MNIST Digit Classification Using Artificial Neural Network

Bennett Gale - s5053019

May 2017

Contents

1 Hand Calculations **2**
- 1.1 Forward Propagation . . . 2
  - 1.1.1 Initial Values . . . 2
  - 1.1.2 Hidden Layer Activity . . . 2
  - 1.1.3 Output Layer Activity . . . 3
- 1.2 Backpropagation . . . 4
  - 1.2.1 Computing the Gradients . . . 4
  - 1.2.2 Updating the Weights . . . 5

2 MNIST Digit Classification **7**
- 2.1 Implementation . . . 7
- 2.2 Results . . . 7
  - 2.2.1 Constant Hyperparameters . . . 7
  - 2.2.2 Variable Learning Rate . . . 8
  - 2.2.3 Variable Minibatch Size . . . 8

1 Hand Calculations

1.1 Forward Propagation

1.1.1 Initial Values

The forward pass involves propagating the input through the network and applying the weights and activation functions at each neuron. The input X is represented as a column vector of length equal to the number of input neurons in the network. In order to compute more than one training example simultaneously on one forward pass, the input vectors can be stacked horizontally to form a single matrix of dimensions $s^{(1)} \times k$, where $s^{(l)}$ is the number of units in the given layer (not including the bias units) and k is the number of examples in the given batch.

Similarly, the expected output of the network, $a^{(3)}$ is represented as a matrix of dimensions $s^{(3)} \times k$. The input and output data can thus be represented as follows:

$$X = \begin{bmatrix} 0.1 & 0.1 \\ 0.1 & 0.2 \end{bmatrix} \qquad Y = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The initial weight values are also represented in matrices as follows.

$$W^{(1)} = \begin{bmatrix} 0.1 & 0.1 \\ 0.2 & 0.1 \end{bmatrix} \qquad W^{(2)} = \begin{bmatrix} 0.1 & 0.1 \\ 0.1 & 0.2 \end{bmatrix}$$

The hidden and output layers have associated bias terms, which always have an output of 1 and are added as a product of their respective bias weights to the output activity of the proceeding layer. The bias weights are given as follows:

$$B^{(1)} = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix} \qquad B^{(2)} = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix}$$

1.1.2 Hidden Layer Activity

The net input of a given neuron is the sum of the outputs of its connected units in the preceding layer multiplied by the weights along their respective synapses. These operations can be expressed using matrix multiplication by taking the dot product of the previous layer's output matrix and their associated weights:

$$Z^{(l)} = a^{(l-1)} W^{(l-1)}$$

Where $a^{(l-1)}$ is the output of the previous layer. In order to incorporate the bias weights into the calculation, $W^{(1)}$ can be augmented with the bias weights $B^{(1)}$, and the input matrix can be augmented with a single row of 1s representing the output of the bias units. The net input of the hidden layer can therefore be given as:

$$
\begin{aligned}
Z^{(2)} &= W^{(1)}X \\
&= \begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.2 & 0.1 & 0.1 \end{bmatrix} \begin{bmatrix} 0.1 & 0.1 \\ 0.1 & 0.2 \\ 1 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 0.12 & 0.13 \\ 0.13 & 0.14 \end{bmatrix}
\end{aligned}
$$

Next, the output of the layer is calculated by applying a logistic activation function in order to produce non-linear output.

$$
\sigma = \frac{1}{1 + e^{-x}}
$$

Applying the logistic activation function element-wise to the net input of layer 2, $z^{(2)}$, gives:

$$
\begin{aligned}
a^{(2)} &= \sigma(Z^{(2)}) \\
&= \begin{bmatrix} \sigma(0.12) & \sigma(0.13) \\ \sigma(0.13) & \sigma(0.14) \end{bmatrix} \\
&= \begin{bmatrix} 0.52996405 & 0.53245431 \\ 0.53245430 & 0.53494295 \end{bmatrix}
\end{aligned}
$$

1.1.3 Output Layer Activity

The activity of the output layer is calculated similarly to that of the hidden layer, except that the output from the hidden layer, $a^{(2)}$, now forms the input.

$$
\begin{aligned}
Z^{(3)} &= W^{(2)}a^{(2)} \\
&= \begin{bmatrix} 0.20624184 & 0.20673973 \\ 0.25948727 & 0.26023402 \end{bmatrix}
\end{aligned}
$$

Applying the logistic activation function element-wise, we get:

$$
\begin{aligned}
a_{(3)} &= \sigma(Z_{(3)}) \\
&= \begin{bmatrix} \sigma(Z_1^{(3)}) & \sigma(Z_3^{(3)}) \\ \sigma(Z_2^{(3)}) & \sigma(Z_4^{(3)}) \end{bmatrix} \\
&= \begin{bmatrix} 0.55137847 & 0.55150162 \\ 0.56451025 & 0.56469382 \end{bmatrix}
\end{aligned}
$$

1.2 Backpropagation

Backpropagation allows the weights of the neural network to be incrementally updated using gradient descent to minimise a cost function.

The quadratic cost function is used to evaluate the output of the network:

$$J(w,b) = \frac{1}{2n}\sum_{i=1}^{n} ||a^{(3)} - Y||^2$$

This represents the error of the output with respect the training labels. By backpropagating this error through the network the weights can be changed proportionally to their contribution to the total error, thereby minimising the cost function and increasing the accuracy of the network's predictions.

1.2.1 Computing the Gradients

In order to backpropagate through the output layer, the partial derivative of the cost function with respect to the layer 3 weights must be computed. That is, the rate that the total cost changes given changes in each weight.

Applying the chain rule, the partial derivative can be expressed as:

$$\begin{aligned}\frac{\partial J(w,b)}{\partial W^{(2)}} &= \frac{\frac{1}{2n}\sum_{i=1}^{n} ||a^{(3)} - Y||^2}{\partial W^{(2)}} \\ &= -(Y - a^3)\frac{\partial a^{(3)}}{\partial Z^{(3)}}\frac{\partial Z^{(3)}}{\partial W^{(2)}} \\ &= -(Y - a^3) \circ \sigma'(Z^{(3)})a^{(2)T}\end{aligned}$$

Let $\delta^{(3)} = -(Y - a^3) \circ \sigma'(Z^{(3)})$. This is calculated using element-wise multiplication between the error and the derivative of the activation function, which results in:

$$\delta^{(3)} = \begin{bmatrix} -0.11097114 & 0.13641259 \\ 0.13877831 & -0.10700466 \end{bmatrix}$$

The gradient $\nabla W^{(2)} J(w,b)$ can then be computed as follows. Note that this operation automatically sums the error across every example.

$$\begin{aligned}\nabla W^{(2)} J(w,b) &= \delta^{(3)} a^{(2)T} \\ &= \begin{bmatrix} 0.01382276 & 0.0138859 \\ 0.01657242 & 0.01665172 \end{bmatrix}\end{aligned}$$

Because the bias units all have an output of one, the bias gradient can be computed simply by taking the mean of $\delta^{(3)}$ along the row axis:

$$\nabla B^{(2)} J(w,b) = \begin{bmatrix} 0.01272073 \\ 0.01588682 \end{bmatrix}$$

The gradients of $W^{(1)}$ and $B^{(1)}$ are calculated similarly, but the fact that they influence the outputs of proceeding layers must be taken into account.

$\delta^{(2)}$ can be calculated as:

$$\begin{aligned}\delta^{(2)} &= W^{(2)T}\delta^{(3)} \circ \sigma'(Z^{(2)}) \\ &= \begin{bmatrix} 0.00069268 & 0.0007321 \\ 0.00414709 & -0.00193044 \end{bmatrix}\end{aligned}$$

$W^{(2)}$'s gradient, $\nabla W^{(1)} J(w,b)$ can be computed as follows.

$$\begin{aligned}\nabla W^{(1)} J(w,b) &= \delta^{(2)} X^T \\ &= \begin{bmatrix} 1.42478355e-04 & 2.15688433e-04 \\ 2.21664739e-04 & 2.86203723e-05 \end{bmatrix}\end{aligned}$$

As with $\nabla B^{(2)} J(w,b)$, $\nabla B^{(1)} J(w,b)$ can be computed by taking the mean of $\delta^{(2)}$ along the row axis:

$$\nabla B^{(1)} J(w,b) = \begin{bmatrix} 0.00071239 \\ 0.00110832 \end{bmatrix}$$

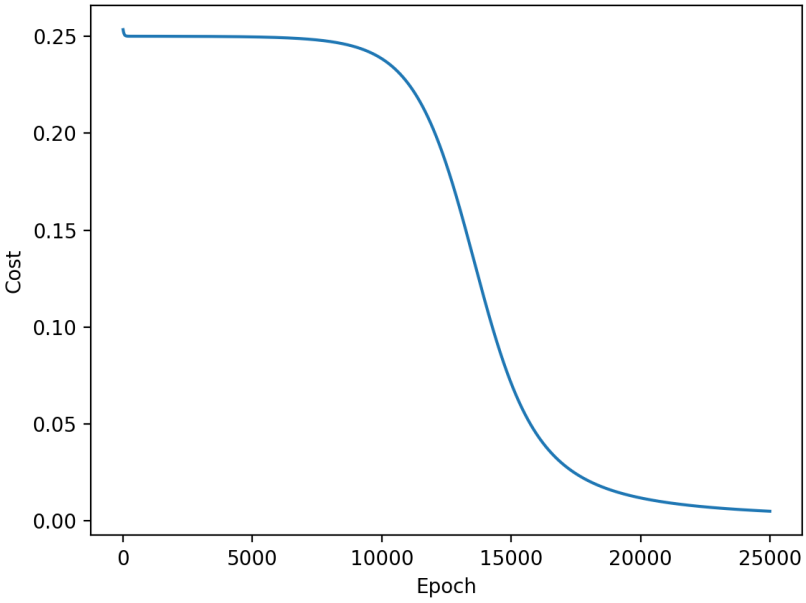
1.2.2 Updating the Weights

The final part of stochastic gradient descent is to update all of the weights in the network. This is done by subtracting from the current weight the gradients multiplied by a scalar learning rate, α. As they examples were added through matrix multiplication in a previous step, they do not need to be summed. Let $\alpha = 0.1$.

$$\begin{aligned}W^{(1)} &= W^{(1)} - \alpha \times \nabla W^{(1)} J(w,b) \\ &= \begin{bmatrix} 0.09998575 & 0.09997843 \\ 0.19997783 & 0.09999714 \end{bmatrix} \\ W^{(2)} &= W^{(2)} - \alpha \times \nabla W^{(2)} J(w,b) \\ &= \begin{bmatrix} 0.09861772 & 0.09861141 \\ 0.09834276 & 0.19833483 \end{bmatrix} \\ B^{(1)} &= B^{(1)} - \alpha \times \nabla B^{(1)} J(w,b) \\ &= \begin{bmatrix} 0.09992876 \\ 0.09988917 \end{bmatrix} \\ B^{(2)} &= B^{(2)} - \alpha \times \nabla B^{(2)} J(w,b) \\ &= \begin{bmatrix} 0.09872793 \\ 0.09841132 \end{bmatrix}\end{aligned}$$

These calculations were implemented and tested in Python. Figure 1 shows the resulting decrease in the cost function over 25000 epochs.

Figure 1: Change in cost over epoch



2 MNIST Digit Classification

2.1 Implementation

The classifier is implemented using stochastic gradient descent with minibatches to update the weights. The quadratic cost function (below) is used to evaluate the average error of each minibatch and tune the weights accordingly.

$$J(w,b) = \frac{1}{2n}\sum_{i=1}^{n} ||a^{(3)} - Y||^2$$

Unlike the regression calculations in part 1 which use the logistic sigmoid function, the multi-class classifier uses the softmax function as the nonlinearity of the output layer. This produces an output vector of normalised values that add up to 1 and represent the probability of each class. This function is given as follows, where k is the dimension of vector z:

$$\sigma = \frac{e^{z_j}}{\sum_{k=1}^{k} e^{z_k}}$$

The use of stochastic gradient descent with minibatches means the weights are updated incrementally after the average error of each minibatch is computed. Algorithm 1 describes the training process used in the implementation.

Algorithm 1 The training process using SGD with minibatches

```
procedure TRAIN(training_data)
    nn ← newNetwork
    for i ← 0; i <= number_of_epochs; i++ do
        for each data partition of minibatch size do
            Do forward pass
            Backpropagate the resulting error
            Update the weights proportionally to their contribution to the error
        end for
        Shuffle and repartition the data
    end for
end procedure
```

2.2 Results

2.2.1 Constant Hyperparameters

The implemented program was trained on a subset of the MNIST data set with 50,000 unique examples. For training a learning rate of 3 is used. The neural network consists of 784 input neurons, with each neuron representing a member of the flattened pixel matrix of each example. The hidden layers are of size 30

and 10 respectively. The data is partitioned into minibatches of 20 examples with 30 epochs.

Figure 2 shows the relationship between training error and generalisation. As the model fits the training data the prediction accuracy on the test data increases at a similar rate. The generalisation accuracy smoothly increases until it begins to level off, peaking at a prediction accuracy of 93.47%. The accuracy begins to fluctuate after approximately 10 epochs, which indicates the additional variance caused by overfitting.

2.2.2 Variable Learning Rate

Figure 3 shows the effect of varying learning rates, with all other hyperparameters consistent with those of the tests done in part 2.2.1.

It can be observed that a learning rate of 1.0 provided superior generalisation. The poor prediction accuracy that occurs with a learning rate of 0.001 is due to the gradient descent iterating too slowly such that a minimum is never reached before training completes. Likewise, larger learning rates 10 and 100 "overshoot" the desired minimum, causing similarly poor results.

2.2.3 Variable Minibatch Size

Figure 4 similarly demonstrates the relationship between minibatch size and generalisation. With other hyperparameters consistent with those of part 2.2.1, the minibatch size was set to each of the values 1, 5, 10, 20 and 100. While the smaller values 1, 5, 10 and 20 had minimal affect on accuracy, a minibatch size of 100 can be seen to have caused a significant decrease. This contradicts the expectation that a larger minibatch size should result in greater accuracy due to providing more data per weight update. This may be due to an unknown side-effect in the implementation.

Figure 2: The change in training error and generalisation accuracy over 30 epochs.

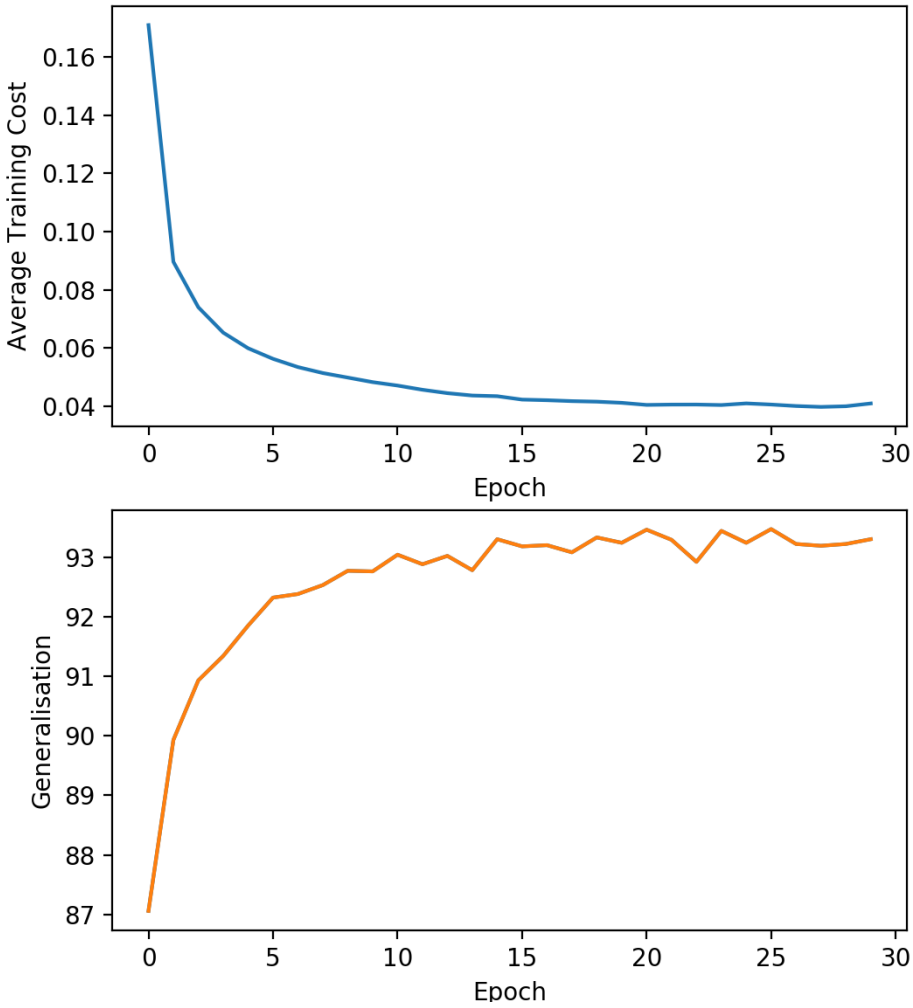


Figure 3: The effect of varying learning rates on generalisation

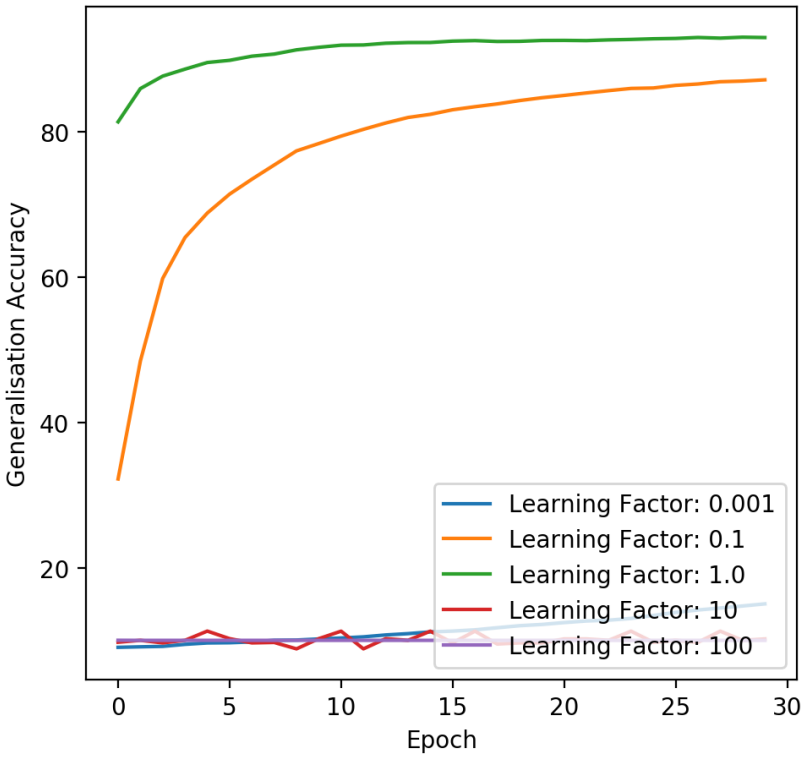


Figure 4: The effect of varying minibatch size on generalisation