Artificial Intelligence – TP

MNIST Digit Recognition

The objective is to program an artificial neural network from scratch to recognise single digits. We will use a classic benchmark: `http://yann.lecun.com/exdb/mnist/`. The files are separated into training data and test data. They are using a binary file format. Some parser exist in several langages, e.g. `https://pypi.python.org/pypi/python-mnist` for Python.

It will be better if the data is normalised so that values range from 0.0 (white) to 1.0 (black), instead of from 0 to 255.

You can choose the programming language you prefer, though the instructor can best understand C, C++, Python, Java, and Haskell.

1 Representation of the Artificial Neural Network

Though we will produce a generic implementation, the first tests will be done with the following architecture:

- An input layer with $n_0 = 784$ (28×28) neurons;
- A hidden layer with 30 neurons;
- An output layer with 10 neurons (digits from 0 to 9);
- All neurons use the logistic activation function: $g(x) = \frac{1}{1+e^{-x}}$ with derivative $g'(x) = g(x)(1 - g(x)$;
- The loss function is the cross-entropy loss function;
- The learning rate is $\alpha = 0.05$.

A neural network can basically be represented by a list of matrices containing the weights between the layers. We have one matrix per layer (except the input layer, which will not be modelled explicity), so that the matrix w^ℓ for layer $\ell \geq 1$ has n_ℓ rows and $n_{\ell-1}$ columns (with n_ℓ the number of neurons in layer ℓ).

It will be more convenient for the implementation to model the biases explicitly instead of assuming we have one additional neuron per layer with an activation that is always 1. We thus have an additional list of column vectors b^ℓ with size $n_\ell \times 1$ (one vector for each layer).

With the above figures we thus have a list of two matrices, respectively of size 30×784 and 10×30, plus a list of two vectors of size 30×1 and 10×1.

In the data structure for representing the network (a class, or a record), it should also be useful to add fields for the number of inputs and the learning rate.

2 Forward Computation

At the beginning of each epoch, the (60 000) examples are partitioned into minibatches. We will use minibatches of size $m = 10$. We will see how to generate those minibatches later.

The forward and backward computations are done for the m examples of a minibatch at the same time. Thus the input a^0 to the neural network is a matrix $784 \times m$ where each column is the contents of a single image corresponding to one example in the minibatch.

The forward computation computes the weighted inputs $z^\ell = w^\ell a^{\ell-1} + b^\ell \times \vec{1}$ (with $\vec{1}$ a row vector of size $1 \times m$) and activation $a^\ell = g(z^\ell)$ for all layers ℓ. Note that all z^ℓ and a^ℓ are matrices of size $n_\ell \times m$, where each column represents the weighted input or activation for one given example of the minibatch. The function performing the forward computation should give as a result the list of all the weighted inputs, and the list of all the activations.

3 Backpropagation

Since we use the cross-entropy loss function, and logistic neurons, by construction, the modified error Δ^L in the output layer is $a^L - y$, where a^L is the activation at the output layer, given by the forward computation, and y is a $10 \times m$ matrix, where each column represents the expected answer for the corresponding example in the minibatch. Thus if the label of the i-th example in the minibatch is 2 then the i-th column of the y matrix will be the column vector $(0, 0, 1, 0, 0, 0, 0, 0, 0, 0)^\top$ (note the first index corresponds to digit 0).

We can then iteratively compute the errors for each layer using the formula: $\Delta^{\ell-1} = (w^\ell)^\top \Delta^\ell \circ g'(z^{\ell-1})$ where $\circ$ is the pointwise matrix multiplication (the *Hadamard product*). Again, note that all those Δ^ℓ are matrices $n_\ell \times m$.

Finally, we can update the weights with $w^\ell \leftarrow w^\ell - \frac{\alpha}{m} \Delta^\ell (a^{\ell-1})^\top$ and the biases with $b^\ell \leftarrow b^\ell - \frac{\alpha}{m} \Delta^\ell \vec{1}^\top$, where $\vec{1}^\top$ is a column vector of size $m \times 1$.

4 Weight Initialisation

Biases in the network can be initialised to 0. Weights should not however: initialise the weights in w^ℓ using a Gaussian centered at 0 and with standard deviation $\frac{1}{\sqrt{n_{\ell-1}}}$.

You can also use a standard deviation of 1 or even a uniform distribution instead of Gaussian. These should give a slightly slower convergence, though.

5 Epochs and Minibatches

Training the network is done through epochs of training. At the beginning of each epoch, the (60 000) examples are randomly shuffled and partitioned into minibatches.

The easiest way to do this is to generate a vector of numbers from 0 to 59 999, representing the indices of the examples in the vectors of training images and labels, randomly shuffle that vector of indices (e.g. by doing 60 000 random permutations) and take the elements in the order given by that vector, m indices at a time.

Using this we can compute the input matrix a^0 for the forward computation and the expected result matrix y, and call the forward computation and finally the backpropogation.

At then end of an epoch of training we should test the accuracy of the trained net. In order to do this, we should use the test data (10 000 images and labels) provided by the MNIST

dataset. For each example in the test data, compute the answer of the network by calling the forward computation, and by simply taking the index of the neuron with the biggest activation in the output layer. Compare this to the expected label given by the data, and count the number errors or successes. Once all data has been tested, you can divide by 100 to get a percentage.

All is now set to train and test the network. Try with 30 training epochs; you should get about 95% of accuracy. Note that you can also test the accuracy of your network before any training and you should get about 10%: one chance in ten to get the good result, which is consistent with the random initialisation done. After the first training epoch you should already be close to 90%.