub view raw

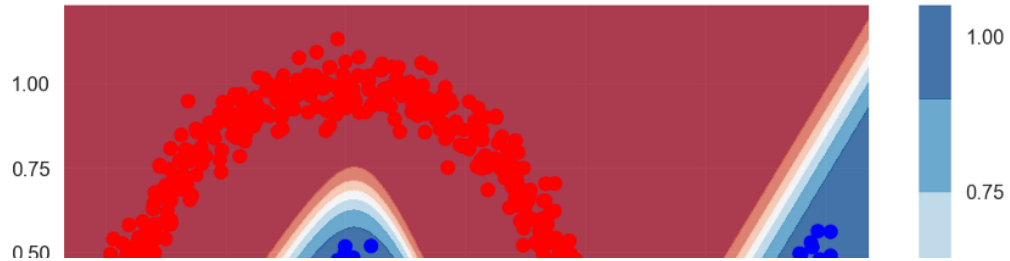
We first add a layer with 4 nodes and *tanh* activation function. Tanh is a commonly used activation function, and we'll learn more about it in another tutorial. We then add another layer with 2 nodes again using tanh activation. We finally add the last layer with 1 node and sigmoid activation. This is the final layer that we also used in the logistic regression model.

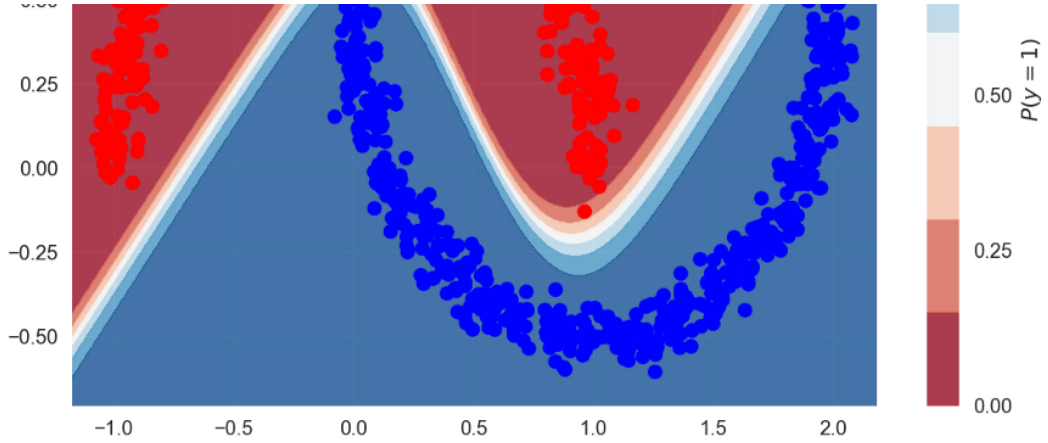
This is not a very deep ANN, it only has 3 layers: 2 hidden layers, and the output layer. But notice a couple of patterns:

- Output layer still uses the sigmoid activation function since we're working on a binary classification problem.
- Hidden layers use the tanh activation function. If we added more hidden layers, they would also use tanh activation. We have a couple of options for activation functions: sigmoid, tanh, relu, and variants of relu. In another article we'll explore the pros and cons of each one. We will also demonstrate why using sigmoid activation in hidden layers is a bad idea. For now it's safe to use tanh.
- We have fewer number of nodes in each subsequent layer. It's common to have less nodes as we stack layers on top of one another, sort of a triangular shape.

We didn't build a very deep ANN here because it wasn't necessary. We already achieve 100% accuracy with this configuration.

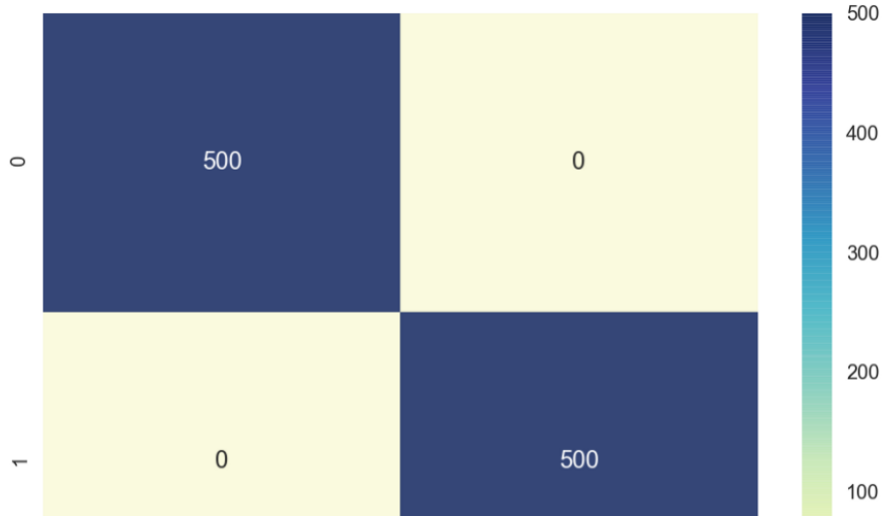
The ANN is able to come up with a perfect separator to distinguish the classes.





100% precision, nothing misclassified.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	500
1	1.00	1.00	1.00	500
avg / total	1.00	1.00	1.00	1000

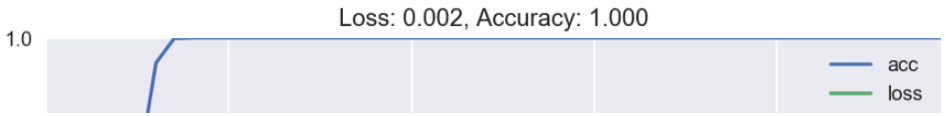


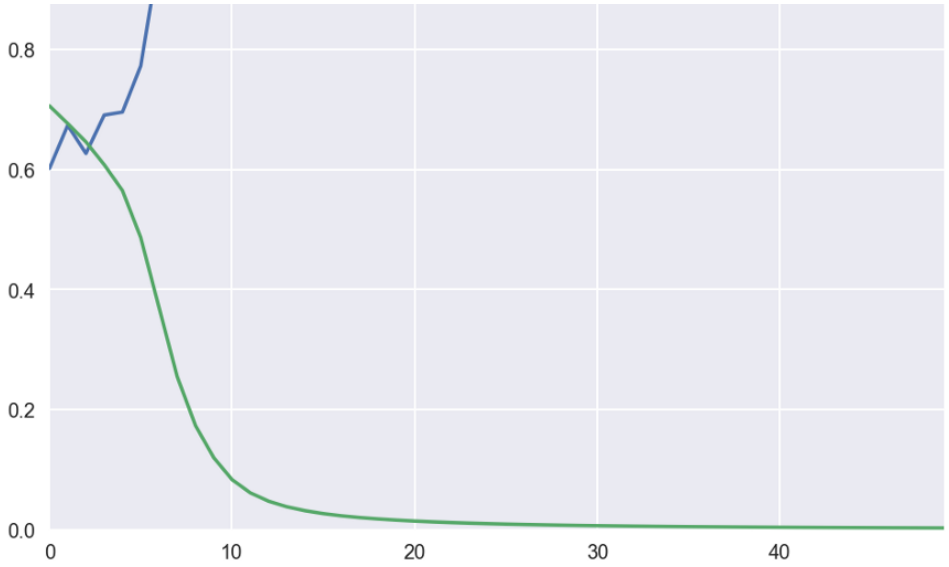
3.2) Complex Data - Circles

Now let's look at the Circles dataset, where the LR model achieved only 50% accuracy. The model is the same as above, we only change the input to the fit function using the current dataset. And we again achieve 100% accuracy.

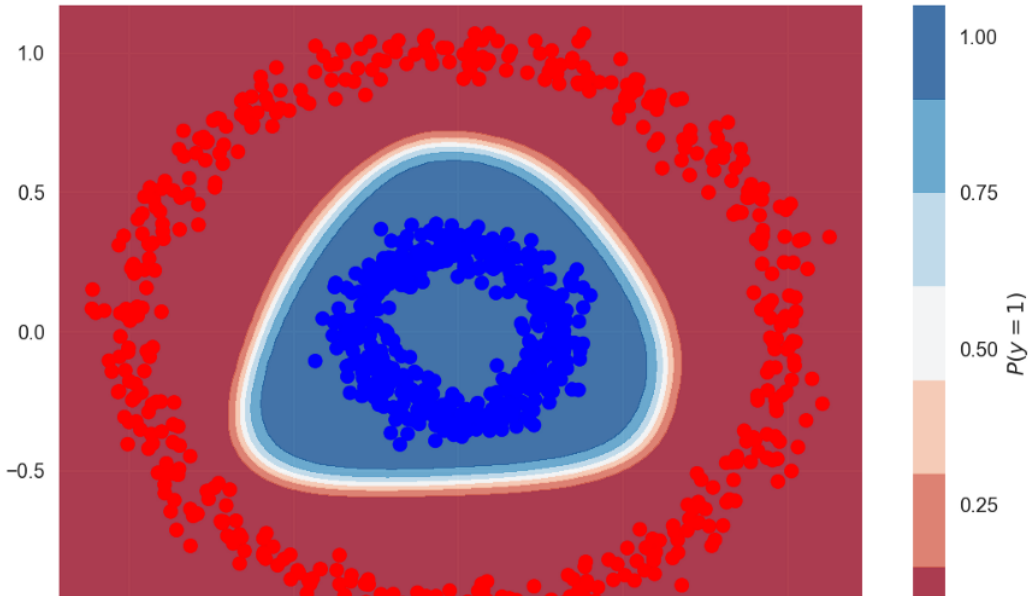
```
X, y = make_circles(n_samples=1000, noise=0.05, factor=0.3, random_state=0)

model = Sequential()
model.add(Dense(4, input_shape=(2,), activation='tanh'))
model.add(Dense(2, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))

model.compile(Adam(lr=0.01), 'binary_crossentropy', metrics=['accuracy'])

history = model.fit(X, y, verbose=0, epochs=50)

plot_loss_accuracy(history)
```

circles_5.py hosted with ♡ by GitHub view raw



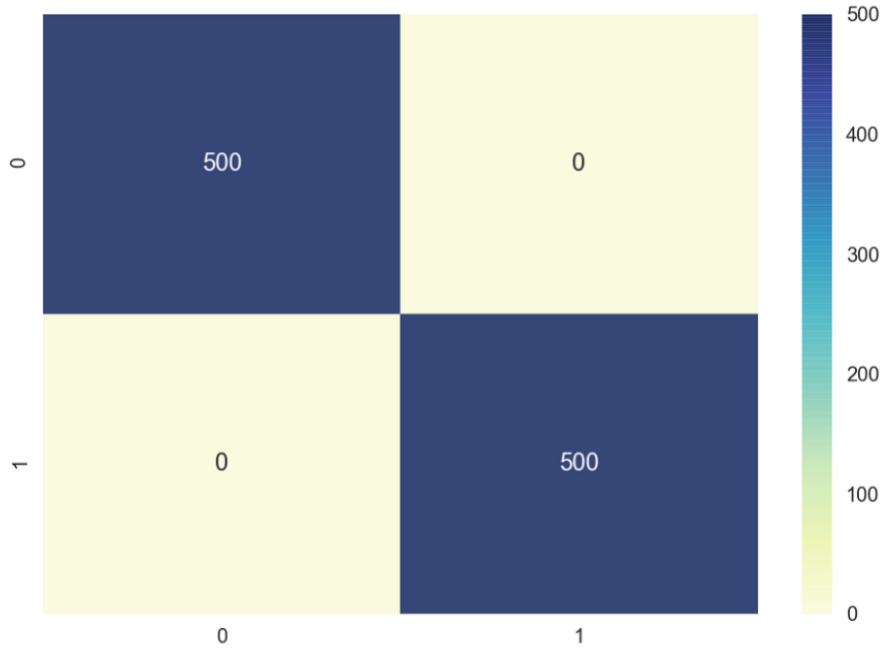


Similarly the decision boundary looks just like the one we would draw by hand ourselves. The ANN was able to figure out an optimal separator.



Just like above we get 100% accuracy.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	500
1	1.00	1.00	1.00	500
avg / total	1.00	1.00	1.00	1000



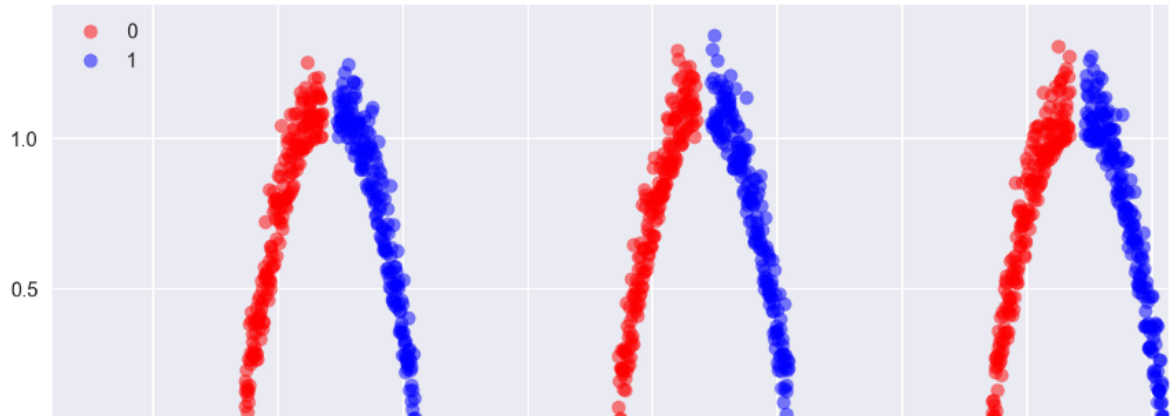
4.3) Complex Data - Sine Wave

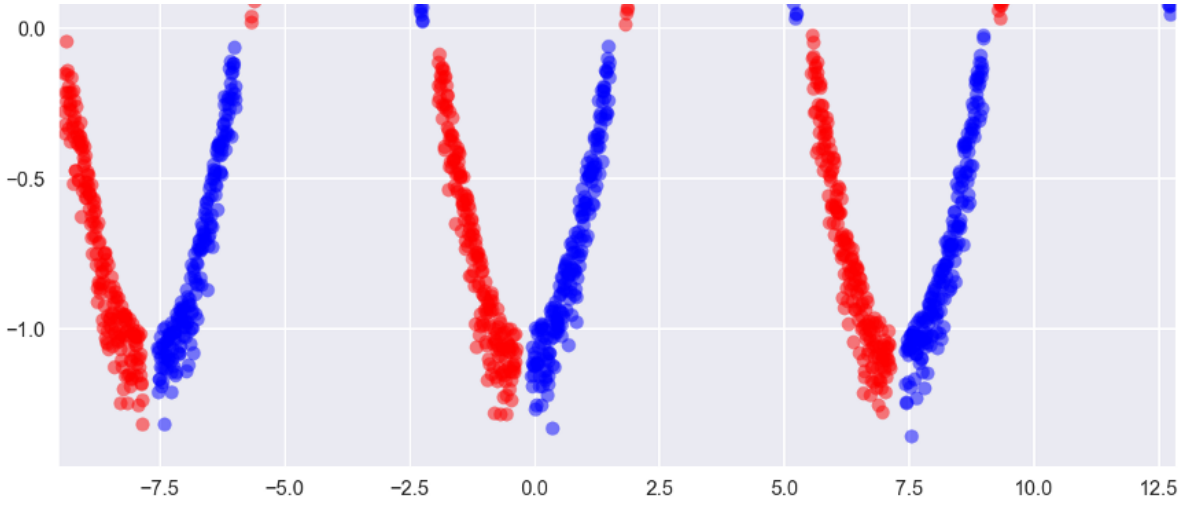
Let's try to classify one final toy dataset. In the previous sections, the classes were separable by one continuous decision boundary. The boundary had a complex shape, it wasn't linear, but still one continuous decision boundary was enough. ANN can draw arbitrary number of complex decision boundaries, and we will demonstrate that.

Let's create a sinusoidal dataset looking like the sine function, every up and down belonging to an alternating class. As we can see in the figure, a single decision boundary won't be able to separate out the classes. We will need a series of non-linear separators.

```
X, y = make_sine_wave()

plot_data(X, y, figsize=(10, 8))
```

sine_1.py hosted with ♡ by GitHub view raw

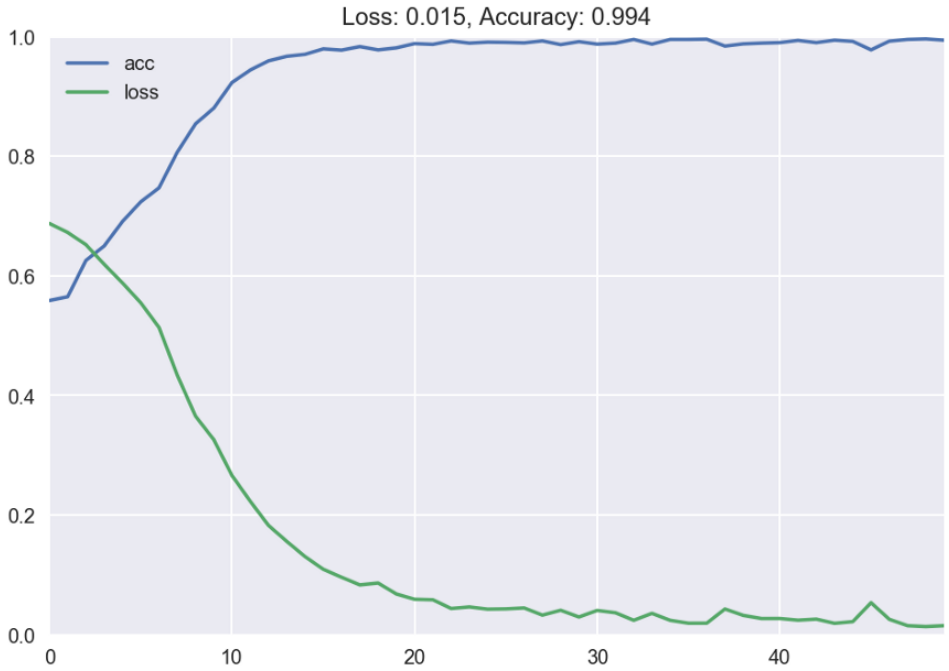




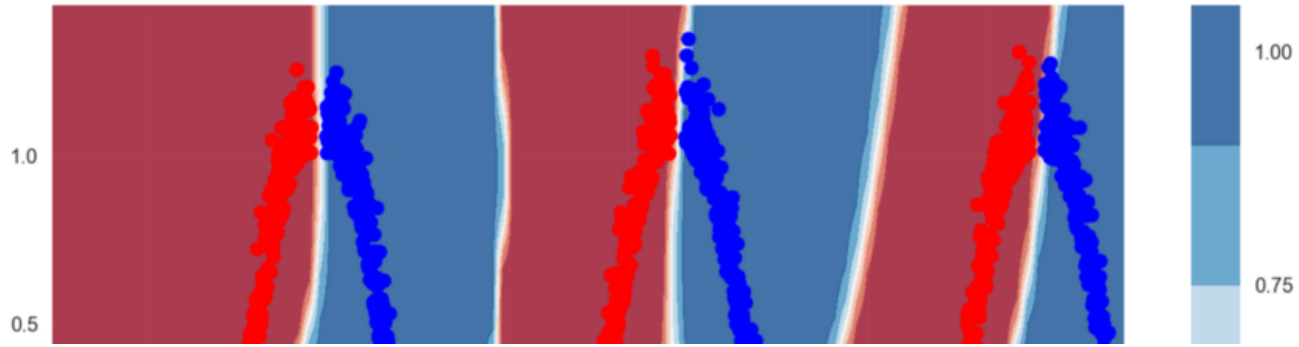
Now we need a more complex model for accurate classification. So we have 3 hidden layers, and an output layer. The number of nodes per layer has also increased to improve the learning capacity of the model. Choosing the right number of hidden layers and nodes per layer is more of an art than science, usually decided by trial and error.

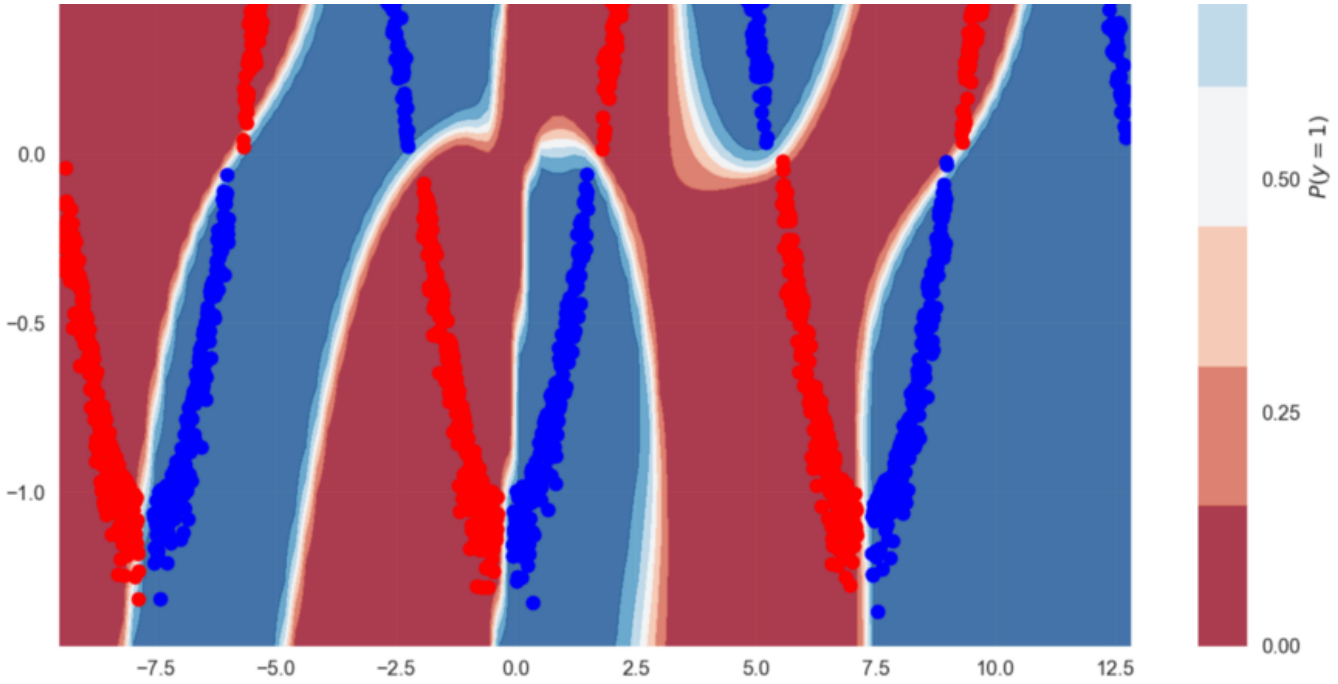
```
model = Sequential()
model.add(Dense(64, input_shape=(2,), activation='tanh'))
model.add(Dense(64, activation='tanh'))
model.add(Dense(64, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))

model.compile('adam', 'binary_crossentropy', metrics=['accuracy'])

history = model.fit(X, y, verbose=0, epochs=50)

plot_loss_accuracy(history)
```

sine_2.py hosted with ♡ by GitHub view raw



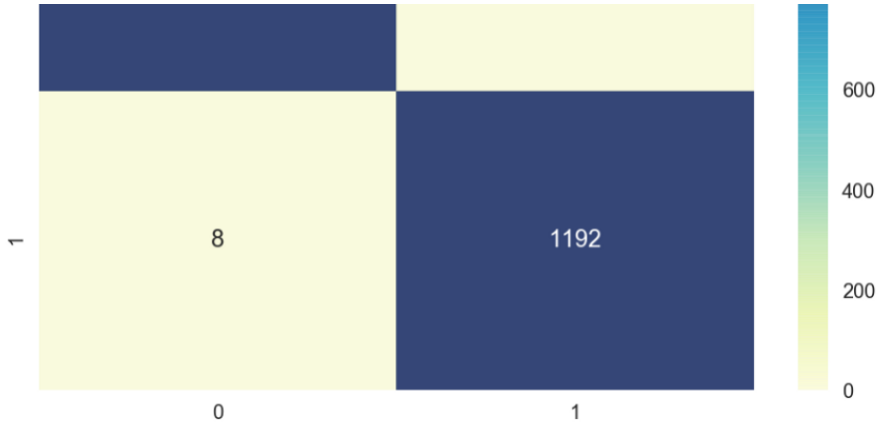
The ANN was able to model a pretty complex set of decision boundaries.





Precision is 99%, we only have 14 misclassified points out of 2400. Pretty good.

	precision	recall	f1-score	support
0	0.99	0.99	0.99	1200
1	0.99	0.99	0.99	1200
avg / total	0.99	0.99	0.99	2400



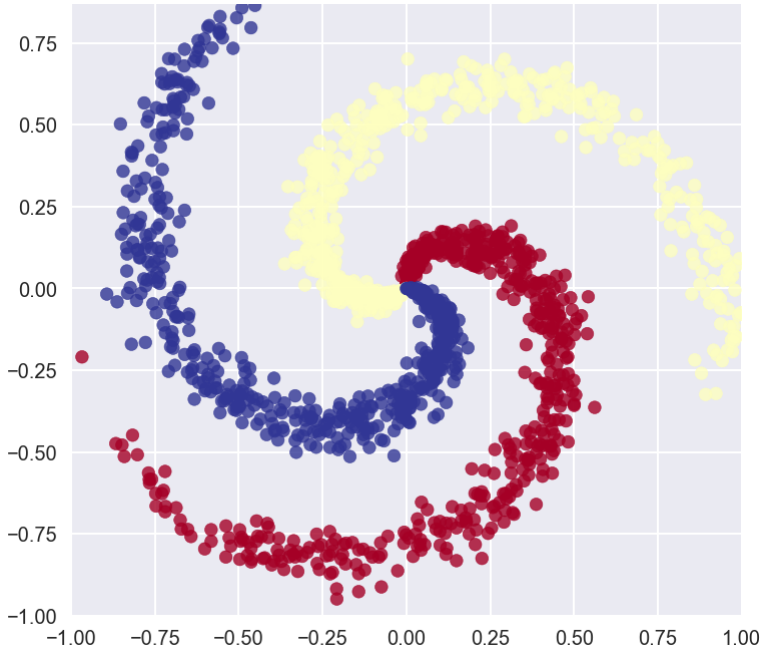
4. Multiclass Classification

In the previous sections we worked on binary classification. Now we will take a look at a multi-class classification problem, where the number of classes is more than 2. We will pick 3 classes for demonstration, but our approach generalizes to any number of classes.

Here's how our dataset looks like, spiral data with 3 classes, using the *make_multiclass* method in scikit-learn.

```
1  X, y = make_multiclass(K=3)
```

multiclass_1.py hosted with ♡ by GitHub view raw



4.1) Softmax Regression

As we saw above, Logistic Regression (LR) is a classification method for 2 classes. It works with binary labels 0/1. Softmax Regression (SR) is a generalization of LR where we can have more than 2 classes. In our current dataset we have 3 classes, represented as 0/1/2.

Building the model for SR is very similar to LR, for reference here's how we built our Logistic Regression model.

```
model = Sequential()
model.add(Dense(1, input_shape=(2,), activation='sigmoid'))

```

```
model.compile('adam', 'binary_crossentropy', metrics=['accuracy'])

history = model.fit(X, y, verbose=0, epochs=20)
```

multiclass_2.py hosted with ♡ by GitHub view raw

And here's how how we will build the Softmax Regression model.

```
model = Sequential()
model.add(Dense(3, input_shape=(2,), activation='softmax'))

model.compile('adam', 'categorical_crossentropy', metrics=['accuracy'])

y_cat = to_categorical(y)
history = model.fit(X, y_cat, verbose=0, epochs=20)

plot_loss_accuracy(history)
```

multiclass_3.py hosted with ♡ by GitHub view raw

There are a couple of differences, let's go over them one by one:

- Number of nodes in the dense layer: LR uses 1 node, where SR has 3 nodes. Since we have 3 classes it makes sense for SR to be using 3 nodes. Then the question is, why does LR uses only 1 node, it has 2 classes so it appears like we should have used 2 nodes instead. The answer is, because we can achieve the same result with using only 1 node. As we

saw above, LR models the probability of an example belonging to class one: $P(class=1)$. And we can calculate class 0 probability by: $1-P(class=1)$. But when we have more than 2 classes, we need individual nodes for each class. Because knowing the probability of one class doesn't let us infer the probability of the other classes.

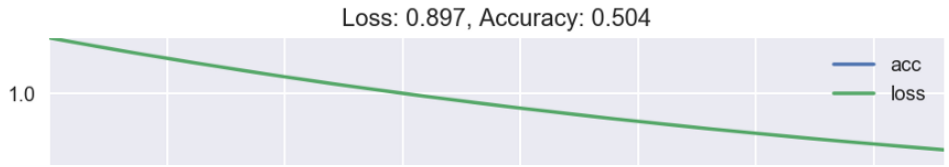
- Activation function: LR used sigmoid activation function, SR uses *softmax*. Softmax scales the values of the output nodes such that they represent probabilities and sum up to 1. So in our case $P(class=0)+P(class=1)+P(class=2)=1$. It doesn't do it in a naive way by dividing individual probabilities by the sum though, it uses the exponential function. So higher values get emphasized more and lower values get squashed more. We will talk in detail what softmax does in another tutorial. For now you can simply think of it as a normalization function which lets us interpret the output values as probabilities.
- Loss function: In a binary classification problem like LR, the loss function is binary_crossentropy. In the multiclass case, the loss function is categorical_crossentropy. Categorical crossentropy is the generalization of binary crossentropy to more than 2 classes. Going into the theory behind loss functions is beyond the scope of this tutorial. But for now only knowing this property is enough.

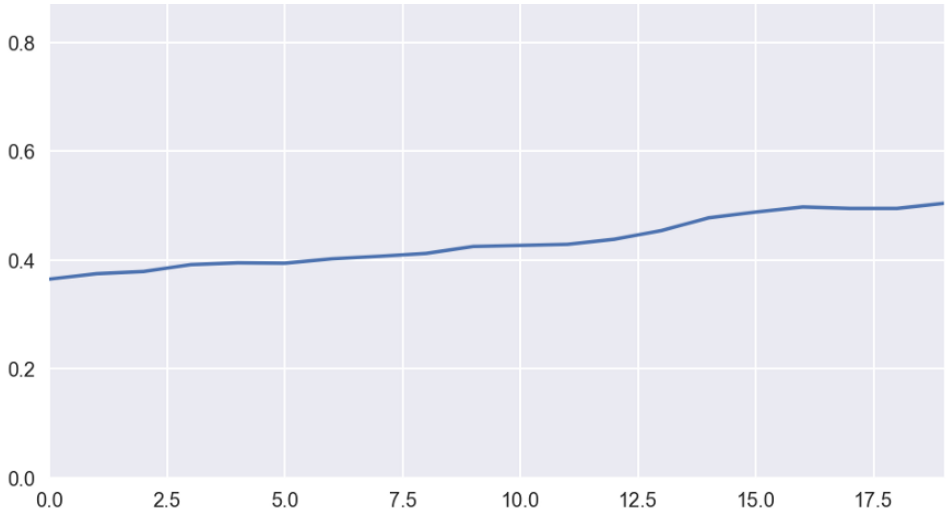
- Fit function: LR used the vector y directly in the fit function, which has just one column with 0/1 values. When we're doing SR the labels need to be in *one-hot* representation. In our case y_cat is a matrix with 3 columns, where all the values are 0 except for the one that represents our class, and that is 1.

It took some time to talk about all the differences between LR and SR, and it looks like there's a lot to digest. But again after some practice this will become a habit, and you won't even need to think about any of this.

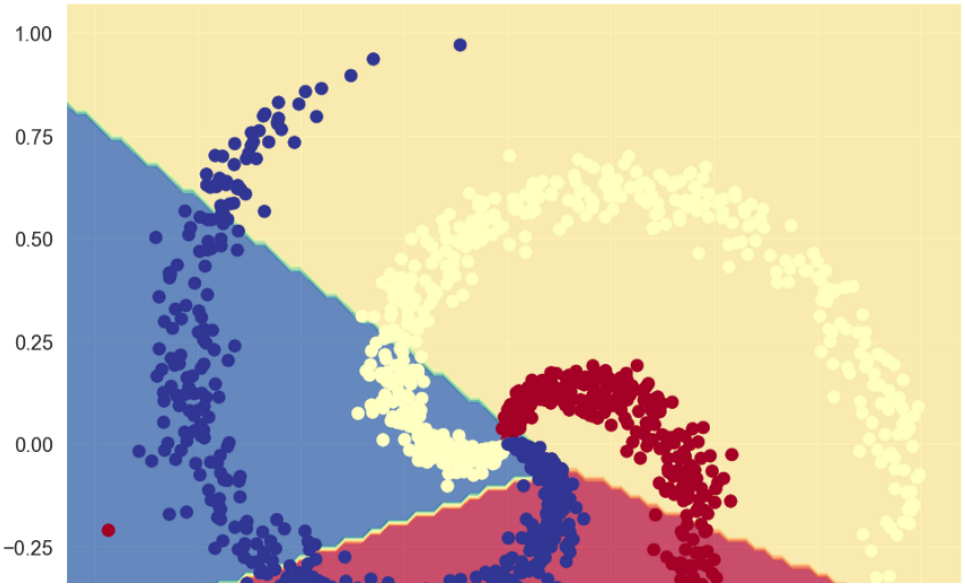
After all this theory let's take a step back and remember that LR is a linear classifier. SR is also a linear classifier, but for multiple classes. So the "power" of the model hasn't changed, it's still a linear model. We just generalized LR to apply it to a multiclass problem.

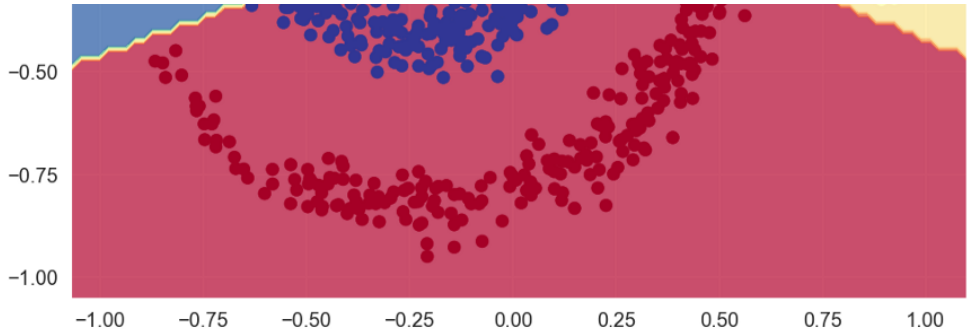
Training the model gives us an accuracy of around 50%. The most naive method which always predicts class 1 no matter what the input is would have an accuracy of 33%. The SR model is not much of an improvement over it. Which is expected because the dataset is not linearly separable.





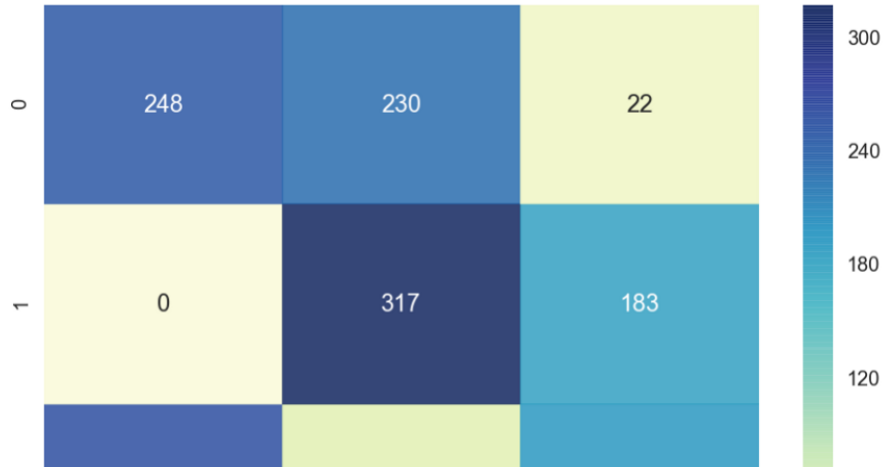
Looking at the decision boundary confirms that we still have a linear classifier. The lines look jagged due to floating point rounding but in reality they're straight.

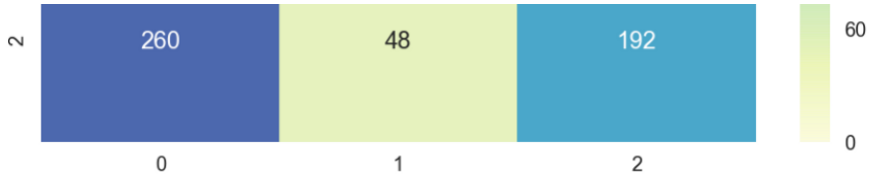




Here's the precision and recall corresponding to the 3 classes. And the confusion matrix is all over the place. Clearly this is not an optimal classifier.

	precision	recall	f1-score	support
0.0	0.49	0.50	0.49	500
1.0	0.53	0.63	0.58	500
2.0	0.48	0.38	0.43	500
avg / total	0.50	0.50	0.50	1500





4.2) Deep ANN

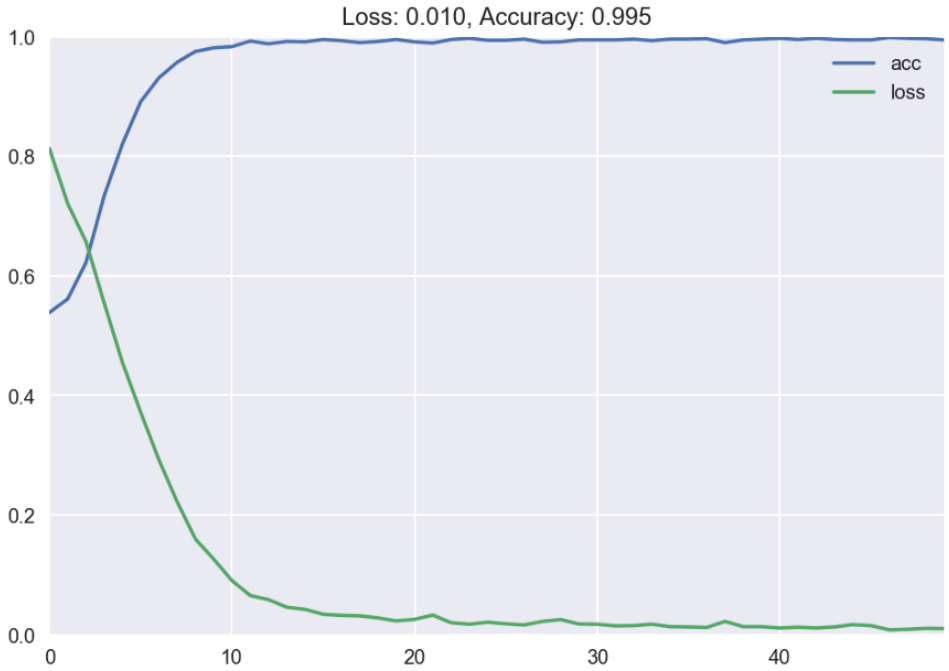
Now let's build a deep ANN for multiclass classification. Remember that the changes going from LR to ANN was minimal. We only needed to add more Dense layers. We will do the same again. Adding a couple of Dense layers with tanh activation function.

```
model = Sequential()
model.add(Dense(64, input_shape=(2,), activation='tanh'))
model.add(Dense(32, activation='tanh'))
model.add(Dense(16, activation='tanh'))
model.add(Dense(3, activation='softmax'))

model.compile('adam', 'categorical_crossentropy', metrics=['accuracy'])

y_cat = to_categorical(y)
history = model.fit(X, y_cat, verbose=0, epochs=50)

plot_loss_accuracy(history)
```

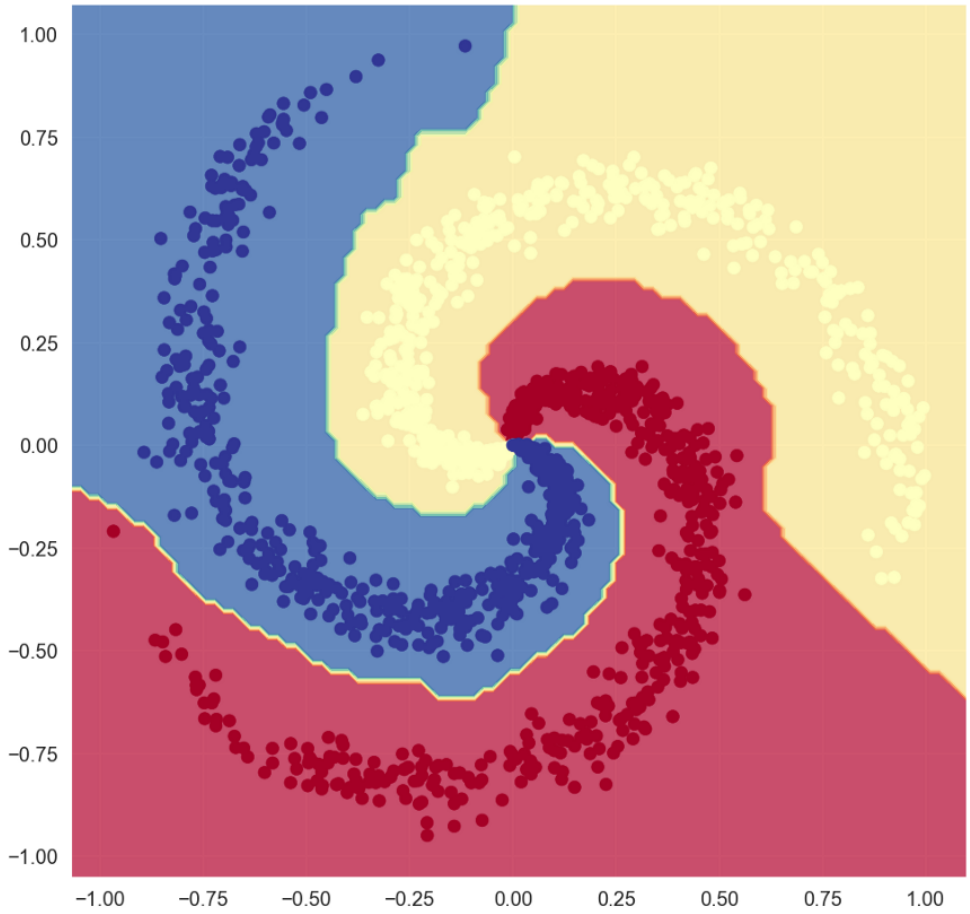
multiclass_4.py hosted with ♡ by GitHub view raw

Note that the output layer still has 3 nodes, and uses the softmax activation. The loss function also didn't change, still categorical_crossentropy. These won't change going from a linear model to a deep ANN, since the problem definition hasn't changed. We're still working on multiclass classification. But now using a more powerful model, and that power comes from adding more layers to our neural net.

We achieve 99% accuracy in just a couple of epochs.

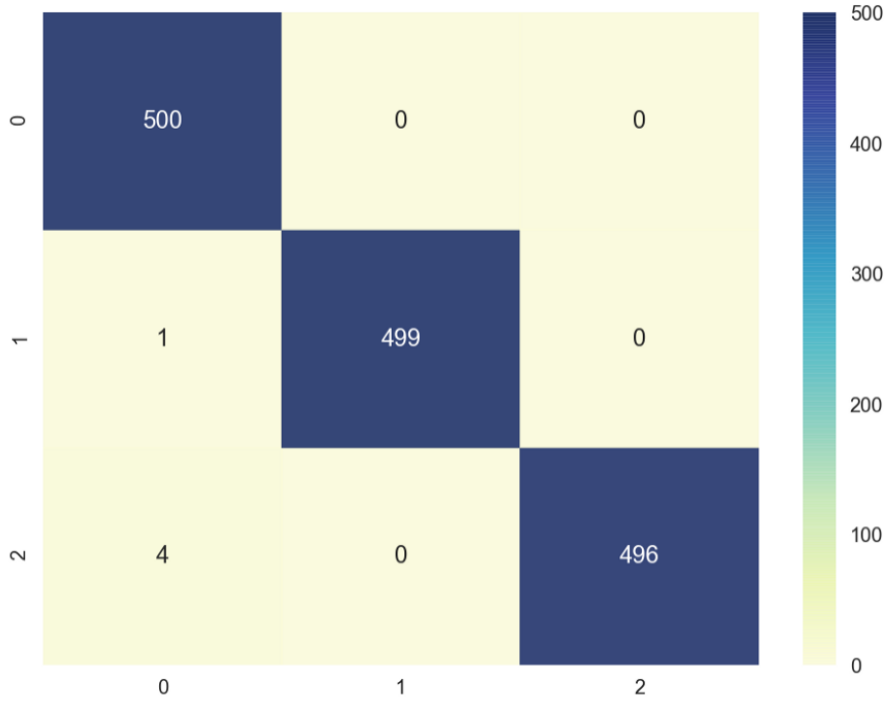


The decision boundary is non-linear.



We got almost 100% accuracy. We totally misclassified 5 points out of 1500.

	precision	recall	f1-score	support
0.0	0.99	1.00	1.00	500
1.0	1.00	1.00	1.00	500
2.0	1.00	0.99	1.00	500
avg / total	1.00	1.00	1.00	1500



5) Conclusion

Thanks for spending time reading this article, I know this was a rather lengthy tutorial on Artificial Neural Networks and Keras. I wanted to be as detailed as possible, while still keeping the article length manageable. I hope you enjoyed it.

There was a common theme in this article such that we first introduced the task, we then approached it using a simple method and observed the limitations. Afterwards we used a more complex deep model to improve on

it and got much better results. I think the ordering is important. No complex method becomes successful unless it has evolved from a simpler model.

The entire code for this article is available here if you want to hack on it yourself. If you have any feedback feel free to reach out to me on twitter.

Machine Learning Deep Learning Neural Networks Data Science Towards Data Science