1 Introduction

This is a brief overview of the basic theory of neural networks. It's written for my own sake and for the purpose of implementing a neural network in python, so it will be quick and dirty.
A neural network is basically a type of function that be can used to approximate a very general set of functions. They have been used for image recognition, chat bots, data analysis and much more.

2 Definition of a Neural Network

A NN consists of a series of layers. Each layer performs a linear transformation on the input data followed by applying a non-linear function to each element of the output data. The linear transformation typically increases the dimension of the data in the intermediate layers (to allow extraction of different features). The non-linear function is usually called an activation function. It is essential that the activation function is non-linear, as otherwise the NN would just be equivalent to a single linear transformation which is not very useful in general. The output/activation functions/something of the intermediate layers are called neurons. The activation function at the final layer is sometimes called the output function and is typically/always tailored to the problem at hand. For example, for fitting problems one chooses the identity (which is not linear, but that's OK w/ respect to the mathematical properties of the NN since it's only at one layer), while for binary classication problems one would choose the logistic sigmoid function

$$f(x) = \frac{1}{1+\exp(-x)}. \tag{2.1}$$

One typically chooses the same activation function for each layer and each element but this is not essential. The choice of output function is partially determined by the problem at hand, and by interpreting the output of the NN in a probabilistic manner one finds a natural error function: minus the log likelihood. One trains the network by attempting to maximize the probability of the training data or equivalently by minimizing the error. One typically chooses output functions that have the nice property that

$$\frac{\partial E}{\partial x_N} = y - t, \tag{2.2}$$

where E is the error function, x_N is the output of the final layer prior to activation, y is the output of the NN (final layer after activation) and t is the training data. It is not essential but certainly looks nice.

Summarizing all this, we write an n-layered neural network as a function $NN : \mathbb{R}^n \to \mathbb{R}^m$:

$$\begin{aligned} NN(x_0) = y = O(x_n) = O(L_n A(x_{n-1})) = O(L_n A(L_{n-1} A(x_{n-2}))) = ... \\ = O(L_n A(L_{n-1} A(...A(L_1 x_0))), \end{aligned} \tag{2.3}$$

here O is the output function, L_k is the linear transformation at the kth layer, A is the activation function, x_k is the 'data' at the kth neuron/in the kth layer prior to activation and x_0 is the input data.

For shits and giggles we can represent a neural graphically like so:

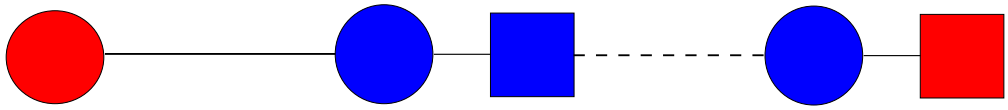

Figure 1

The blue dots or maybe blue dots + blue box are the neurons. The lines represent the linear transformations. The red dot is the input and the red box is the output.

3 Training the neural network

The matrix elements on the linear transformations L_k are to be optimized to minimize the error function. This is done via a training set which contains input data and 'correct' output data. We will just describe gradient descent which calculates the gradient $\nabla_w E$ where w is some matrix element of one of the linear transformations and w is updated at each step according to

$$w \to w - \eta \nabla_w E. \tag{3.1}$$

η is called the learning rate and is typically small. There are also algorithms where the learning rate changes according to some criteria. Here we consider 'on-line training' which just means that at each training step we pick some element in training set (x, t), calculate the gradient of the error function calculated with this training element, update the weight and then move on to some other training element (perhaps picked at random).

We can calculate the gradient using the chain rule. This is called backpropagation in the neural network literature. It is evidently computationally cheaper to use 'backpropagation' rather than calculating the gradient numerically. We just do it because we're badass.

We start at the output neuron and use the fact we have chosen the 'canonical' output function (also known as the canonical link). Let w^n be the matrix of L_n and w^n_{ij} the element at the ith row and jth column.

$$\frac{\partial E}{\partial w^n_{ij}} = \frac{\partial E}{\partial x^n_i}\frac{\partial x^n_i}{\partial w^n_{ij}} = \frac{\partial E}{\partial x^n_i} z^{n-1}_j. \tag{3.2}$$

Here x^k_h is the hth element of the data at the kth layer prior to activation and z^k_h is the hth element of the data at the kth layer after activation. For the canonical link

$$\frac{\partial E}{\partial x^n_i} = y_i - t_i, \tag{3.3}$$

as mentioned. This fact is probably the main reason why we started by expressing the derivate with respect to w_{ij} as a derivate with respect to x instead. Below we will see that we find a recursive relationship between the derivates with respect to x at one layer and the derivates at the next layer. This will allows to calculate the gradients easily.

For a weight at an arbitrary layer we have

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}^k} &= \frac{\partial E}{\partial x_i^k} z_j^{k-1} = \frac{\partial E}{\partial z_i^k}\frac{\partial z_i^k}{\partial x_i^k} z_j^{k-1} = \sum_h \frac{\partial E}{\partial x_h^{k+1}}\frac{\partial x_h^{k+1}}{\partial z_i^k}\frac{\partial z_i^k}{\partial x_i^k} z_j^{k-1} \\ &= \sum_h \frac{\partial E}{\partial x_h^{k+1}} w_{hi}^{k+1}\frac{\partial z_i^k}{\partial x_i^k} z_j^{k-1} \end{aligned} \tag{3.4}$$

The essential point is

$$\frac{\partial E}{\partial x_i^k} = \sum_h \frac{\partial E}{\partial x_h^{k+1}} w_{hi}^{k+1}\frac{\partial z_i^k}{\partial x_i^k}. \tag{3.5}$$

We apply this formula until we reach the final layer at which point we know that $\frac{\partial E}{\partial x_i^n} = y_i - t_i$. Let us write the complete expression

$$\frac{\partial E}{\partial w_{ij}^k} = (y_v - t_v) w_{vc}^n A'(x_c^{n-1}) ... A'(x_g^{k+2}) w_{gh}^{k+2} A'(x_h^{k+1}) w_{hi}^{k+1} A'(x_i^k) z_j^{k-1}. \tag{3.6}$$

All indices except i, j are summed over.

Because of the recursive structure it should be easy to write a function to calculate the gradient.

4 Conclusion

That should be all we need for now. Obviously everything can be generalized and complicated and there are also different implementations that are better at other stuff. There's plenty more to do.