Maths Skills (MTCS) G5071

Lecture 4

Kingsley Sage
Room 5C16, Pevensey III
khs20@sussex.ac.uk

Lecture 4

- Introduction to Applications
  - The applications in cognitive science that are used as examples on this course are non-symbolic computations for vision, neural networks and adaptive behaviour.
  - You will have already met some of these ideas in Further AI, but here the maths will be more important and uses a consistent notation.
  - Examples will be using differential calculus on a variety of functions for neural net learning.
  - You will be able to see how the formal maths notation maps onto the computational structures in the sample Artificial Neural Networks (ANNs).
  - The idea of gradient descent on an energy landscape is common to many computational systems.

Artificial Neural Networks

- ANNs are formed by connecting units which, very approximately, correspond to biological neurons.
- In an important class of ANNs, inputs and outputs are represented by real numbers, units compute an output as a functions of its input.
- A single unit has *N* inputs, which will be called x_1, x_2 etc up to *N*; the output of the unit is *y*, so we can write $y=f(x_1,x_2,\ldots,x_N)$ to show the output is a function of the inputs.
- The form of the response function *f* determines the behaviour of the network which can be classified as linear or non-linear.
- A very simple response function would be to just add up all the inputs, but these are usually weighted to reflect how much the unit is affected by a particular unit.

Linear units and rules

- For each input x_i, there is a corresponding weight w_i and the unit combines its inputs by computing a weighted sum according to:

$$a = \sum_{i=1}^{N} x_i w_i$$

- A unit can simply pass this activation to its output $y=a$ and called a linear unit because if you plot *y* against any particular input, keeping all the inputs constant, then you get a straight line.

Linear units and rules

- Neural networks are trained by adjusting the weights to improve their performance – we regards the weights as variables so it makes sense to regard the output as a function of the inputs and of the weights:

$$y = f(x_1, x_2, \ldots, x_N, w_1, w_2, \ldots, w_N)$$

- The inputs and weights affect the outputs:

$$\frac{\partial y}{\partial x_i} = w_i \quad \text{and} \quad \frac{\partial \mathrm{y}}{\partial \mathrm{w_i}} = x_i$$

- This partial derivative can be used in a program to train a linear ANN as it estimates the local gradient for each weight.

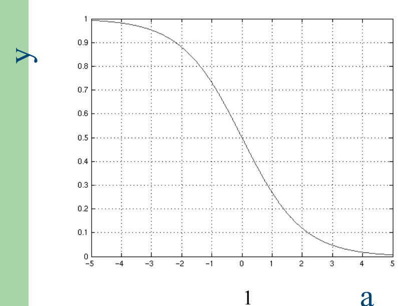
Non-linear units

- ANNs built from linear functions have a very limited range of responses.
- We can give the networks enormously greater computational power by making the relationship between activation and output non-linear.
- Instead of *y*=*a*, interesting ANNs use a relationship such as:

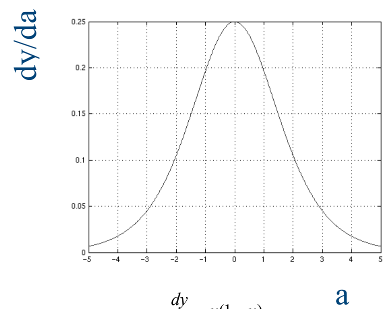
$$y = \frac{1}{1 + e^{-a}}$$

Non-linear units

- This functions is sometimes called the logistic function – note that *a* is just the weighted sum, of the inputs, as for the linear unit.



$$y = \frac{1}{1+e^{-a}}$$



$$\frac{dy}{da} = y(1-y)$$

Non-linear rules

- From the analysis of a linear unit, where *a*=*y*, we know that:

$$\frac{\partial a}{\partial x_i} = w_i \quad \text{and} \quad \frac{\partial a}{\partial w_i} = x_i$$

- The next step requires the application of the rules of differentiation to the logistic function. It can be shown that:

$$\frac{dy}{da} = y(1-y)$$

Non-linear rules

- Next, by the chain rule:

$$\frac{\partial y}{\partial x_i} = \frac{dy}{da} . \frac{\partial a}{\partial x_i} \quad \text{and} \quad \frac{\partial y}{\partial w_i} = \frac{dy}{da} . \frac{\partial a}{\partial w_i}$$

- We can get:

$$\frac{\partial y}{\partial x_i} = y(1-y)w_i \quad \text{and} \quad \frac{\partial y}{\partial \mathrm{w_i}} = y(1-y)x_i$$

- Which can be used in a program to compute the derivatives used in the training of non-linear ANNs

Non-linear learning rule

- We want to change the weights so to take a step downhill on the error surface – this is called *gradient descent*.
- Each particular weight needs to be changed in the right direction to reduce the error *E*, and the bigger effect, the more it should be changed.

Non-linear learning rule

- For on-line learning, the adjustment should be proportional to minus the partial derivative of the error *E* with respect to that weight:

$$\Delta w = -\alpha . \frac{\partial E}{\partial w_i}$$

- The constant α is used to keep the step small – typically set to a value between 0.00001 and 0.2 by the experimenter.

Non-linear learning rule

- Now we can train a single unit by putting everything together to give the adjustment to a weight after the presentation of an input *t*:

$$\Delta w_i = -\alpha(2(y-t)y(1-y))x_i$$

- This is the essential weight adjustment rule for training.

Training a network

- We shall stick to the simple case of the pattern associator network.
- The first change is in notation as we need to distinguish between the units, and to do this we have to use an extra subscript in some places.
  - Output of unit *y*: y_j
  - Activation of unit *y*: a_j
  - Target value for unit *y*: t_j
  - Weight for input *i* going to unit *j*: w_{ji}
- So the error function *E* is the sum:

$$E = \sum_j (y_j - t_j)^2$$

Training a network

- E is called the sum of squares error function, and we simply have to keep track of which unit we are talking about:

$$\frac{\partial E}{\partial y_j} = 2(y_j - t_j)$$

- We can go on and add layers and derive the ANN backpropagation rule but this is beyond the scope of the course here.

Next time …

- Introduction to matrices and vectors
  - Maths notation
  - Matrix and vector multiplication
  - Matrix operations