Topology of ANN

My Artificial Neural Network implementation consists of three main components, an input layer, hidden layers, and an output layer. The **input layer** consists of 784 nodes corresponding to the 28 by 28 pixel dimensions of the MNIST dataset images. Each node in the input layer represents a single feature of the model specifically a pixel value ranging from 0 to 255 from the input image which gets normalized in the preprocessing to a value in [0..1].

For the **hidden layers** , I opted for a configuration of two hidden layers each consisting of 64 nodes. The choice of just 2 hidden layers with 64 nodes was motivated by the goal to have a balance between model complexity and performance, I did not want my network to be too deep as deeper architectures with more nodes may offer greater capacity to capture intricate patterns in the data but they also risk overfitting, especially given that the size of the MNIST dataset is relatively modest.[i]

The **activation function** used in both layers is the ReLU (Rectified Linear Unit) to introduce non-linearity and enable the network to learn complex patterns in the input data. I opted for ReLU because it is simple and effective in combating a vanishing gradient problem during training.[ii]

To further enhance the model robustness and mitigate overfitting, I applied dropout regularization with rate of 0.2 after each hidden layer. This deactivates 20% of the neurons during each training iteration preventing the network from being overly reliant on specific features or neurons. This ensures that the resulting model is resistant to noise in each input and ensures it does not memorize training data.

The **output layer** consists of 10 nodes , each node corresponds to the 10 possible classes which are digits 0 to 9 in the MNIST dataset. I applied a **SoftMax** activation function to the output layer to convert the output scores into probabilities to enable multi-class classification.

Preprocessing steps

Before training the Artificial Neural Network on the dataset, a preprocessing step was applied to normalize the pixel values of the input images. This process scales the pixel values to a range between 0 and 1 ensuring that input data is consistent, this makes the training more stable and leads to faster convergence.

```
transform = transforms.Compose([transforms.ToTensor()])
mnist_train = datasets.MNIST(data_directory, train=True, download=download_dataset, transform=transform)
mnist_test = datasets.MNIST(data_directory, train=False, download=download_dataset, transform=transform)
```

The above code snippet shows how the above preprocessing step was done using **transforms**.

Furthermore, I utilised batch processing and shuffling during the training step. Batch processing divides the training set into batches of a fixed size, specifically 512 [iii] in my implementation. This allows the network to make updates to its parameters based on gradients computed from each 512-size batch. I opted for Batch processing because it enables more efficient optimization as the gradients are computed and applied incrementally rather than the whole dataset at once.

The order of the training data is shuffled before each epoch preventing the network from relying on the order of supplied data to learn. This contributes to the network being general.

```
train_loader = data.DataLoader(mnist_train, batch_size=batch_size, shuffle=True, num_workers=1)
```

The above code snippet shows how the batch processing and shuffling was done.

Loss Function

The loss function is a very important component in a neural network as it measures the difference between the predicted output of the network and the actual target values. In my implementation, the Cross Entropy Loss function is utilized.

I opted for the Cross Entropy Loss because it is commonly used for classification tasks especially when dealing with multiple classes[iv], this met the description of my own model's purpose which is classifying images of handwritten digits into multiple possible classes (digits 0 to 9). This loss function calculates the loss by comparing predicted probabilities of each class generated by the model to actual class labels.

The aim of the neural network is to minimize this loss during training and effectively adjusts its parameters to improve its ability to classify input images.

```
loss_function = torch.nn.CrossEntropyLoss()
```

The above code snippet shows how I instantiated my loss function.

Optimizer

In neural network training, the optimizer plays a crucial role in adjusting the parameters of the model to minimize the loss function during the process of training. The optimizer updates the weights of the neural network based on the gradients of the loss function with respect to the weights it updates.

In my implementation, I opted for the **Adam** optimizer as it offers the benefits of both AdaGrad and RMSProp while addressing their limitations[v]. Adam dynamically adjusts the learning rates for each parameter based on the history of its gradients. This makes it well-suited for training neural networks with varying parameter weights.

Adam adjusts the learning rate to be larger if the parameter at hand has been updated less frequently which indicates that its impact on the loss is minimal. Conversely, it makes the learning rate smaller for updates of parameters that have been changed frequently indicating their impact on the loss function is great. This adaptive mechanism helps prevent overshooting or jumping around optimal parameters.

Utilising Adam streamlines the training process as it automates the adjustment of learning rate, saving time that would be spent experimenting.

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

The above code snippet shows my initialisation of Adam, I used 0.001 as the starting learning rate but Adam changes it by adapting it to the parameter.

ANN Training & Validation

Training procedure:

The training of the neural network involves multiple epochs where each epoch represents a complete pass through the entire training dataset. The MNIST dataset is split into batches, and the model is updated batch by batch. The batch-wise training is done using a batch size of 512 as mentioned in the above preprocessing section. This allows the network to generalize better by updating weights based on a subset of the data rather than on individual samples or the entire dataset at once.

During training, after computing the forward pass and obtaining the predictions from the neural network, a loss is calculated using the cross-entropy loss function. This loss as stated in the loss function section provides a measure of how well the network's predictions match actual labels. The Adam optimizer is then used to update the weights of the network based on this loss, utilizing its adaptive learning rate.

```
loss.backward()
optimizer.step()
```

Code snippet for the loss function computation and optimization using Adam

Validation:

```
mnist_train, mnist_validation = data.random_split(mnist_train, (48000, 12000))
```

To assess the model's performance and mitigate overfitting, I used a separate validation dataset different from the training set. This dataset was crucial to evaluating model performance at the end of each training epoch. It allows for monitoring the model on unseen data and seeing if it makes progress on being a better classifier.

During the validation, no weight updates are made instead the model undergoes forward passes with the validation data and an accuracy is calculated using a *calculate_accuracy (outputs,expected)* method that calculates the percentage of the model's predictions that are correct by comparing model predictions with actual validation set labels. I used these results as part of a regularisation mechanism which I will delve in below.

Early stopping:

To further combat overfitting, in addition to dropout, I implemented early stopping. This technique involves cutting the training process short before all specified epochs are completed if the validation performance does not improve after a predefined benchmark. In my implementation, I configured the training to stop if the validation accuracy does not improve for five consecutive epochs, with a maximum of 100 epochs allowed. This prevents the model from memorizing the training data and instead encourages true learning enhancing the model's ability to generalize.

```
if best_accuracy == -1 or accuracy > best_accuracy:
    print("New best epoch ", epoch, "accuracy", accuracy)
    best_accuracy = accuracy
    best_model = model.state_dict()
    best_epoch = epoch
if best_epoch + no_improvement_epochs <= epoch:
    print("Done!\n")
    break
```

Model Evaluation:

Finally , after training and validation phases, I evaluated the model on a test set to assess how well it has learned to generalized beyond the data it was trained and validated on. The final model accuracy, along with training and validation losses, are then recorded in a log.txt file for each epoch.

Neural Network Architecture & Hyperparameters

To determine the neural network architecture and hyperparameters, I devised an experiment whereby I adjusted the number of hidden layers, the number of nodes per layer and dropout rate while keeping certain parameters constant.

My experiments were structured into 18 different configurations, each representing a unique combination of hidden layers, nodes per layer and dropout rate. For each configuration , I trained the model and evaluated the test set to measure its accuracy and the number of epochs required for convergence.

Constant values:

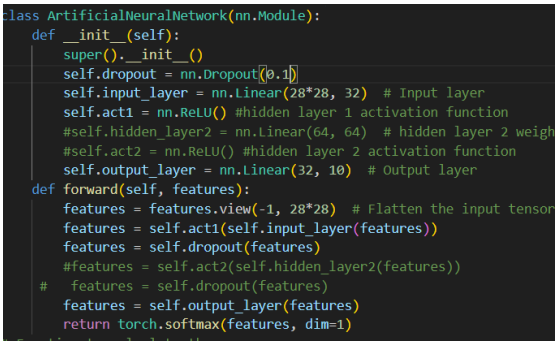
Activation function: ReLU

Loss function: Cross-Entropy-Loss

Optimizer : ADAM with Lr=0.001, usage of Adam adapts learning rate therefore I did not have to experiment with different learning rates

Batch size: 512

Experiment no	Hidden layers	Nodes per layer	Dropout Rate	Test set accuracy	Num epochs to converge
1	1	32	0.1	0.939453125	8
2	1	32	0.2	0.94921875	14
3	1	32	0.4	0.91796875	17
4	1	64	0.1	**0.970703125**	31
5	1	64	0.2	0.96484375	29
6	1	64	0.4	0.95703125	13
7	1	128	0.1	**0.972515625**	20
8	1	128	0.2	0.966796875	13
9	1	128	0.4	0.96875	13
10	2	32	0.1	0.9453125	27
11	2	32	0.2	0.939453125	16
12	2	32	0.4	0.921875	20
13	2	64	0.1	**0.974609375**	28
14	2	64	0.2	0.96875	21
15	2	64	0.4	0.94921875	17
16	2	128	0.1	**0.984375**	17
17	2	128	0.2	**0.98046875**	17
18	2	128	0.4	0.96875	14



```
class ArtificialNeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.dropout = nn.Dropout(0.1)
        self.input_layer = nn.Linear(28*28, 32)  # Input layer
        self.act1 = nn.ReLU() #hidden layer 1 activation function
        #self.hidden_layer2 = nn.Linear(64, 64)  # hidden layer 2 weigh
        #self.act2 = nn.ReLU() #hidden layer 2 activation function
        self.output_layer = nn.Linear(32, 10)  # Output layer
    def forward(self, features):
        features = features.view(-1, 28*28)  # Flatten the input tensor
        features = self.act1(self.input_layer(features))
        features = self.dropout(features)
        #features = self.act2(self.hidden_layer2(features))
     #    features = self.dropout(features)
        features = self.output_layer(features)
        return torch.softmax(features, dim=1)
```

Example of code, when performing experiment 1

After conducting the experiment, I settled on the following Artificial Neural Network and hyperparameter configuration as it gave me a relatively high accuracy on the test set and even though it is not the very best configuration as per experiment, it strikes a better balance between mitigating overfitting and avoiding underfitting also while keeping the complexity of the network moderate.

<u>Chosen Configuration</u>

Hidden layers : 2

Nodes per layer: 64

Dropout rate : 0.2

Activation function: ReLU

Loss function : Cross Entropy Loss

Batch size : 512

[i] Brownlee, J. (2019) How to avoid overfitting in Deep Learning Neural Networks, MachineLearningMastery.com. Available at: https://machinelearningmastery.com/introduction-to-regularization-to-reduce-overfitting-and-improve-generalization-error/ (Accessed: 21 April 2024).

[ii] Piepenbreier, N. (2023) ReLU activation function for Deep Learning: A complete guide to the Rectified Linear Unit • datagy, datagy. Available at: https://datagy.io/relu-activation-function/ (Accessed: 21 April 2024).

[iii] Pramoditha, R. (2023) Determining the right batch size for a neural network to get better and faster results, Medium. Available at: https://medium.com/data-science-365/determining-the-right-batch-size-for-a-neural-network-to-get-better-and-faster-results-7a8662830f15 (Accessed: 21 April 2024).

[iv] What is cross-entropy loss function? (2023) 365 Data Science. Available at: https://365datascience.com/tutorials/machine-learning-tutorials/cross-entropy-loss/ (Accessed: 21 April 2024).

[v] Brownlee, J. (2021) Gentle introduction to the adam optimization algorithm for deep learning, MachineLearningMastery.com. Available at: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/ (Accessed: 21 April 2024).