Introductory Notes on Neural Networks

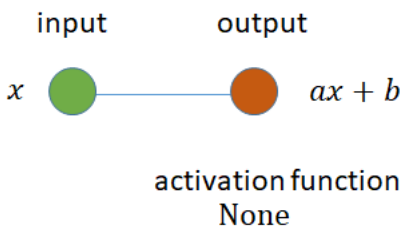
Contents

1. Regression Models

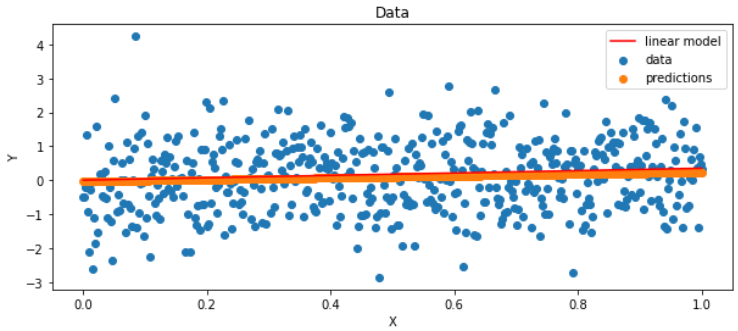
1.1. Linear Regression

Consider a series of input datapoints $\{x_i\}$, where each datapoint has a corresponding label $y(x_i)$. Let's assume the labels are given by a $y(x) = ax + b$, and fit a linear line.

We can do so in terms of a neural network using a single input node (taking in the datapoint x_i), followed by one layer with a single node with weight a and bias b (so that the resulting node has the value $ax + b$. We do not use an activation function, so that the output of the network will simply be $ax + b$.




```
model = Sequential()
model.add(Dense(1, input_shape=(1,), activation=None))
model.compile(loss='mse', optimizer='Adam')
```



Datapoints (blue) generated according to $ax + b + \epsilon$, where the error ϵ is normally distributed. The above plot shows the analytical solution to this problem (red) together with the fit by the 'neural network' (orange).

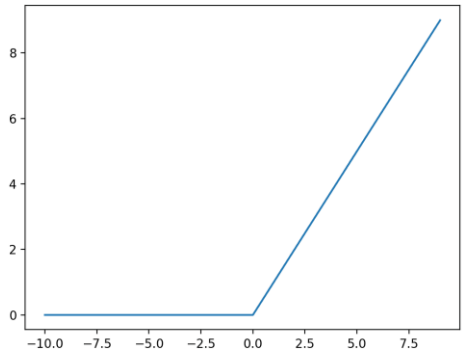
1.2. Nonlinear Regression

Next we will study non-linear models of the form $y(x) = f(x)$, where $f(x)$ is a nonlinear function of the input data.

Instead of using the input data x as a basis for our problem, a nonlinear function $f(x)$ in the input data (like polynomials) might be better suited as a basis for our regression problem. We do not have to choose our basis functions f ahead of time, but instead we can fit a parametrisation of f directly from our dataset.

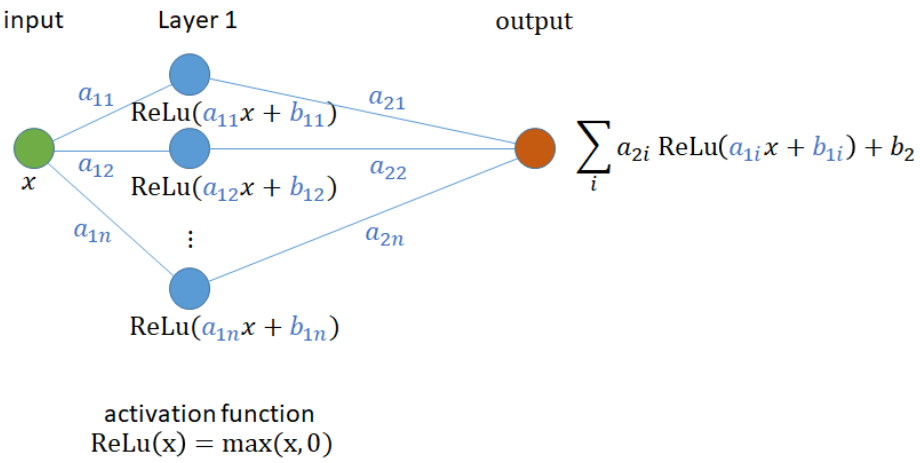
To fit a nonlinear function on our input data we will add intermediate layers with element wise activation functions to our network. In this example we will work with a rectified linear (ReLu) activation function and use it to fit model of the form $y(x) = x^2$.

The ReLu activation function $\mathrm{ReLu}(x) = \max(x, 0)$ is a piecewise linear function that sets all negative values equal to zero.

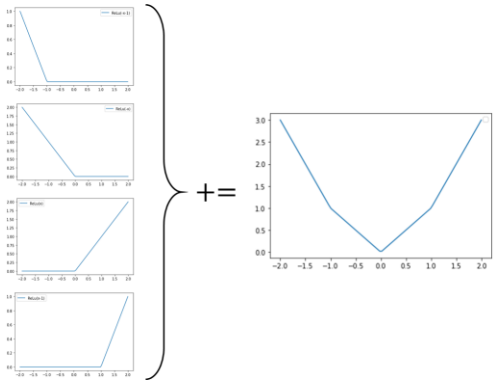


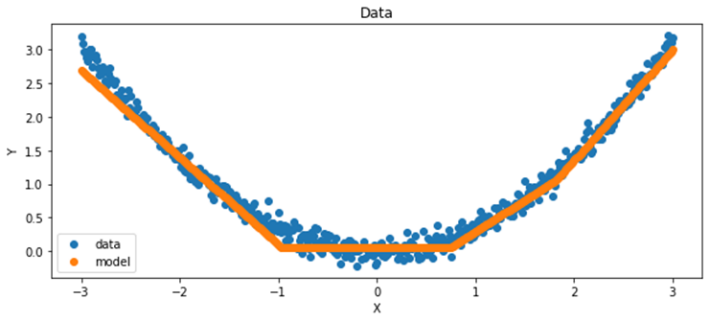
By adding an intermediate layer of n notes in between the input node and output node, each of the nodes in the intermediate layer will transform the input according to $x \to \mathrm{ReLu}(ax + b)$.

```
model = Sequential()
intermediate_nodes = 3
model.add(Dense(intermediate_nodes, activation="relu", input_shape=(1,)))
model.add(Dense(intermediate _nodes, activation="relu", input_shape=(intermediate_nodes,)))
model.add(Dense(1, activation=None, input_shape=(intermediate_nodes, )))
```



The final layer (with no activation function) then obtains a weighted sum of the different ReLu functions $\mathrm{ReLu}(a_i x + b_i)$ that have been fitted to our dataset in the intermediate layer. For every connection in our network we will have one weight parameter a, for every node in our network we can choose to have an additional bias parameter b. These parameters specifying how the ReLu functions have to be fitted to our dataset will be optimized when we 'train' the network on our data (typically this is done using some gradient descent algorithm that tries to minimise a specified error).





Datapoints (blue) generated according to $ax^2 + \epsilon$, where the error ϵ is normally distributed. The above plot shows the fit by a neural network (orange) that contains 3 nodes in the intermediate layer. Note the fit of the model to our data consists of the sum of a flat part and 3 independently fitted ReLu fnctions (one from each of the nodes in the intermediate layer). The total number of parameters in this network is 10 (6 weights for the 2x3 connections between the nodes, and 1 bias parameter from each of the 4 nodes).

1.2.1 More Layers and More Accuracy

By adding an additional layer to our neural network, each next layer will be able to combine the approximations fitted by the previous layers into an even more accurate fit. As an example we show here a fit to the same dataset as in the previous section, now using a network containing two layers each containing 3 nodes with ReLu activation functions.

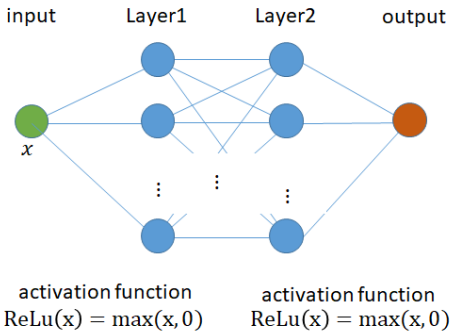
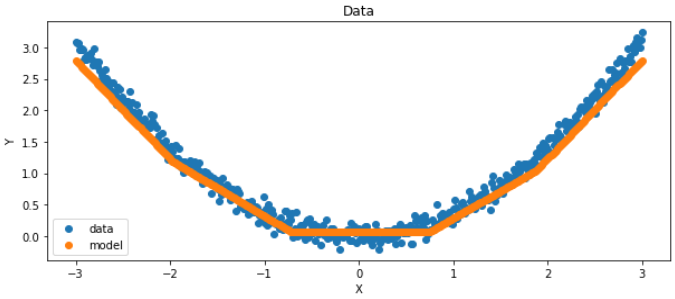


Diagram of a neural network containing two intermediate dense layers with ReLu activation functions.



Datapoints (blue) generated according to $ax^2 + \epsilon$, where the error ϵ is normally distributed. The above plot shows a fit (orange) by the neural network shown on the left.

1.2.2 Different Activation Functions

The downside of using ReLu activation functions is that the fits to our data will become clunky linear interpolations. The advantage of using ReLu functions is that $\mathrm{ReLu}(x) = \max(x, 0)$ is very easy to calculate on a computer, making both network and training very fast.

There are various activation functions one can choose from that might be better or less suited to the problem one is considering.

As an example we show here a fit to the same dataset as in the previous section, now using a network containing two layers containing 3 nods; the first with a sigmoidal activation function and the second with a ReLu activation function. Note that the smoothness of the sigmoidal activation function helps us to achieve a smoother fit to the data.

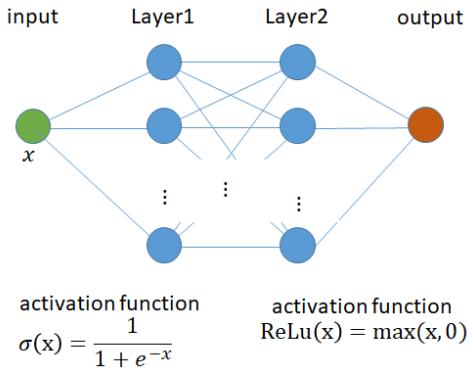
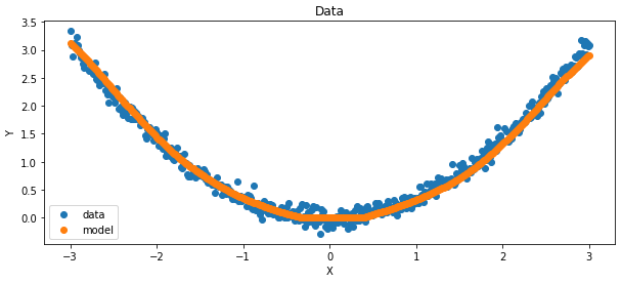


Diagram of a neural network containing two intermediate dense layers, one with a sigmoid activation function and the second with a ReLu activation function.



Datapoints (blue) generated according to $ax^2 + \epsilon$, where the error ϵ is normally distributed. The above plot shows a fit (orange) by the neural network shown on the left.

2. Multivariate Regression

2.2 Multivariate Linear Regression

So far we have considered examples acting on one-dimensional datasets. Everything we have been discussing so far can realtively easily be generalised to higher dimensional datasets.

```
input_dim = 2
model = Sequential()
model.add(Dense(1, activation=None, input_shape=(input_dim, )))
```

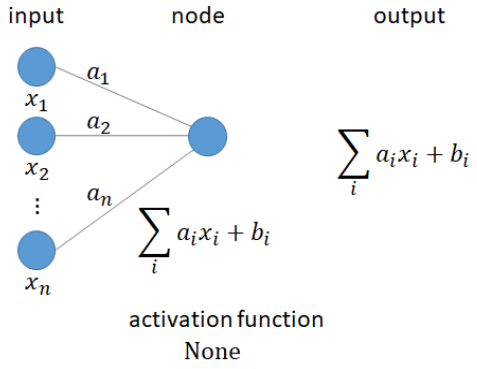
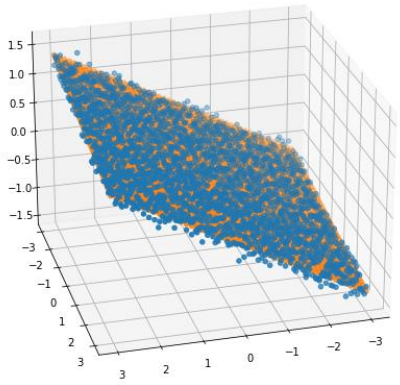


Diagram of a neural network containing multiple input nodes and a single output node, performing a multivariate version of the linear regression model we discussed earlier.



Datapoints (blue) generated according to $\alpha_1 x_1 + \alpha_2 x_2 + \epsilon$, where the error ϵ is normally distributed. The above plot shows a fit to the data (orange) by the neural network shown on the left.

2.2 Multivariate Nonlinear Regression

```
input_dim = 2
model = Sequential()
nodes = 3
model.add(Dense(nodes, activation="sigmoid", input_shape=(input_dim,)))
model.add(Dense(nodes, activation="relu", input_shape=(nodes,)))
model.add(Dense(1, activation=None, input_shape=(nodes, )))
```

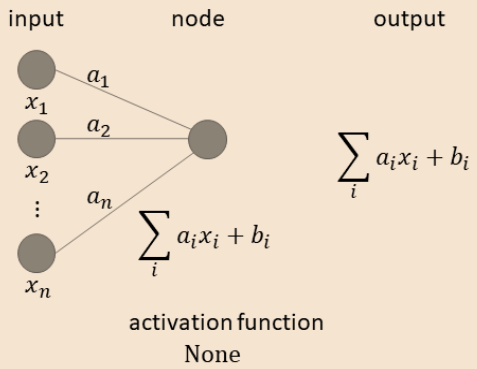
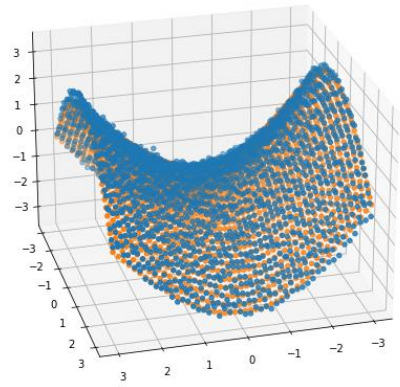


Diagram of a neural network containing multiple input nodes and a single output node, performing a multivariate version of the linear regression model we discussed earlier.



Datapoints (blue) generated according to $\alpha_1 x_1 + \alpha_2 x_2 + \epsilon$, where the error ϵ is normally distributed. The above plot shows a fit to the data (orange) by the neural network shown on the left.

3. Logistic Regression

3.1 Binary Logistic Regression

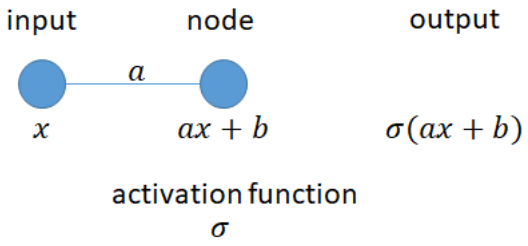
In this example we have a dataset $\{x_i\}$ where every datapoint is labelled $y(x_i)$ as either 0 or 1, and there is a correlation between $\{x\}$ and the probability that $y(x) = 1$.

To fit the probability function we use a logistic (or sigmoid) function

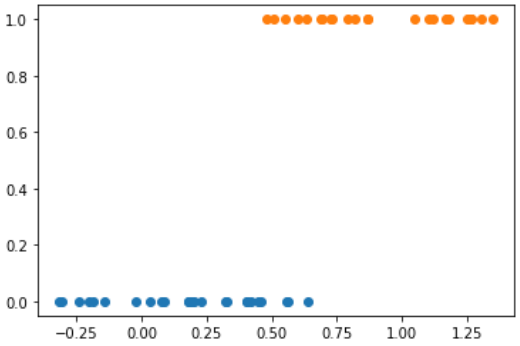
$$p(x) = \frac{1}{1 - e^{-(ax+b)}}$$

In terms of a neural network, the above problem can be constructed when we have a single input node (taking in the datapoint x_i), followed by one layer with a single node with weight a and bias b (so that the resulting node has the value $ax_i + b$). Using a sigmoid activation function on this layer ensures the output will be of the form $(1 - \exp(-ax_i - b))^{-1}$.

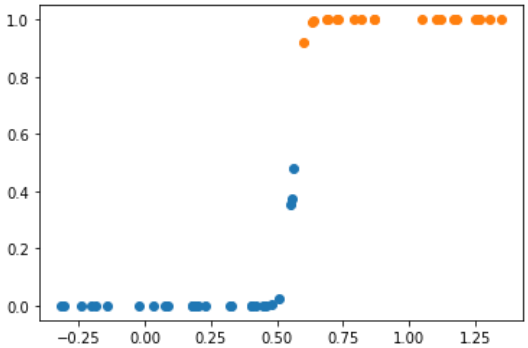
By training our neural network to perform optimally on the entire dataset $\{x_i\}$ we find the values of the weight a and bias b for which we best predict $p(x)$




```
model = Sequential()
model.add(Dense(1, input_dim=1, activation='sigmoid'))
model.compile(loss = 'binary_crossentropy', optimizer='Adam')
```



Plot of the different input points $\{x_i\}$ with their corresponding labels $y(x_i)$ that are shown here in blue (when $y(x_i) = 0$) or in orange (when $y(x_i) = 1$). Note the two classes of input points overlap in the centre around datapoints for which $x = 0.5$.



Plot of the different input points $\{x_i\}$ with their predicted labels $y(x_i)$ that are shown here in blue (when $y(x_i) < 0.5$) or in orange (when $y(x_i) > 0.5$). Note the shape of the logistic (or sigmoid) function $y(x)$ around $x = 0.5$, indicating uncertainty about the class these points belong to.

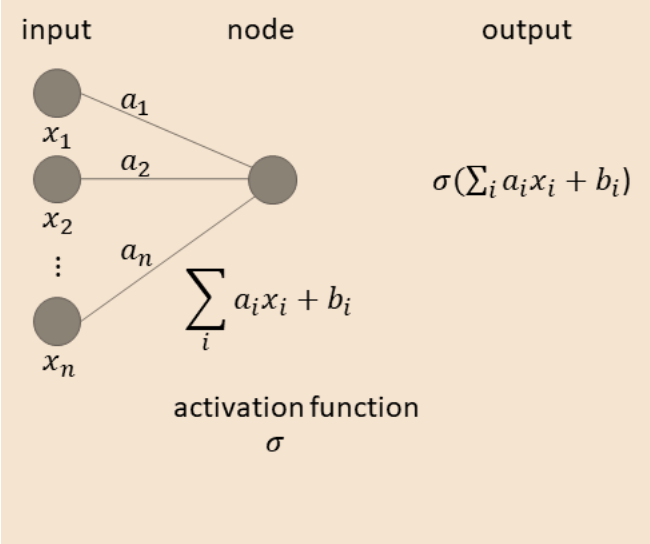
3.2 Multivariable (Univariate) Logistic Regression

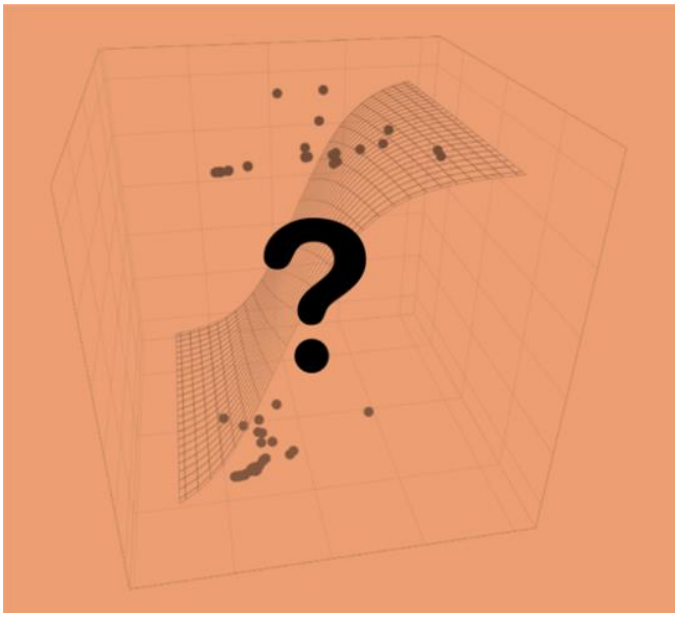
Let's now consider a dataset where we have multiple independent variables $\{x_i^1, x_i^2, \dots\}$, and each set of datapoints is labelled $y(x_i^n)$ to belong to one single binary class that can be either 0 or 1.

To fit the probability function we now use a logistic (or sigmoid) function.

$$p(\vec{x}) = \frac{1}{1 - e^{-(\sum_i a_i x_i + b_i)}}$$

```
model = Sequential()
model.add(Dense(1, input_dim=data_dim, activation='sigmoid'))
model.compile(loss = 'binary_crossentropy', optimizer='Adam')
Still have to test this
```





3.3 Multinomial Logistic Regression

Next we consider a dataset where we have multiple independent variables $\{x_i^1, x_i^2, \dots\}$, and each set of datapoints is labelled $y(x_i^n)$, and can belong to one of N multiple classes.

We can first approach this by considering this problem as running N multivariable logistic regression problems in parallel. Will continue with the rest of this later…

4. Convolutional Layers

Using these we can try to find correlations. Will continue with this later…