U N I V E R S I T Y O F C A P E T O W N

Department of Electrical Engineering

CSC3022F - Machine Learning Assignment 3

Ian Edwards - EDWIAN004

Plagiarism Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the Harvard convention for citation and referencing. Each contribution to, and quotation in, this final year project report from the work(s) of other people, has been attributed and has been cited and referenced.
3. I have not allowed and will not allow anyone to copy my work with the intention of passing it off as their own work or part thereof.

Table of Contents

Part 1 - XOR Gate

The following section details a multi-layer perceptron solution to implementing a XOR Gate.

1.1 Introduction

Single Perceptrons are only capable of solving linearly separable problems. The XOR Gate is not linearly separable and thus cannot be implemented by a single perceptron. However, since primitive gates such as AND, NOT and OR gates are linearly separable, one can chain together various perceptrons to implement a XOR gate. The following diagram shows how a XOR Gate is not linearly separable:

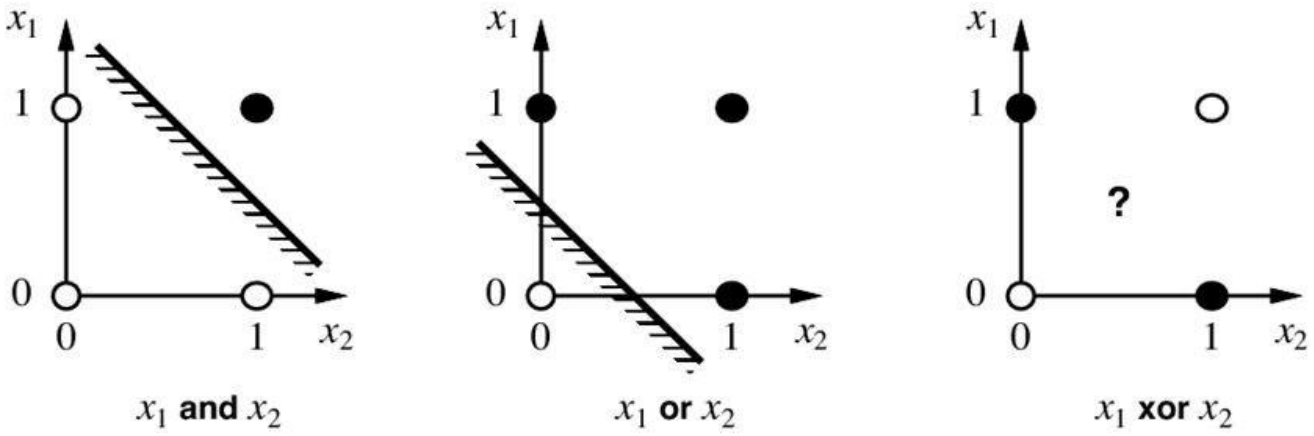

Figure 1: Diagram of gates and linear separability (Chandradevan, 2017)

1.2 Network Topology

Inputs > 0.75 will be treated as an on signal otherwise a 0 is interpreted. A XOR Gate can be represented with boolean logic as: AB' + A'B. This can be represented alternatively as such:
AB' + A'B
= AB' + A'B + AA' + BB'
= A(A' + B') + B(A' + B')
= (A' + B')(A + B)
= (AB)' (A + B) ← Can work with these logic gates or the next line.
= (AB)' (A' B')'

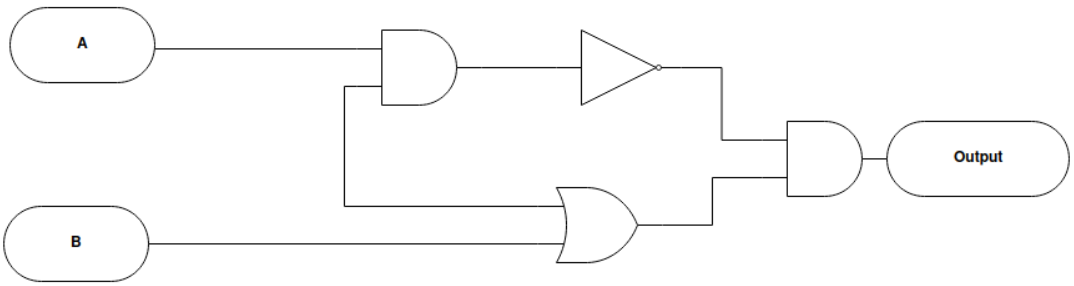

Figure 2: XOR Gate representation as (AB)' (A + B)

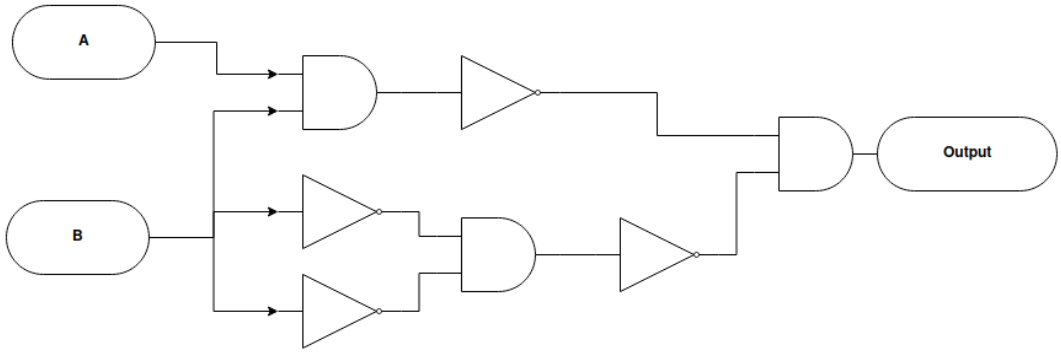

Figure 3: XOR Gate representation as (AB)' (A' B')'

Given the following boolean logic, Figure 2 was first implemented by simply training 3 individual perceptrons - an AND, NOT and OR perceptron. The individual perceptrons were trained with generated data and the inputs then evaluated by evaluating each perceptron in turn:
E.g. X1 = 1, X2 = 0 → AND(NOT(AND(X1, X2)), NOT(AND(NOT(A), NOT(B))) → using Figure 3.
Figure 3's configuration was chosen as it meant less training, which could take a while to reach the accuracy metric of 98%. This way only AND and NOT perceptrons needed to be trained. Furthermore, the OR perceptron approach, as shown in Figure 2, was attempted but was unable to converge hence FIgure 3's approach was followed - which converged for all training attempts.

1.3 Training Data, data generation and learning rates

The data used to train the AND and NOT perceptrons was randomly generated for each perceptron using random.uniform(-0.25, 1.25) (to account for noise). 300 training samples and 100 validation samples were generated for each perceptron. A 3-way holdout method was used to generate and evaluate the data so as to prevent information leak with training not happening on validation or test set data. Validation data is used to evaluate the model each training epoch and generate an accuracy metric - this has no effect on the model, only that training continues until this accuracy metric is maximized above 98%. If the training got stuck in an infinite training loop switching from one accuracy value to another never converging, the training was stopped, and another data set was generated and training was restarted. This was found to be effective to avoid getting stuck in local minima and rather achieve higher accuracies.

It was found that if learning rates were too high (generally above 0.2) then it took longer to get desirable accuracies if it did at all - the model would jump weights around too much, never converging. Too small learning rates did not produce desirable weight values in the training time given (0.0001). Thus a training rate of 0.001 was chosen for all perceptrons.

2. Part 2: Image Classification with ANN

2.1 Introduction

Part 2 concerns itself with classifying handwritten digits from the MNIST10 dataset using an Artifical Neural Network (ANN). An ANN will be developed with PyTorch and Torchvision, implementing a fully connected multi-layer neural network. The MNIST dataset has 60000 training images and 10000 test images available, which can be loaded into tensors and run through a built ANN model to train it. In MLPs threshold activation functions are used - but in ANNs one would use continuous activation functions such as a ReLU function which are differentiable (unlike a threshold function). Differentiability is needed for stochastic gradient descent in the ANN. The use of a continuous activation function with at least one hidden layer in the network makes the network a universal function approximator.

2.2 Network Topology and preprocessing

The network makes use of 4 layers:

- Input layer: Input features = 784 out features = 128
- Hidden layer 1: Input features = 128 out features = 64
- Hidden layer 2: Input features = 64 out features = 10
- Output layer: 10 input features outputting the classification.

Reasoning for architecture:
A **fully connected feedforward** ANN is used to solve this problem as opposed to the commonly used CNN approach. This is because the use of a feed-forward ANN here is easy to implement, and is lighter weight. They offer relatively good accuracy performance albeit require more time to train given how it requires flattening 2D input out increasing the number of features dramatically.

Images read in from the MNIST data set are 28 x 28 - this was investigated with some custom functions written in Classifier.py such as investigate_data(). Thus to feed the data into the feedforward ANN, data must be one dimensional, so 28x28 images are flattened out producing 784 neurons in the input layer. This is done with “images.view(images.shape[0], -1)” in code, where images are tensors of 28x28 images. The output layer must be 10 neurons so as to classify 10 different digits. Thus the hidden layers must be between these sizes. If there are too few neurons one ends up underfitting, if too many one overfits. Using 2 hidden layers one is able to approximate any smooth mapping with high accuracy and an arbitrary decision boundary can be represented with arbitrary accuracy (Fedevych, 2017). Hence two hidden layers were chosen, of arbitrary decreasing size - 128 and 64 respectively. Preprocessing done on the data includes making data able to be usable and seen by the model. Thus transforming data into RGB channels between 0 and 255 scaled to 0 to 1 range converted to tensors. The data is ensured to be the same size normally with transforms.resize() but this wasn't necessary as all inputs were 28x28. The mean and standard deviation across the tensors is set and the tensors are outputted. The transformed data is wrapped in a data loader to allow one to iterate through the dataset in batch sizes of 64 as set in the code where during training, batches of data are passed into the model and evaluated. At the start of each epoch, the data loader reshuffles the data to reduce overfitting.

A rectified linear unit (ReLU) activation function is placed after hidden layer 1, followed by another after Hidden layer 2 followed by a LogSoftmax function after the output layer. A Softmax activation function is

useful for classification problems such as this because output vector values are converted to probabilities which can be used to choose the correct classification (Brownlee, 2020). ReLU functions are used because they are easy to train with and offer good performance.

2.3 Loss function, optimizer, training and validation

The loss function used was a cross correlation loss function because it is typically the default loss function used for multi-class classification scenarios (Brownlee, 2019). The more a value deviates from its ground truth the higher the penalty or loss it receives. A good ideal cross entropy value is 0. An Adam optimizer is used which applies gradient descent in the ANN to reduce loss and improve accuracy - this is a commonly used good optimizer that is good at avoiding getting stuck in local minima during gradient descent.

The network is thus trained by iterating through a number of epochs (default 10) and during each epoch the torch.nn.model.train() function is called to set the model for training and not evaluating. Tensors of images and their corresponding labels are loaded from the training set data loader, the images are flattened, the optimizer gradient is set to 0 for every image inputted, and a forward pass is conducted on each input image in the batch ("model(images)" in code.) The loss is calculated ("criterion(output, labels)") the gradients of all variables are calculated with respect to the loss ("loss.backward()") and then updates are applied with the Adam optimizer: "optimizer.step()". The overall loss is summed and averaged over a single epoch, per epoch. The same is done with the accuracy of the model for an epoch. Thereafter, plots can be generated of the model's loss and accuracy over the number of training epochs. Similarly during each training epoch, the model is validated against the validation dataset, and validation loss and validation accuracy is recorded and plotted as well - to be plotted over the training loss and training accuracy to give an idea of how well the model can generalize.

2.4 Additional Notes:

Additional functions like "plotRecordedData()" and "checkImage()" are used which plot the historical data regarding the data set, and predict a given image from a given file path, respectively.

A three-way holdout method for sampling was used in which the full training dataset is split into 50000 training samples and 10000 validation samples. The test dataset is held till the end on which testing is conducted to get an idea of how well the model generalizes. Furthermore, the model is able to be saved and loaded - simply uncomment line 58 to save the model after training - improve how quickly one can run the program. If the program detects a saved model in the directory, it will be loaded and retraining is **not** done.

See Figure 4 on the next page showing the Cross Entropy Loss and Classification Accuracy for training and validation. Training is in blue and validation is in red. It shows that whilst training accuracy finishes very high at almost 98% with negligible loss, when seeing how well it generalizes on a validation set, the accuracy is found to be slightly lower and loss slightly higher - clearly the training set values are slightly optimistic. The test accuracy was found to be around 97.5% which is acceptable.

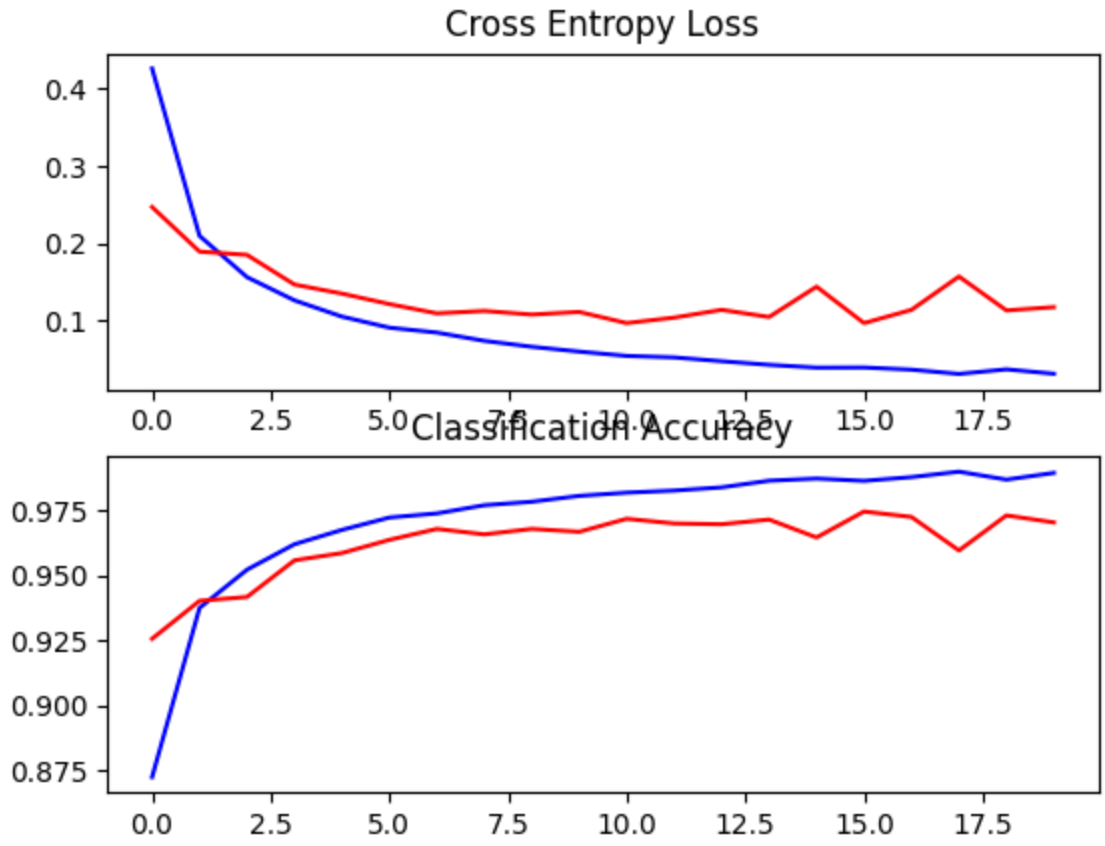

Figure 4: Training and validation set Cross Entropy Loss and Accuracy during training epochs

3. References

- Chandradevan, R., 2017. *Radial Basis Functions Neural Networks — All we need to know*. [online] Medium. Available at: <https://towardsdatascience.com/radial-basis-functions-neural-networks-all-we-need-to-know-9a88cc053448> [Accessed 13 June 2021].
- Fedevych, Y., 2017. *From "Introduction to Neural Networks for Java, Second Edition"*. [online] Medium. Available at: <https://medium.com/@gk_/from-introduction-to-neural-networks-for-java-second-edition-eb9a833d568c> [Accessed 13 June 2021].
- Brownlee, J., 2020. *Softmax Activation Function with Python*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/softmax-activation-function-with-python/> [Accessed 13 June 2021].
- Brownlee, J., 2019. *How to Choose Loss Functions When Training Deep Learning Neural Networks*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/> [Accessed 13 June 2021].