CSC411 Machine Learning
Project 2: Deep Neural Network

Ariel Kelman
Student No: 1000561368

Gideon Blinick
Student No: 999763000

20 February 2018

1 Introduction

This section provides an introduction to Project 2, including the data used, and notes on reproducing the results. Sections 2-6 discuss the implementation of a simple neural network to classify images from the MNIST database, section 7 is a theoretical anlaysis of the efficiency of backpropogation, and sections 8-10 work with deep neural networks.

1.1 Digits

The following figure shows 10 random images from the training set of each of the digits.

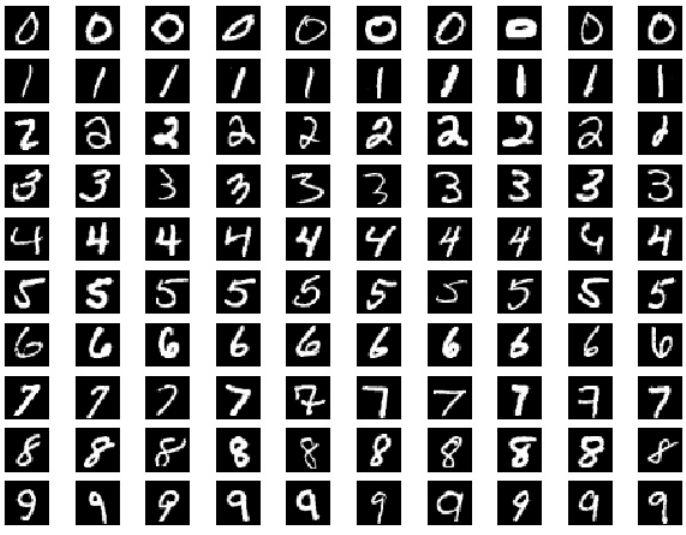

Figure 1: 10 random samples from the training set for each digit. This image was generated using `plot_samples()`.

The data was downloaded from the assignment webpage, and imported into Python using the provided code. The data was already divided into training and testing sets...

1.2 Results & Report Reproducibility

All results and plots can be generated using the python file... To reproduce the report, simply run it through latex.

2 The Network

The following function implements a neural network with no hidden layers, with the output passed through a softmax layer to estimated probabilities.

```
def no_hidden_layers():
```

The network is described by the weights and biases from the 784 = 28*28 inputs (representing pixel intenisites of the input image) to ten "output" nodes (with the identity as the activation function). The output from this layer is what is passed through the sofmax function $p_k = \frac{e^{o_k}}{\sum_q e^{o_q}}$.

The weights are represented as a 784 by 10 matrix W, where w_{ij} (i^{th} row, j^{th} column) represents the weight from the i^{th} input to the j^{th} output. When computing the network on a given sample, the transpose of W is multiplied by the column vector (or matrix when computing on multiple samples) representing the input. The biases are represented by a 10 by 1 vector, one entry for each output.

Throughout the code, $y\_$ is a matrix representing the correct results for each image; each column is a vector of zeros with a 1 in the place representing the correct digit. y is a matrix representing the output probabilities from the network of the same dimensions as $y\_$; each column is a vector representing the probabilities for each digit for a particular sample.

3 Gradient

The cost function is taken to be $-\sum_k y_k ln(p_k)$ for one sample, where y_k is 1 for the correct class and 0 otherwise, and p_k is the prediction probability for class k. Writing this in vector notation (replacing the sum with a vector dot product) and summing over all the samples gives:

$$C = -\sum_s y^{(s)} ln(p^{(s)})$$

where ln is applied pointwise, and s is an index over the samples. $y^{(s)}$ is a vector of 0's with a 1 in the position representing the correct digit.

The following results will be used in the derivations throughout this section:

$$\frac{\partial p_k}{\partial o_q} = \begin{cases} -p_k p_q & \text{if } k \neq q \\ p_q(1-p_q) & \text{if } k = q \end{cases}$$

These results follow directly from the definition of the softmax function.

3.1 Gradient wrt w_{ij}

First, we derive $\frac{\partial C}{\partial o_q}$ (for one sample), to be used in the derivation of $\frac{\partial C}{\partial w_{ij}}$.

$$\begin{aligned}
\frac{\partial C}{\partial o_q} &= \left[\frac{\partial C}{\partial p_q}\frac{\partial p_q}{\partial o_q} + \sum_{k\neq q}\frac{\partial C}{\partial p_k}\frac{\partial p_k}{\partial o_q}\right] \\
&= -\left[y_q\frac{1}{p_q}p_q(1-p_q) + \sum_{k\neq q} y_k\frac{-1}{p_k}p_k p_q\right] \\
&= -\left[y_q(1-p_q) - \sum_{k\neq q} y_k p_q\right] \\
&= \left[-y_q + \sum_k y_k p_q\right] \\
&= p_q - y_q
\end{aligned}$$

The last line follows because $\sum_k y_k = 1$. Using this result, and applying the chain rule gives:

$$\begin{aligned}
\frac{\partial C}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}}\sum_s y^{(s)} ln(p^{(s)}) \\
&= \sum_s\sum_q \frac{\partial C}{\partial o_q}\frac{\partial o_q}{\partial w_{ij}} \\
&= \sum_s\sum_q (p_q - y_q)x_i\delta_{jq} \\
&= \sum_s (p_j - y_j)x_i
\end{aligned}$$

As the outputs are linear functions of both the inputs and the weights, the derivative with respect to a weight is given by the value of the input to that weight, namely x_i. The Kronecker delta δ_{jq} results from $\frac{\partial o_q}{\partial w_{ij}}$ being nonzero only when $j = q$. The (s) superscript indicating the sample index was omitted throughout the derivation for clarity of presentation.

3.2 Vectorized Gradient Code

The following code computes the gradient with respect to the wieghts and biases (representing all input training images in a matrix X, 784 by the number of samples) gives

```
        def grad():
```

4 Training

The neural network was trained on the full training set of 60000 images without momentum. A learning rate of xxx was used for xxx iterations, giving a training accuracy of xxx and a testing accuracy of xxx. The weights and biases were initialized using a uniform distribution to values between 0 and 1.

The following figure shows a learning curve...

5 Training with Momentum

The neural network was trained on the full training set of 60000 images with momentum, set to xxx. A learning rate of xxx was used for xxx iterations, giving a training accuracy of xxx and a testing

accuracy of xxx. The weights and biases were initialized using a uniform distribution to values between 0 and 1.

The following figure shows a learning curve...

6 Analysis of Momentum

For the analysis in this section, the weights w_{ab} and w_{cd} were used (recall that W is a 784 by 10 matrix). After training the network as described in the previous section, these weights had the values $w_{ab} = xxx$ and $w_{cd} = yyy$.

6.1 Contour Plot

The following plot shows the variation of the cost function (negative log loss) around the values for w_{ab} and w_{cd} chosen above.

6.2 Trajetory without Momentum

6.3 Trajetory with Momentum

6.4 Discussion

7 Vectorized Backpropogation

Consider a neural network with N layers each containing K neurons. By convention, the input layer will not counted toward N layers, but the final output layer will be counted (thus there are N sets of weights and biases that must be learned). Any function (such as softmax) applied to the final layer is ignored; none of these conventions will affect the asymptotic runtime. We also assume that the results of the forward pass of the network are cached, to be used for both vectorized backpropogation and computing the gradient with respect to each weight independenly. We can also ignore the biases; including them could be accomodated by adding an extra neuron to every layer (except the last) with a constant value of 1. This would be equivalent to increasing K by 1.

Denote each layer by the numbers $1, 2...N$ (we can refer to the input layer as layer 0). The efficiency of computing the gradient with respect to a weight to/from a particular layer does not depend on which weight it is, so $w^{a\to b}$ can stand for an arbitrary weight from layer a to b. A particular neuron in layer n will be denoted by σ^n with a subscript to denote that only on neuron in the layer is intended.

Then the following holds:

$$\frac{\partial C}{\partial w^{N-1\to N}} = \frac{\partial C}{\partial \sigma_q^N}\frac{\partial \sigma_q^N}{\partial w^{N-1\to N}}$$

$$\frac{\partial C}{\partial w^{N-2\to N-1}} = \sum_q \frac{\partial C}{\partial \sigma_q^N}\frac{\partial \sigma_q^N}{\partial \sigma_m^{N-1}}\frac{\partial \sigma_m^{N-1}}{\partial w^{N-1\to N}}$$

Since there are K neurons in each layer,

THE ABOVE NOTAION IS REALLY CONFUSING, I'LL TRY TO IMPROVE IT