Using neural nets to recognize handwritten digits

We first focus on the problem of recognizing handwriting. Recognizing handwriting intuitively seems easy, but is a hard problem to specify an algorithmic solution for. Neural networks approach the problem by learning from training examples.

Perceptrons

Perceptrons were one of the first proposed artificial neuron models (more commonly used today is the *sigmoid neuron*). They take several binary inputs and produce a single binary output. This is done by weighting each input and then thresholding the output.

We can make increasingly complex and abstract decisions by *layering* our nodes, where deeper layers of nodes take as inputs the outputs of earlier layers.

With *learning algorithms*, we can automatically tune the weights and biases (thresholds) of a neural network, in response to external stimuli rather than by direct programmer intervention.

Sigmoid neurons

The downside of networks of perceptrons is that small changes in input can produce large changes in output. Thus, it's challenging to modify the behavior of the network in this small way towards the desired behavior.

Sigmoid neurons overcome this problem. Like perceptrons they have multiple inputs. However, these inputs need not be binary values – they can be anything between 0 and 1. The output is defined by

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}$$

Despite seeming dissimilar, the sigmoid function produces similar results to perceptrons. Supposing $w \cdot x + b$ is a large positive number, then $\sigma(z) \approx 1$, and if it is very negative, $\sigma(z) \approx 0$. Thus, we can see that the sigmoid function asymptotically approximates a perceptron.

Through some analysis, we can also see that the output value is a linear function of displacements in weights and biases. Thus, the sigmoid neuron is a model that we can gradually displace towards the desired outputs, as desired.

The architecture of neural networks

We call the layer of input neurons the *input layer*, the layer of output neurons the *output layer*, and all others *hidden layers*. Networks that always feed output values forward in the network (rather than having loops) are called *feedforward networks*. Networks that relax this constraint and allow loops are called *recurrent neural networks*.The problem of input (confusingly) depending on output is solved by having neurons firing and waiting, rather than continuously.

A network to classify handwritten digits

In recognizing digits, the first problem we have to consider is how to split up an image of lots of digits into a series of images, each containing a single one. This problem is called *segmentation* (not considered further).

To recognize digits, we use a three-layer network. The input layer is just the raw pixel data – each input image is a 28×28 image, so we have 784 input neurons with values between 0.0 and 1.0 based on the relative blackness of the pixels.

The hidden layer will have some number n of neurons, that will be fixed based on experimentation.

The output layer contains 10 neurons, each indicating how much the network "thinks" it was that the input was a given digit. Note that there are other ways we could plausibly have defined this – four neurons and thresholds, for example. In practice, this approach (10 neurons) has turned out to be effective. We can also justify this by thinking about how information is being encoded in the network.

Learning with gradient descent

To quantify how well we are doing in our classification given our current weights and biases, we employ a *cost function*

$$C(w, b) = \frac{1}{2n} \sum_x ||y(x) - a||^2$$

where w is the collection of all weights, b the collection of all biases, n the total number of training inputs, and a the output vector for the given input. It is sometimes known as the *mean squared error* of the *quadratic cost function*.

We want to minimize this cost function. It's a good idea to do this rather than trying to optimize directly for the number of images we classify correctly, or other metrics, since the function is "smooth". However we could reasonably swap out this function for others, and this is done sometimes in practice.

We optimize using *gradient descent*. Note that in terms of the gradient and change in weight vectors (here, v), we have

$$\Delta C \approx \nabla C \cdot \delta V$$

Thus, we can determine our choice of ∇C such that we always decrease the cost over time. Our update rule is

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}$$

Computing all of the necessary derivatives can be costly, motivating an approach called *stochastic gradient descent*. It works by computing the gradient for a small sample of the training inputs and averaging it, providing an estimate of the true gradient. That is,

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j}$$