Artificial Neural Networks
The XOR Problem

—

University of Cape Town

Department of Computer Science

14 June 2021

CSC3022F

Peter Mhlanga **MHLPET015**

Introduction

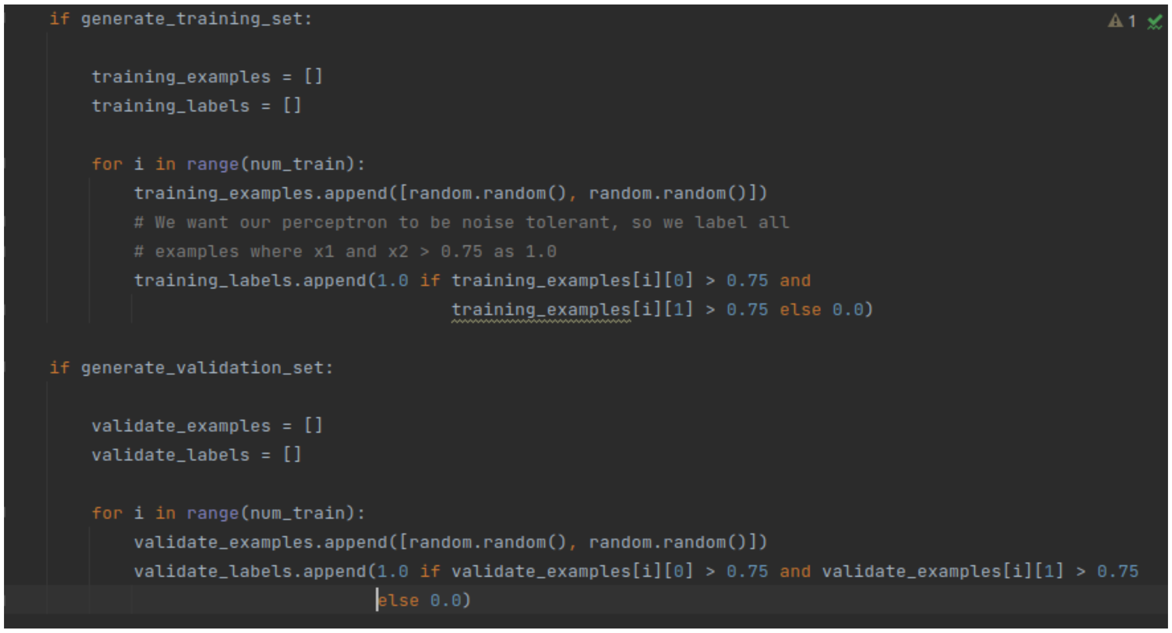
The task was to solve the XOR problem using only Perceptrons. Single Perceptrons are only capable of solving linearly separable problems. The XOR Gate is not linearly separable but can be implemented by chaining together individual Perceptrons. Hence to solve the XOR problem we could use only Perceptrons that implement OR, AND or NOT gates. These Perceptrons would use a threshold activation function and the final network was to be noise tolerant. According to the specifications, any input > 0.75 would be treated as an on signal and the network would be able to detect that.

Design

The first part of the design process was to design the simple gates that could be fully implemented using a single Perceptron. For this we started by designing the AND, NOT and OR gates. After writing the code that implemented the functionality of a single perceptron, the next task was to write code that could train that Perceptron to behave as an AND, NOT and OR gate. For this, we generated a training set (list) and validation set (list) for each gate. The Perceptron would need to use the training sets to train and the validation sets were used to check the performance of the perceptrons after training them. This would allow us to determine the optimum values of **number of training examples**, **learning rate** and **bias** to use for training each of the gates.

1. Generation of Training Data

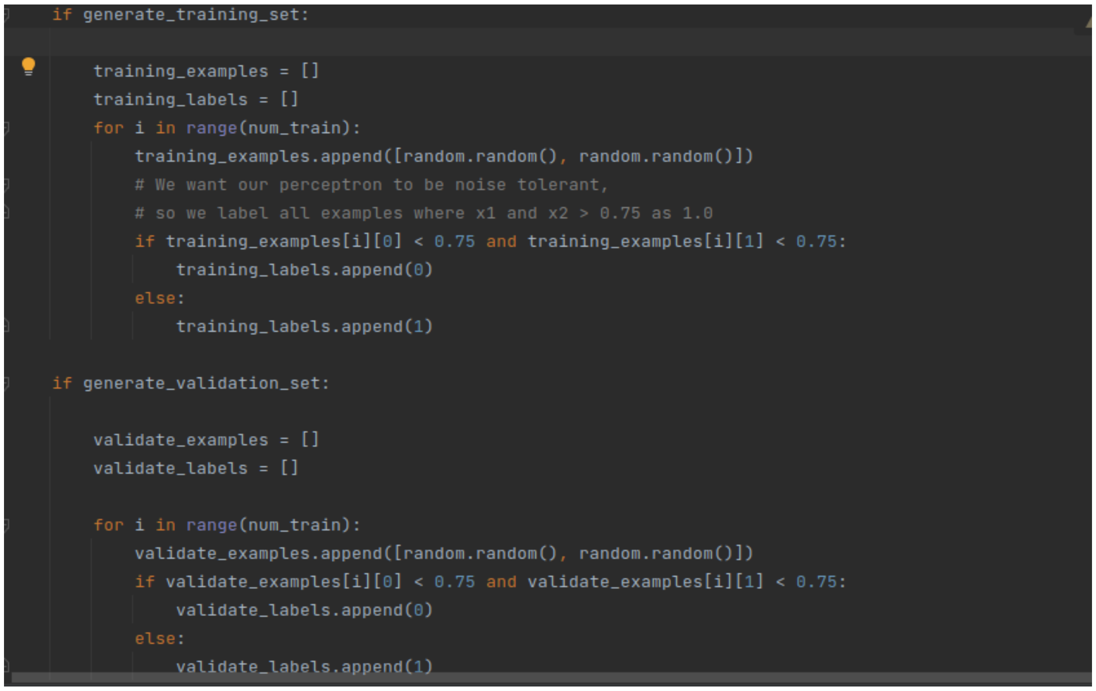
For the AND gate, we created two sets: One for training the gate and one for validating the set performance. The number of training examples was set to be 1000 which was determined to be adequate for training it after a couple of tests. The selection of training examples was randomised so as to reduce/eliminate bias that might be caused by using a particular set of training examples. The code used is as shown below:



```
if generate_training_set:

    training_examples = []
    training_labels = []

    for i in range(num_train):
        training_examples.append([random.random(), random.random()])
        # We want our perceptron to be noise tolerant, so we label all
        # examples where x1 and x2 > 0.75 as 1.0
        training_labels.append(1.0 if training_examples[i][0] > 0.75 and
                               training_examples[i][1] > 0.75 else 0.0)

if generate_validation_set:

    validate_examples = []
    validate_labels = []

    for i in range(num_train):
        validate_examples.append([random.random(), random.random()])
        validate_labels.append(1.0 if validate_examples[i][0] > 0.75 and validate_examples[i][1] > 0.75
                     else 0.0)
```

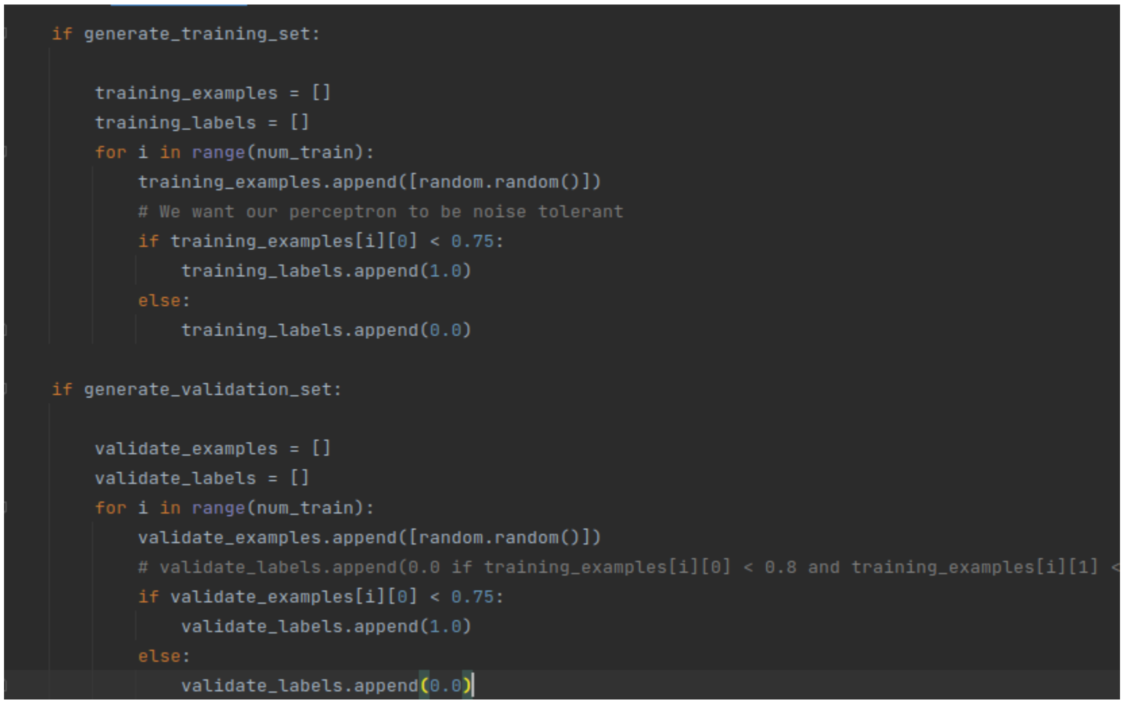
The training examples were generated using the **random()** function in the random library which returns a number between 0 and 1. In the case of the AND gate as shown in the code above, the labels would be appended onto the training labels list such that the output resembles that of an AND gate with a threshold of 0.75 to account for noise tolerance.

The OR Gate was also created from the Perceptron using the same approach. In the case of the OR gate the labels were appended to the training_labels list such that its output resembled that of an OR gate with a 0.75 threshold to account for noise as shown in the following code snippet:



```python
if generate_training_set:

    training_examples = []
    training_labels = []
    for i in range(num_train):
        training_examples.append([random.random(), random.random()])
        # We want our perceptron to be noise tolerant,
        # so we label all examples where x1 and x2 > 0.75 as 1.0
        if training_examples[i][0] < 0.75 and training_examples[i][1] < 0.75:
            training_labels.append(0)
        else:
            training_labels.append(1)

if generate_validation_set:

    validate_examples = []
    validate_labels = []

    for i in range(num_train):
        validate_examples.append([random.random(), random.random()])
        if validate_examples[i][0] < 0.75 and validate_examples[i][1] < 0.75:
            validate_labels.append(0)
        else:
            validate_labels.append(1)
```

In the case of the NOT Gate, the examples would be a list of (n x 1) dimensions since it takes only one input. the The training outputs were appended to the training_labels list such that its output resembled that of an NOT gate with a 0.75 threshold to account for noise as shown in the following code snippet:



```python
if generate_training_set:

    training_examples = []
    training_labels = []
    for i in range(num_train):
        training_examples.append([random.random()])
        # We want our perceptron to be noise tolerant
        if training_examples[i][0] < 0.75:
            training_labels.append(1.0)
        else:
            training_labels.append(0.0)

if generate_validation_set:

    validate_examples = []
    validate_labels = []
    for i in range(num_train):
        validate_examples.append([random.random()])
        # validate_labels.append(0.0 if training_examples[i][0] < 0.8 and training_examples[i][1] <
        if validate_examples[i][0] < 0.75:
            validate_labels.append(1.0)
        else:
            validate_labels.append(0.0)
```

2. Construction of the XOR Gate

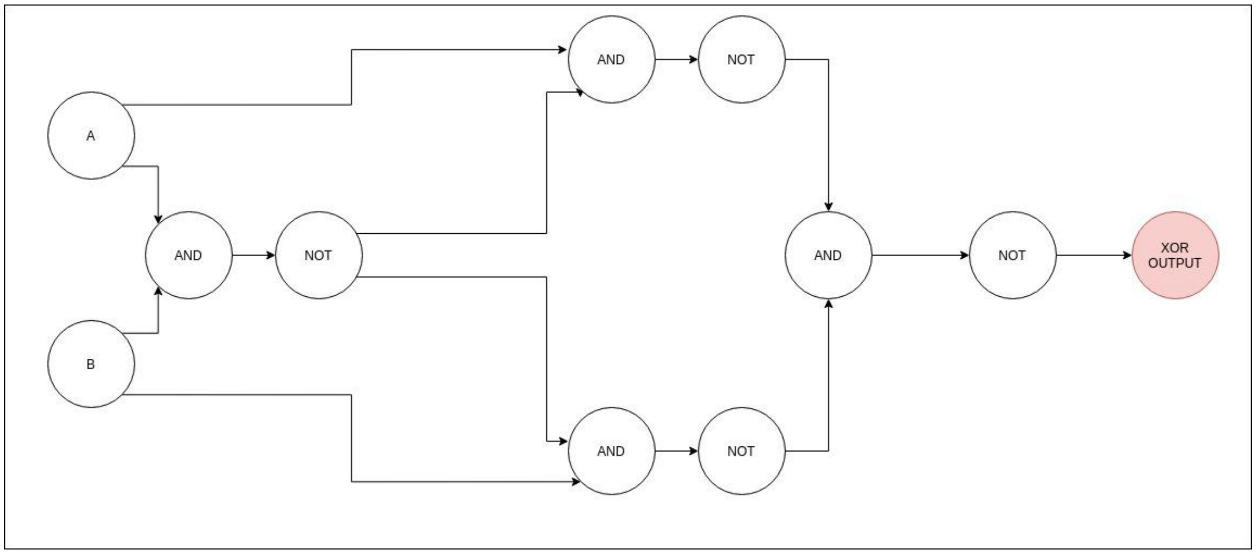
After creating and training the OR, AND and NOT gates, we trained them. During the training process we determined through trial and error the best values of Bias, Accuracy and learning rate for the three gates which are shown below:

NOT Gate: Bias = 1.0 Learning rate = 0.1 Highest Achieved Accuracy=100%

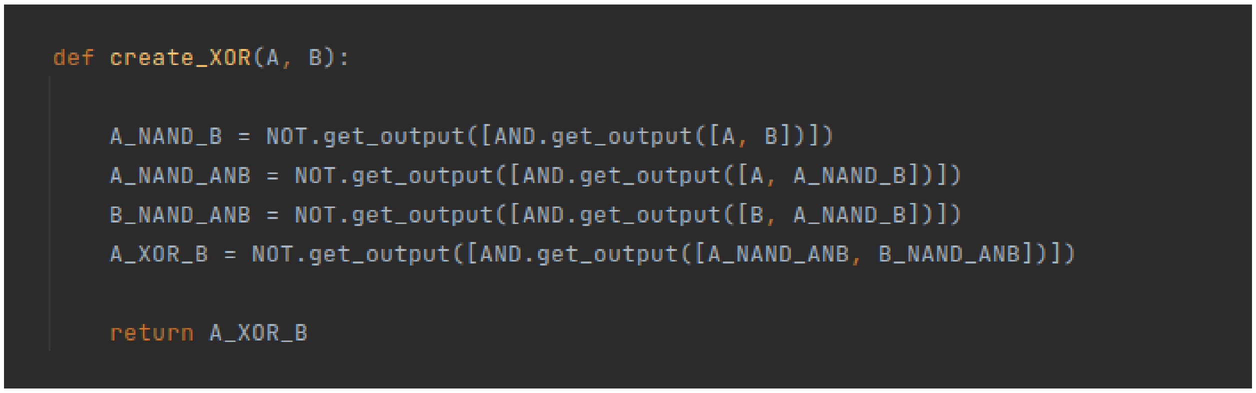
AND Gate: Bias = -1.5 Learning rate = 0.20 Highest Achieved Accuracy=99%

OR Gate: Bias =-1.55 Learning rate = 0.1 Highest Achieved Accuracy=88%

Given that the OR Gate was highly inaccurate compared to other gates, we had to make a design decision to build the XOR Gate using only NAND Gates, which are a combination of AND and NOT gates. The resulting topology of the network is as shown in the diagram below:



In the diagram above, A and B represent our inputs to the XOR gates, AND and NOT nodes represent our Perceptrons and XOR OUTPUT is the output node of our Network. The following code snippet shows the function that created thes network:



```python
def create_XOR(A, B):

    A_NAND_B = NOT.get_output([AND.get_output([A, B])])
    A_NAND_ANB = NOT.get_output([AND.get_output([A, A_NAND_B])])
    B_NAND_ANB = NOT.get_output([AND.get_output([B, A_NAND_B])])
    A_XOR_B = NOT.get_output([AND.get_output([A_NAND_ANB, B_NAND_ANB])])

    return A_XOR_B
```

3. **How different learning rates affect the accuracy of the Network**

Observations showed that for all the gates; the accuracy reduced with increase in learning rate. The higher the learning rate was, the longer the time the perceptrons needed to train so as to get 98% accuracy. The OR gate was still the worst performing in the training. To specify, the gates would not be able to train to 98% accuracy if the learning rate was anything higher than 0.65.

Conclusion

The overall performance of the system was successful after the decision to build it using only NAND gates was used. Most of the results during the XOR testing returned expected values even when it was given noisy data. As a result the experiment was a success.

Artificial Neural Networks
Image Classification

University of Cape Town

Department of Computer Science

14 June 2021

CSC3022F

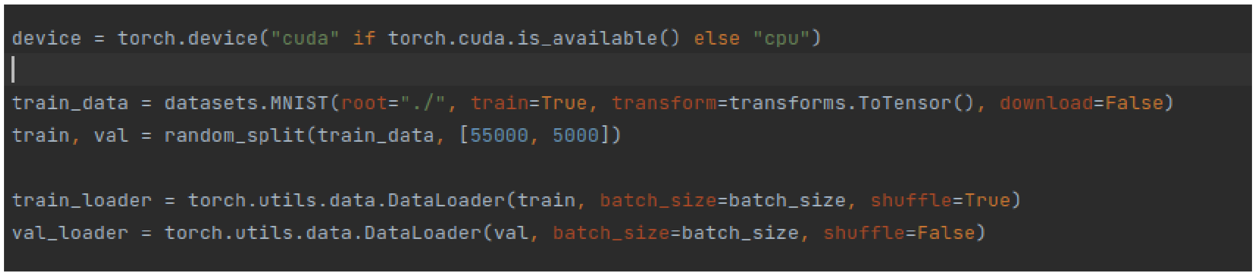
Peter Mhlanga **MHLPET015**

Introduction

The task was to design an ANN that is capable of classifying handwritten digits from the MNIST10 dataset.

Design

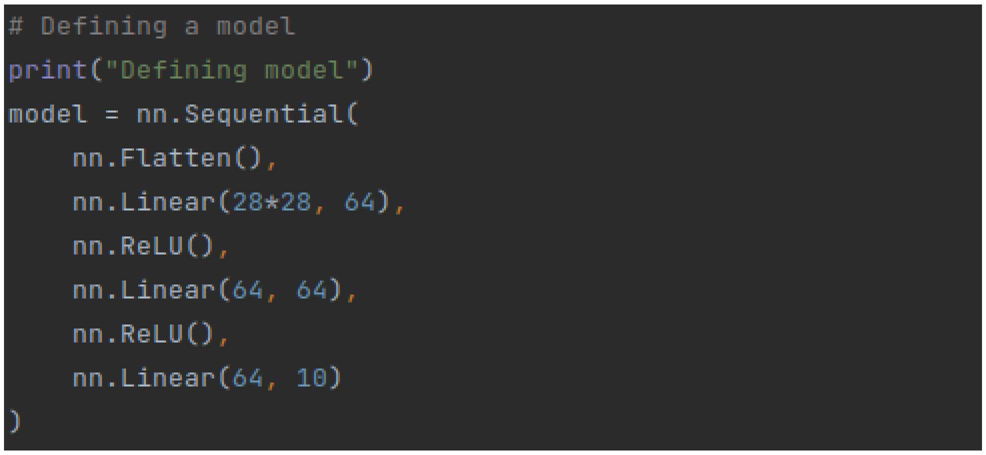
The first part of the design process was to import the data to be used in training and validating the model. We were going to run all the code on the GPU because unlike the CPU it could do tasks in parallel. So we needed to initialise a variable (device to help us do this. The code which did this is shown in the picture below:



```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

train_data = datasets.MNIST(root="./", train=True, transform=transforms.ToTensor(), download=False)
train, val = random_split(train_data, [55000, 5000])

train_loader = torch.utils.data.DataLoader(train, batch_size=batch_size, shuffle=True)
val_loader = torch.utils.data.DataLoader(val, batch_size=batch_size, shuffle=False)
```

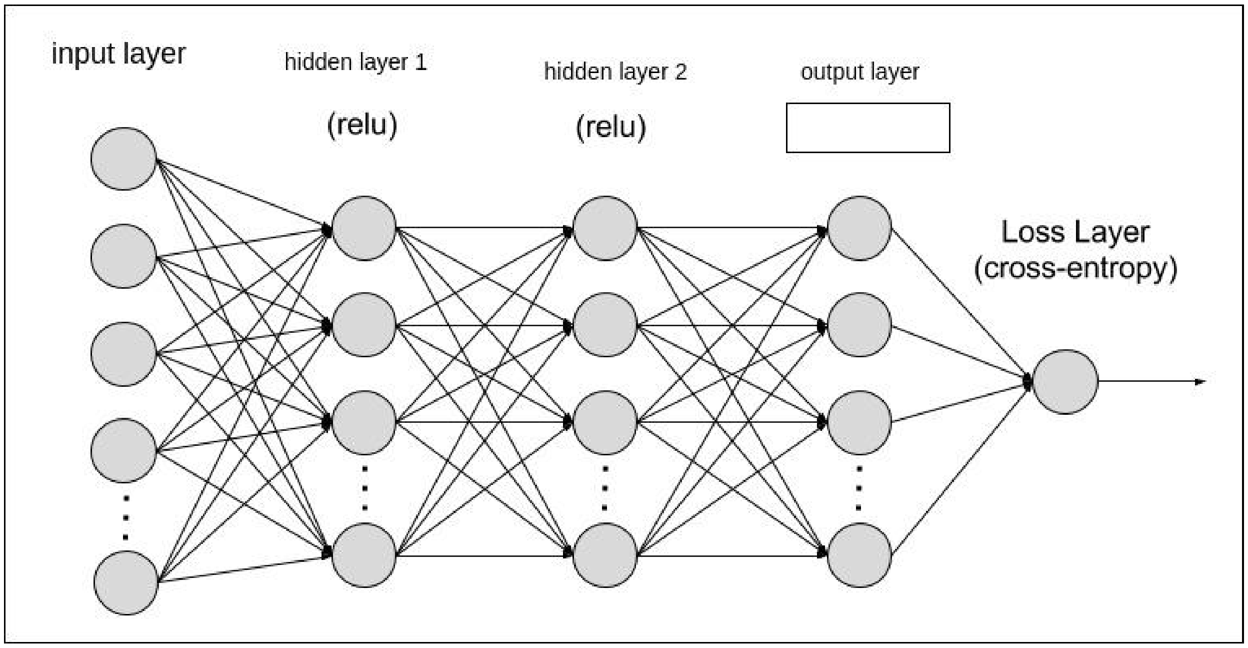
1. Defining of Model

The Sequential model was chosen because it is a simple model that allows us to build deep neural networks by stacking layers one on top of another. The output of one level is used in the next level of the neural network. Whilst classifying the images, we want the network to take the image and break it down into different components in the next level that depend on the previous level components which is why the Sequential model was the best model to be used.



```
# Defining a model
print("Defining model")
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28*28, 64),
    nn.ReLU(),
    nn.Linear(64, 64),
    nn.ReLU(),
    nn.Linear(64, 10)
)
```

The structure of the topology of the network is show in the image below:



The input layer is defined with 784 input features and 64 output features.

The hidden layer 1 is defined with 64 input features and 64 output features.

The hidden layer 2 is defined with 64 input features and 10 output features that represent our output. The outputs from the output layer are taken and used to calculate the total cross entropy loss in the Loss Layer.

2. Selection of Activation Function

In both hidden layers , we used the ReLu(Rectified Linear Unit) activation function. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.

This means that the neurons will only be deactivated if the output of the linear transformation is less than 0. For the negative input values, the result is zero, that means the neuron does not get activated. Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh function. Hence the use of the ReLu function in our model.

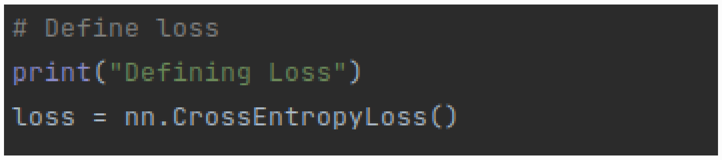
3. Pre-Processing steps performed on the training set.

The model is being trained with densely-connected layers (i.e. Linear layers). The problem with these layers is that they only accept 1D data. To cater for this, we added a Flatten() layer as the first layer, which essentially flattened the multidimensional data into 1 Dimension so that our network would be able to use it. This was our pre-processing step on the training set.

2. Defining Loss function

The goal of the experiment was to reduce the difference between the predicted output and the actual output. This is determined by the Loss function. The Cross Entropy Loss function was used. This is because we wanted our model to classify our images and return the class which a particular image belongs to using the highest probability. We know that with this loss function, as the predicted probability approaches 1, log loss slowly decreases. As the predicted probability decreases, however, the log loss increases rapidly. Log loss penalizes both types of errors, but especially those predictions that are confident and wrong so we would get a good

prediction on the correct ones and the wrong images for each particular class/label of the images. The following image shows how we defined the loss function in code:



```
# Define loss
print("Defining Loss")
loss = nn.CrossEntropyLoss()
```

3. **Selection of Optimizer**

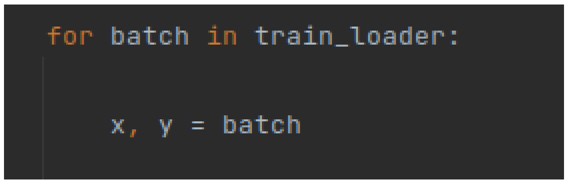
We used the Stochastic Gradient Descent optimiser. We chose this over the Gradient Descent because in the SGD algorithm,the derivative is computed taking one point at a time unlike using the whole set which leads to a faster computation of our loss gradient. This was not necessary but using it increased computation time.

4. **Training and Validation of Network**

Training

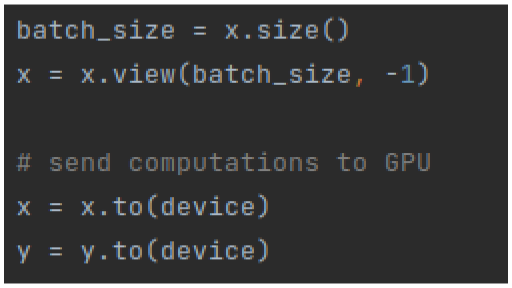
After defining all the necessary variables and functions, the next step was to train the model.We will present this as steps; NB: All computations of the images during training and validation were sent to the GPU of the computer to allow for faster computations.

1. Obtain the training batch of images and labels from the training loader.



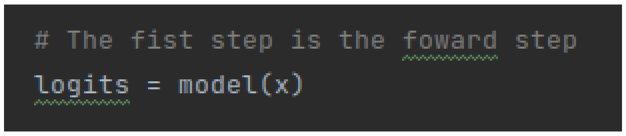
```
for batch in train_loader:

    x, y = batch
```

2. Reshape all the images by converting them from an image (28x28)pixels into a vector and allow processing via GPU.



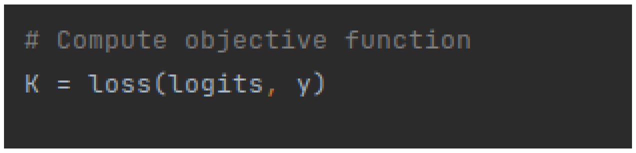
```
batch_size = x.size()
x = x.view(batch_size, -1)

# send computations to GPU
x = x.to(device)
y = y.to(device)
```

3. Carry out the forward propagation where we compute the logits.



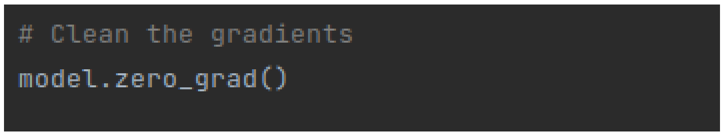
```
# The fist step is the foward step
logits = model(x)
```

4. Calculate the loss using the labels and results from the forward propagation.



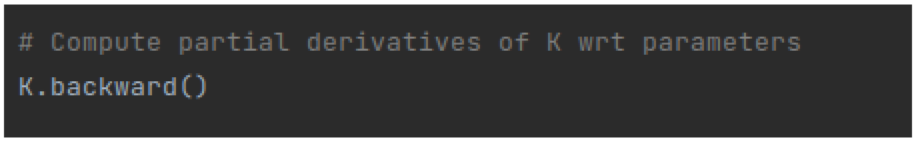
```
# Compute objective function
K = loss(logits, y)
```

5. Reset the gradients to zero



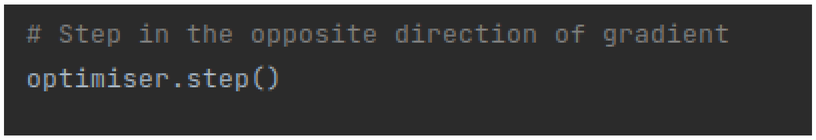
```
# Clean the gradients
model.zero_grad()
```

6. Carry out backward propagation where we calculate the partial derivatives of the loss with respect to the parameters and accumulate these new partial derivatives of the objective function.



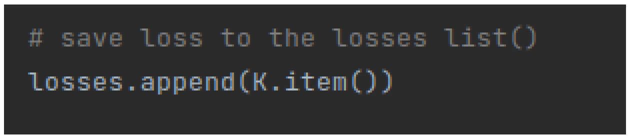
```
# Compute partial derivatives of K wrt parameters
K.backward()
```

7. Use the optimiser in order to step in the opposite direction of the gradient.



```
# Step in the opposite direction of gradient
optimiser.step()
```

8. Compute Losses and save them to a list



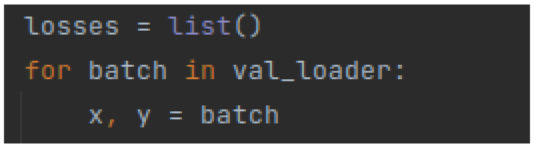
```
# save loss to the losses list()
losses.append(K.item())
```

9. Repeat Step 1 to 8 until training losses reach a desired minimum.

Validation

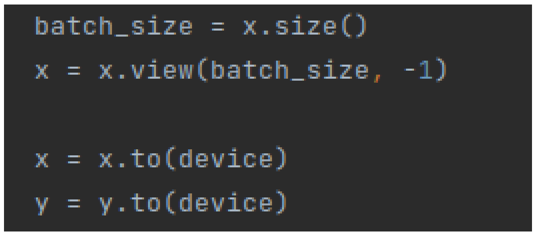
Validation was run in parallel to training so as to evaluate the progress being made after every training epoch. In this case we were using validation data instead of training data. The steps are outlined below;

10. Obtain the training batch of images and labels from the validation loader.



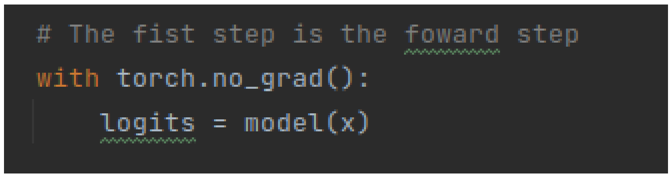
```
losses = list()
for batch in val_loader:
    x, y = batch
```

11. Reshape all the images by converting them from an image (28x28)pixels into a vector.



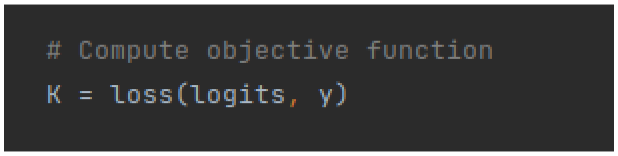
```
batch_size = x.size()
x = x.view(batch_size, -1)

x = x.to(device)
y = y.to(device)
```

12. Carry out the forward propagation where we compute the logits. In this case we specified to torch that it should not keep track of graphs and gradients but compute the final outcome only.



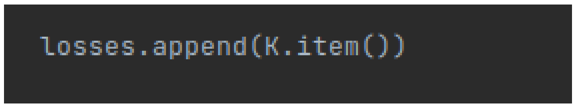
```
# The fist step is the foward step
with torch.no_grad():
    logits = model(x)
```

13. Calculate the loss using the labels and results from the forward propagation.



```
# Compute objective function
K = loss(logits, y)
```

14. Compute Losses



```
losses.append(K.item())
```

15. Repeat Step 10 to 14 for a certain number of epochs

After this we programmed for user input and output to the user.

Conclusion

The experiment was successful because our program was able to take an input as an image and correctly classify it after an adequate number of training episodes.