MTH 786P
Machine Learning
Queen Mary University of London

Adrika Datta
Student ID: 220997078

January 2023

Contents

Chapter 1

Introduction

'AI is the new electricity'
-Andrew Ng

AI has reached new heights since the dawn of GPUs and high performance computing. Although we haven't yet answered Alan Turing's question "Can machines think?" in the seminal paper "I.—COMPUTING MACHINERY AND INTELLIGENCE", we have made great progress in computer vision, natural language processing - some of which even exceed human abilities. The MNIST dataset, part of the larger NIST dataset, is one such problem where machines are almost as capable as us humans. It is a problem that is considered "solved"; specially with the advent of powerful convolutional networks, vision transformers, it is still considered the "hello world!" problem in the field of machine learning. In this work, a simple neural network model is presented that can recognise handwritten digits with 92% accuracy. This model is created with pure Python (well certainly NumPy) and takes no help from any high level APIs like TensorFlow and/or Keras. The first chapter takes a deep dive into the data and the relevant processing techniques employed to prepare the dataset, the second chapter presents a brief overview of neural networks, followed by the Training and Validation chapter and finally a section is dedicated to conclusion.

Chapter 2

Data

In this work a classification machine learning model is created - to recognise handwritten digits from the MNIST dataset. The data is provided by the QMUL portal for the development of this work. Although the original MNIST dataset (a subset of the NIST database, created by Yann LeCunn and others) contained 60,000 training images, 10,000 test images of 28*28 resolution, the dataset this work is based on has 41,000 images for training and testing of the same 28*28 resolution.

2.1 Data Preprocessing

The MNIST data comes fairly preprocessed and cleaned in CSV format (comma separated values). Each row is an instance of the sample (one digit) - each column is the pixel density of that instance. There are a total 728 pixels and their corresponding values ranging to 0 - 255. In this section a brief overview of the data processing techniques used in this work are presented.

1. Shuffling: After the data is loaded as a pandas dataframe, it is converted into a numpy n-dimensional array and shuffled randomly so as to make sure the training and the cross validation sets are fairly mixed with no single instance occurring too many times in either set.

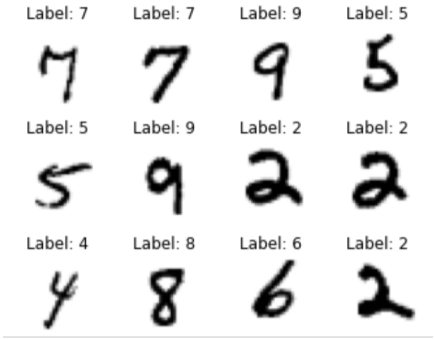
2. Train / Test sets: The dataset is split into training and cross validation sets - the ML models will be trained on 80% of the data, and then its performance will be tested on the CV set with 20% unseen data.

3. Normalization: The pixel values of each digit range from 0-255. This wide range of values can become a problem when calculating gradients during back-propagation. Each pixel value is divided by 255 to down-scale the values to stay between 0 - 1.

4. Transposition - The train set vectors are transposed to feed into the neural network model - this makes every row a pixel value and every column corresponding to an instance of the data (in this case - one digit).

5. One hot encoding - The labels are one hot encoded - every digit has its own vector of 10 classes where the index corresponding to the digit is 1, and 0 elsewhere.

The following image illustrates the dataset that the model is trained on alongside their corresponding labels on top.



After the dataset is processed, it is now ready to be fed into a neural network. The final dimensions of train set and test set vectors are presented below:

X_train shape: (784, 33600)
Y_train shape: (33600,)
X_val shape: (784, 8400)
Y_val shape: (8400,)

X dimensions are - (pixels, examples) and y (labels) values are - (example,)

Chapter 3

The Neural Network

3.1 Theory behind neural networks

3.1.1 Introduction

The subset of machine learning that deals with training deep neural networks is termed as deep learning. This section presents a high level introduction of what neural networks are, how they are trained and hyperparameter optimization.

Neural networks are an interconnected layer of neurons(or units) designed to map inputs from one vector space into another vector space, the output. A typical ANN has three types of layers, the input layer, where the data is fed, the output layer and the hidden layer between them. Neurons of one layer are connected to every neuron in its preceding and successive layers. The connections are defined by a real number known as the weight coefficient. The weight coefficient is determined based on the importance of that particular connection. Optimizing the weight coefficients is termed as the learning of the ANN.

3.1.2 Forward Propagation

During training each neuron calculates the weighted sum of its input and a bias term with the following equation:

$$Y = W.X + b$$

where y is the output of the neuron, w is the weight of the connection connected to it , and b is the bias term. This output is then passed through an activation function to calculate the output for the neuron in the next layer.

$$z = \text{ReLU}(Y)$$
$$\text{where ReLU}(g) = \max(0, g)$$

Activation functions are used to introduce non-linearities in the network. There are different kinds of activation functions for eg - Linear, ReLU, leaky ReLU, sigmoid, logistic sigmoid, the application of which depends on the complexity of the output, as some activation functions make training

faster and some more accurate. For the purposes of the concerned problem, the softmax and ReLU functions will be used. The final output of this process is then applied to an optimisation function or error function, commonly known as cross-entropy loss shown in equation 6. This calculates the difference between the ANN output with the current weights and biases to the training labels and tells how far off are the predictions from the ground truth. This is known as the forward propagation step in training. The cross-entropy loss for our classification is show below:

$$cross - entropy = -\frac{1}{N}\sum_{i=1}^{N}\sum_{i=1}^{N} t_{i,j} log(p_{i,j})$$

where N is the number of samples, k is the number of classes, log is the natural logarithm, t_i,j is 1 if sample i is in class j, 0 otherwise. P_i,j is the probability that sample i belongs in class j.

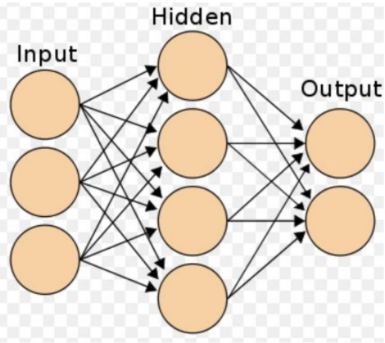
3.1.3 Backward propagation

Arguably, the back-propagation algorithm forms the core of the neural network. In this stage, starting from the outermost layer, the gradients of the loss with respect to the weights and biases are calculated, and subsequently adjusted with the knowledge of the direction the parameters should propagate to reach the global optima.

$$w_{ij}^{k+1} = w_{ij}^{k} - \alpha(\frac{\delta E}{\delta w_{ij}})^{k}$$

$$b_{i}^{k+1} = b_{i}^{k} - \alpha(\frac{\delta E}{\delta bi})^{k}$$

Where w_i,j is the weight between neurons i and j, alpha is the learning rate, b_i is the bias of the i^{th} neuron, k is the iteration number, E is the loss computed from the cross-entropy loss function mentioned in the previous function. The diagram below illustrates a neural network.



3.2 Training and Validation

In this work, the weights and biases of the ANN are initialized randomly from an uniform distribution and scaled. The 'init_params' function takes as inputs the network architecture and returns

the parameters in a python dictionary in the format: W1:..., b1. . . , Wn:..., bn:.... Following this, all the activation functions are defined - in this case, ReLU and softmax. ReLU returns the maximum between the computed value and 0, effectively eliminating any values lesser than or equal to 0. The softmax function on the other hand returns the probabilities for each of the classes signifying the likelihood that a given input belongs to a certain class. All of the probabilities sum to one at the output.

The forward propagation function takes the X values and params dictionary (returned by the init_params function) and returns the activations dictionary to be used by the back propagation function. The input layer activations are the input features itself, and the subsequent layer activations are calculated by dot product of weights and features with the bias added. For all the layers except the outer layer, the activations are calculated with ReLU output. For the outer layer softmax is used.

Following the forward propagation, the loss is calculated with the cross entropy loss function - which returns the average loss across all the examples in the dataset, which makes way for the next step of the training - calculating the gradients. The aim of this algorithm is to tune the parameters in such a way that the loss is minimized at the end of training.

The back propagation function takes the labels (Y), params and activations dictionary, calculates the derivatives and subsequently the gradients of all the layers: starting from the outer layer and returning a gradients dictionary. In line with the gradient descent algorithm; the neural network parameters are updated proportional to the learning rate (set manually) and the gradients calculated by the back prop function.

The gradient descent function implements the gradient descent optimization algorithm by calling the helper functions defined above. It runs a for loop for the number of iterations provided, and in the end plots a graph of the loss and accuracy as they changed during the training of the model.

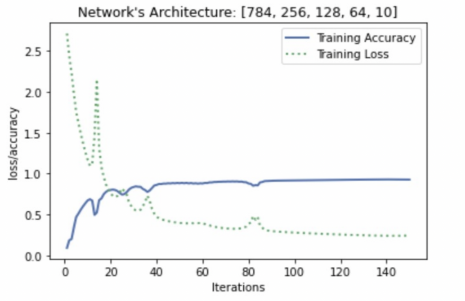
4 models of varying architectures are tested on the data, number of iterations are kept constant at 150, and the provided learning rate is 0.1. A larger learning rate may lead to overshoot leading to failure in convergence while a smaller rate will take too long to converge, thus a middle ground had to be reached. Deciding the number of iterations was also an empirical one - too less would lead to underfitting, and too much would overfit the model - it would not generalize well on unseen data. The below figure illustrates these figures:

```
layer_architecture = [[784, 256, 128, 64, 10],
                      [784, 128, 32, 10],
                      [784, 128, 10],
                      [784, 32, 10]]
iterations = 150
learning_rate = 0.1
```

Chapter 4

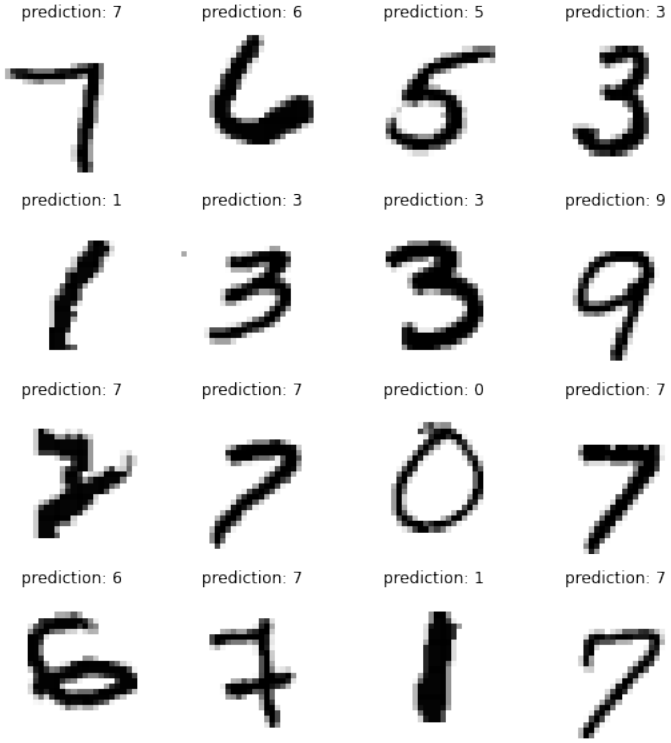
Results & Conclusion

After training the four models for 150 iterations, the highest test accuracy of 92% was achieved by the model of architecture [784, 256, 128, 64, 10]. This model achieved a train set accuracy of 92.7% indicating that it generalized well on the cross validation dataset. The lowest performing model had much simpler architecture with only one hidden layer of 32 neurons. It got a test set accuracy of 83% (all model performances are shown in the accompanying notebook). The loss/accuracy v iterations graph of the best performing model is shown below:



The learning plateaued at around 50th iteration, with a brief bump at the 80th iteration - but flattening out again after that.
The validation set images and the corresponding model predictions of the first 16 images of the dataset are presented below:

The following figure shows 16 images and their corresponding predictions by the model. At 92% accuracy it got all digits correctly classified.



END