**ANN PROJECT (Classifying Digits Using MNIST Dataset)**

1.Introduction

The goal of this project is to design an image classifier that takes in an image of a handwritten number and produces a predicted class for that number. This class ideally, correctly identifies the given image. To build this we will use MNIST dataset which contains thousands of small gray scale images of hand-written digits. Each image depicts one of the numbers 0 t0 9. Using deep learning, we can take a data driven approach to training an algorithm that can examine these images and discover patterns that distinguish one number from another.

Our algorithm will need to attain some level of understanding of what makes a hand-drawn 1 look like a 1 and how images of 1s differ from images of say 2s or 3s. The first step in recognizing patterns in images is learning how images are seen by computers.

2.Visualise the data

Any grayscale image is interpreted by a computer as an array. A grid of values for each grid cell is called a *pixel*, and each pixel has a numeric value.

Each image in the MNIST database is 28 pixels high and wide, hence it is understood by computer as a 28x28 array. In a typical grayscale image, white pixels are encoded as the value 255 and black pixels are encoded as zero. Gray pixels fall somewhere in between with light-gray being closer to 255. Also, color images have similar numerical representations for each pixel color.

3.Preprocess the data

These MNIST images have gone through a pre-processing step. They have been re-scaled so that each image has pixel values in a range from 0 to 1 as opposed to from 0-255. To achieve this, every pixel value are divided by 255.

This step is called *normalization* and it is a common practice in many deep learning techniques.

Normalization will help our algorithm to train better. The reason we typically want normalized pixel values is because neural networks rely on gradient calculations. These networks are

trying to learn how important or how weighty certain pixel would be in determining the class of an image. Hence, normalizing these pixel values helps these gradient calculations stay

consistent, and not get so large that they slow down or prevent a network from training.

4.Deciding the classifier

Now to classify these images we can use multi-layer perceptron. Next question is how do we input this image data into an MLP?

We know that MLPs only take vectors as input. So, in order to use an MLP with images, we must first convert any image array into a vector. This process is known as *flattening*.

Example - In case of a 3x3 image we have a matrix with 9 pixel values. Instead of representing this as a 3x3 matrix, we ca construct a vector with 9 entries, where the first 3 entries of our vector correspond to the first row of our old array. The second 3 entries correspond to the second row and so on. After converting images into vector, they can then be fed into the input layer of an MLP.

5.MLP Structure and Class Scores

We will now create a neural network for discovering patterns in our training data. After training, our network should be able to look at totally new images that it hasn't trained on, and classify the digits contained in those images. This previously unseen data is called as test data.

At this point, our images have been converted into vectors with 784 entries (since each image size is a 28x28 matrix). So, the first input layer in our MLP should have 784 nodes.

We also know that we want the output layer to distinguish between ten different digit types, 0 to 9. Hence, we will want the last layer to have 10 nodes. So, our model will take in a flattened image and produce 10 output values, one for each possible class 0 to 9. These output values are often called class scores.

*Class Score* - Indicate how sure the network is that a given output is of a specific class.

A high-class score indicates that a network is very certain that a given input image falls into a certain class. We can imagine that the class score for a handwritten 3, for example,

will have a high score for the class 3 and a low score for the rest of the classes. But it may also have a small score for 8 or any other class that looks similar in shape to a 3.

Now the next question is how many hidden layers we want to include and how many nodes should be in each one of those layers. There is not one correct answer here, but one or two hidden layers should work fine for this simple task.

6.Loss and Optimization

As a network trains, we measure any mistakes that it makes using a *loss function* whose job is to measure the difference between the predicted and true class labels. Then using *back propagation*, we can compute the gradient of the loss with respect to the models' weights. In this way, we quantify how bad a weight is and find out which weights in the network are responsible for any errors. Finally using that calculation, we can choose an optimization function like *gradient descent* to give us a way to calculate a better weight value.

Towards this goal, the first thing we will need to do is make this output layer a bit more interpretable. It is very common to apply a *softmax* activation function to convert these scores into probabilities. To apply a softmax activation function to this output layer, we begin by evaluating the exponential function at each of these scores, then we add up all of these values. Let's denote this sum as 'S'. Then we divide each of these values by the sum. Plugging in the maths, we get 10 values where each value yields the probability that the image depicts its corresponding image class.

Now our objective is to update the weights of the network in response to any mistake (assuming that the hand-written digit is misclassified), so that next time it views the image, it predicts the correct label.

In a perfect world, the network would predict that the image is 100 percent likely to be the true class. In order to get the model's prediction closer to the ground truth, we will need to define

some measure of exactly how far off the model currently is from perfection. We can use a loss function to find any errors between the true image classes and our predicted classes, then backpropagation will find out which model parameters are responsible for those errors.

Since we are constructing a multi-class classifier, we will use categorical cross entropy loss. Let us assume that the model's predicted probability of the true class is 0.155. Cross entropy loss

looks at this probability value (in decimal form) and takes the negative log loss of that value. Hence, in this case loss = -log (.155) = 0.809. In other case, considering that the weights were slightly different, let us assume that the model's predicted probability of the true class is 0.455. This prediction is much better than the former and when we calculate the cross-entropy loss,

loss = -log (.455) = 0.341 we get a much smaller value.

In general, it's possible to show that the categorical cross entropy loss is defined in such a way that the loss is lower when the model's prediction agrees more with the true class label, and it's higher when the prediction and the true class label disagree.

As a model trains, it's goal will be to find the weights that minimize this loss function and therefore give us the most accurate predictions. So, as stated earlier a loss function and

backpropagation gives us a way to quantify how bad a network weight is, based on how close a predicted and the true class label are from one another.

Next, we need a way to calculate a better weight value. Thinking of loss function as a surface that resembles mountain range, to minimize this function, we need to find a way to descend to the lowest valley. This is the role of an optimizer. The standard method for minimizing the loss and optimizing for the best weight value is called *"Gradient Descent"*.