Assignment: ANN with Backpropagation for Digit Classification

1. Structure of the ANN

My Artificial Neural Network (ANN) implementation adheres to the following structure:

- **Input Layer:** 784 nodes (corresponding to the 784 pixel values of a 28x28 image)
- **Hidden Layer 1:** 100 nodes
- **Hidden Layer 2:** 50 nodes
- **Output Layer:** 10 nodes (corresponding to the 10 possible classes of digits)

I chose to set up my ANN as a three-layer network (input layer, two hidden layers, and output layer). The weights and biases for each layer are initialized randomly. The `relu` function is used as the activation function for the hidden layers, and the `softmax` function is used for the output layer to provide a probability distribution for multi-class classification.

The `forward_propagation` function calculates the outputs of the network for a given input, and the `back_propagation` function updates the weights and biases based on the error between the network's output and the actual output.

The following are the main operations in the algorithm:

Forward Propagation

In forward propagation, the data flows from the input layer through the hidden layers to the output layer. The goal is to calculate the output for each neuron all the way to the final output of the network.

1. `input_to_hidden1 = np.dot(X, self.weights_ih.T)`: calculates the weighted sum of the inputs for the first hidden layer. It multiplies the input data `X` by the weights connecting the input layer to the first hidden layer (`self.weights_ih`), and adds the bias (`self.bias_h1.T`).
2. `self.hidden1 = self.relu(input_to_hidden1 + self.bias_h1.T)`: applies the ReLU activation function to the weighted sum to get the output of the first hidden layer.
3. The same process is repeated for the second hidden layer and the output layer. The only difference is that the softmax function is used as the activation function for the output layer, since we want to obtain a multi-class probability distribution as output.

Backward Propagation

Backward propagation is the process of updating the weights and biases of the network in response to the error the network produced in the current forward pass. The goal is to adjust the weights and biases to minimize this error.

1. `outputs_error = y - self.outputs`: calculates the error of the output layer by subtracting the actual output from the expected output.
2. `d_weights_ho = np.dot(outputs_error.T, self.hidden2) / count`: calculates the gradient of the loss with respect to the weights of the output layer. It is used to update the weights in the direction that minimizes the loss.
3. `d_bias_o = np.sum(outputs_error, axis=0, keepdims=True).T / count`: calculates the gradient of the loss with respect to the biases of the output layer. It is used to update the biases in the direction that minimizes the loss.
4. The same process is repeated for the first and second hidden layers.

5. Finally, the weights and biases are updated using the calculated gradients and the learning rate. The learning rate is a hyperparameter that determines the step size at each iteration while moving toward a minimum of a loss function. It controls how much we are adjusting the weights with respect to the loss gradient.

This process of forward propagation and backward propagation is repeated for 500 epochs during the training process. Each epoch is a complete pass through the entire training dataset.

2. Equations for Updating Weights

Output Layer (weights from hidden layer 2 to output layer)

$$\Delta W_{ho} = \frac{1}{N} \sum_{i=1}^{N} (\mathbf{y}_i - \hat{\mathbf{y}}_i) \cdot \mathbf{h}_2^T$$

$$\Delta b_o = \frac{1}{N} \sum_{i=1}^{N} (\mathbf{y}_i - \hat{\mathbf{y}}_i)$$

Where:

- ΔW_{ho} is the change in the weights from hidden layer 2 to the output layer.
- Δb_o is the change in the biases for the output layer.
- $\mathbf{y}_i$ is the true label (one-hot encoded) for the i-th sample.
- $\hat{\mathbf{y}}_i$ is the predicted output for the i-th sample.
- $\mathbf{h}_2$ is the activation of the second hidden layer.
- N is the number of samples.

Hidden Layer 2 (weights from hidden layer 1 to hidden layer 2)

$$\Delta W_{hh} = \frac{1}{N} \sum_{i=1}^{N} \delta_{h2,i} \cdot \mathbf{h}_1^T$$

$$\Delta b_{h2} = \frac{1}{N} \sum_{i=1}^{N} \delta_{h2,i}$$

Where:

- ΔW_{hh} is the change in the weights from hidden layer 1 to hidden layer 2.
- Δb_{h2} is the change in the biases for the second hidden layer.
- $\delta_{h2,i}$ is the error term for hidden layer 2, given by:

$$\delta_{h2,i} = (\mathbf{y}_i - \hat{\mathbf{y}}_i) \cdot W_{ho} \cdot f'(\mathbf{h}_2)$$

- $\mathbf{h}_1$ is the activation of the first hidden layer.
- $f'(\mathbf{h}_2)$ is the derivative of the ReLU activation function applied to the second hidden layer activations.

Hidden Layer 1 (weights from input layer to hidden layer 1)

$$\Delta W_{ih} = \frac{1}{N} \sum_{i=1}^{N} \delta_{h1,i} \cdot \mathbf{x}^T$$

$$\Delta b_{h1} = \frac{1}{N} \sum_{i=1}^{N} \delta_{h1,i}$$

Where:

- ΔW_{ih} is the change in the weights from the input layer to hidden layer 1.
- Δb_{h1} is the change in the biases for the first hidden layer.
- $\delta_{h1,i}$ is the error term for hidden layer 1, given by:

$$\delta_{h1,i} = (\delta_{h2,i} \cdot W_{hh}) \cdot f'(\mathbf{h}_1)$$

- $\mathbf{x}$ is the input vector.
- $f'(\mathbf{h}_1)$ is the derivative of the ReLU activation function applied to the first hidden layer activations.

Update Rules

The weights and biases are updated using the following rules:

$$W_{ho} \leftarrow W_{ho} + \eta \Delta W_{ho}$$

$$b_o \leftarrow b_o + \eta \Delta b_o$$

$$W_{hh} \leftarrow W_{hh} + \eta \Delta W_{hh}$$

$$b_{h2} \leftarrow b_{h2} + \eta \Delta b_{h2}$$

$$W_{ih} \leftarrow W_{ih} + \eta \Delta W_{ih}$$

$$b_{h1} \leftarrow b_{h1} + \eta \Delta b_{h1}$$

Where η is the learning rate.

3. Percentage of Correctness

This is the function I use to calculate the metrics:

```
test_accuracy = np.mean(predictions == targets)
print(f'Test Accuracy: {test_accuracy * 100}%')

for i in range 10:
    correct = np.sum((predictions == i) & (targets == i))
    total = np.sum(targets == i)
    correctness = correct / total if total > 0 else 0
    print(f'Class {i}: Correctness {correctness *

 100}%')
```

Here are the values from the most recent run of the algorithm:

- Finished training
- Test Accuracy: 86.30%
- Class 0: Correctness 95.29%
- Class 1: Correctness 96.84%
- Class 2: Correctness 80.55%
- Class 3: Correctness 81.09%
- Class 4: Correctness 86.92%
- Class 5: Correctness 77.56%
- Class 6: Correctness 91.43%
- Class 7: Correctness 89.78%
- Class 8: Correctness 81.32%
- Class 9: Correctness 80.12%

4. Validation Accuracy Over Time

The figure below illustrates the change in accuracy metric value over the validation set.

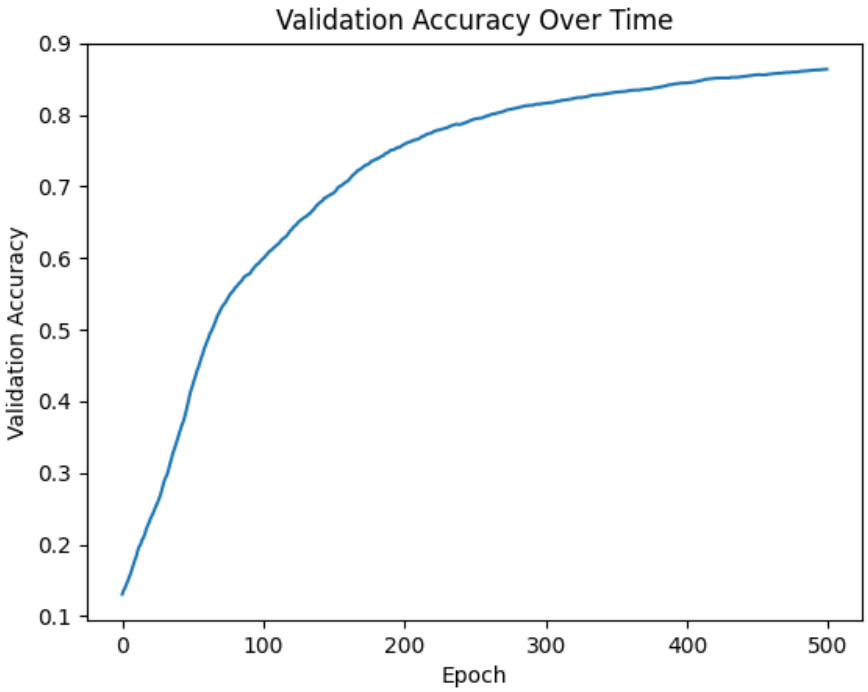

Figure 1: Validation Accuracy Over Training Epochs

It can be seen that it improves in a log-like manner and reaches above 80% accuracy towards the 500 epochs point.