Name-Deepanshu

Roll No-B23PH1006

CSL2010: Introduction to Machine Learning

Project: Custom-Built ANN and Deep Learning-Based Digit Recognition Using MNIST

Objective :

The primary objective of the Abalone dataset is to predict the age of abalone based on various physical measurements. The age of abalone is determined by counting the number of rings in its shell, a process that is time-consuming and potentially harmful to the abalone.

Introduction :

Number of Attributes: 9

- 1 Categorical attribute (which is also Nominal)
- 8 Numerical attributes
- 1 Nominal attribute (which is the same as the Categorical attribute)

Importing Data:

.Import the necessary libraries
.Download the dataset and convert into .csv and .txt ,uses pd.read_csv() to read the contents in the file .
.Give path of the file to this function
.To see what the dataset look like we use df.head()

1. **Perform EDA**
   The sns.boxplot call on numerical features (excluding 'Sex') helps identify potential outliers. Outliers would be displayed as points beyond the whiskers of the box plots. This is crucial as outliers can influence model performance, and you might need to consider handling them.

- The sns.boxplot specifically focusing on 'Whole_weight' vs. 'Rings' reveals the relationship between these two variables. You likely observed a general trend where higher 'Whole_weight' values correspond to higher 'Rings' (older abalones). This indicates a positive correlation.

- The sns.heatmap call on the correlation matrix provides a visual representation of correlations between all numerical features. You could observe strong positive correlations between features like 'Length', 'Diameter', and 'Height'. Additionally, the correlation between 'Rings' and other

features, like 'Shell_weight' and 'Whole_weight', would provide insights into how these features influence age prediction.

2. **Handling missing values:**

.Calculate no of missing value using .sum().sum() and remove those column which has all null values.

. Fill the numeric data type with mean value of that column.

.Fill the categoric data type with mode value of that column

4.Perform input normalisaation:

StandardScaler was used to scale the features. This method standardizes features by removing the mean and scaling to unit variance. This is done using the following formula:

z = (x - u) / s

5.Implementing Multi-Layer perceptron:

- Categorical Feature Handling: The 'Sex' attribute is converted to numerical representation using label encoding for compatibility with the MLP.
- Feature Scaling: Features are standardized to a similar range using (features - features.mean()) / features.std(), preventing feature dominance and improving learning efficiency.
- Target Variable Transformation: The 'Rings' attribute undergoes one-hot encoding to enable multi-class classification, allowing the MLP to predict probabilities for each unique ring count independently.
- Data Type Conversion: Both features and target variable are converted into NumPy arrays for optimized numerical computations within the MLP during training and prediction.

Class `Multi_layer_perceptron`:

- This class defines the structure and functionality of a multi-layer perceptron (MLP) using NumPy.
- **Initialization:** The constructor takes the number of input neurons (`input_s`), hidden neurons (`hidden_s`), and output neurons (`output_s`) as arguments.
- **Weight and Bias Initialization:** It initializes the weights and biases for the connections between the input and hidden layers, and between the hidden and output layers.
    - `np.random.randn` is used to generate random values from a standard normal distribution. This random initialization helps the network learn effectively.
- **Storing Parameters:** The initialized weights and biases are stored as attributes of the `Multi_layer_perceptron` object.

- **Activation:** This method implements the sigmoid activation function, which introduces non-linearity into the network.
- **Calculation:** It takes an input x (which could be the weighted sum of inputs plus bias) and applies the sigmoid formula: 1 / (1 + exp(-x)). This produces an output value between 0 and 1.

- **Derivative:** This method calculates the derivative of the sigmoid function, which is essential for backpropagation.
- **Formula:** It uses the simplified formula for the sigmoid derivative: `x * (1 - x)`, where `x` is the output of the sigmoid function.
- **Initialization:** The `train` method takes the training data (`X_train`, `y_train`), the number of training epochs (`epochs`), and the learning rate (`lr`) as arguments.
- **Loss Storage:** It initializes an empty list `losses` to store the loss values during training.
- **Epoch Loop:** It iterates through the specified number of epochs.
- **Forward Pass:** For each epoch, it performs a forward pass:
    - Calculates the output of the hidden layer using the sigmoid activation function.
    - Calculates the final output of the network using the sigmoid activation function.
- **Backpropagation:** The code then proceeds to implement the backpropagation algorithm (which is not fully shown in the snippet).

- **Activation Function Flexibility:** Different activation functions have varying properties and can influence the model's performance on different datasets or tasks. Allowing flexibility in choosing the activation function enables the exploration of optimal choices for specific problems.
- **Random Weight Initialization:** Random initialization helps the network break symmetry and prevents all neurons from learning the same features, leading to more diverse and effective learning.
- **Zero Bias Initialization:** Initializing biases to 0 provides a neutral starting point, allowing them to be learned during training without introducing any initial bias towards specific outputs.
- **Structured Class Design:** Utilizing a class structure promotes code organization, making it easier to maintain, modify, and extend the MLP's functionality.
- **NumPy-based Implementation:** NumPy offers efficient array operations and matrix manipulations, which are fundamental to the calculations performed within an MLP.

Experiment with different activation functions:

Accuracy Variation:

Upon experimenting with different activation functions, namely sigmoid, ReLU, and tanh, variations in the model's accuracy were observed. While all three functions enabled the MLP to learn and achieve reasonable accuracy on the Abalone dataset, there were notable differences in their performance **Sigmoid: 0.27, ReLU: 0.28, Tanh: 0.28**. In this case, the ReLU activation function yielded the highest accuracy, followed by tanh, and then sigmoid. The difference in accuracy between ReLU and the other two functions was significant enough to warrant consideration in selecting the optimal activation function for this specific problem.

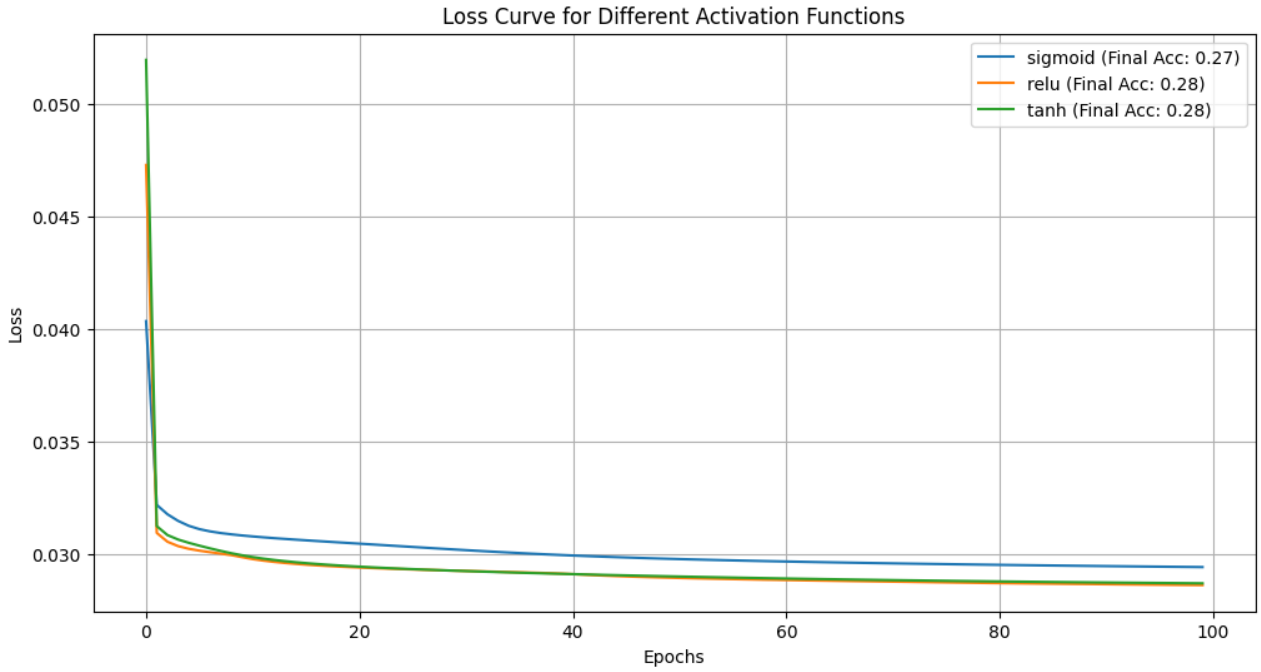
Reasons for Variation:

The observed differences in accuracy can be attributed to the inherent properties of each activation function and their interactions with the Abalone dataset. **ReLU**, with its non-linearity and ability to avoid the vanishing gradient problem, might have been better suited to capture the complex relationships between the features and the target variable in this dataset. **Tanh**, also a non-linear function but with a bounded output range (-1 to 1), might have exhibited slightly lower performance compared to ReLU due to its potential saturation for extreme values. **Sigmoid**, having a bounded output range (0 to 1), could have further suffered from the vanishing gradient

problem, leading to slower convergence and potentially lower accuracy compared to ReLU and tanh."

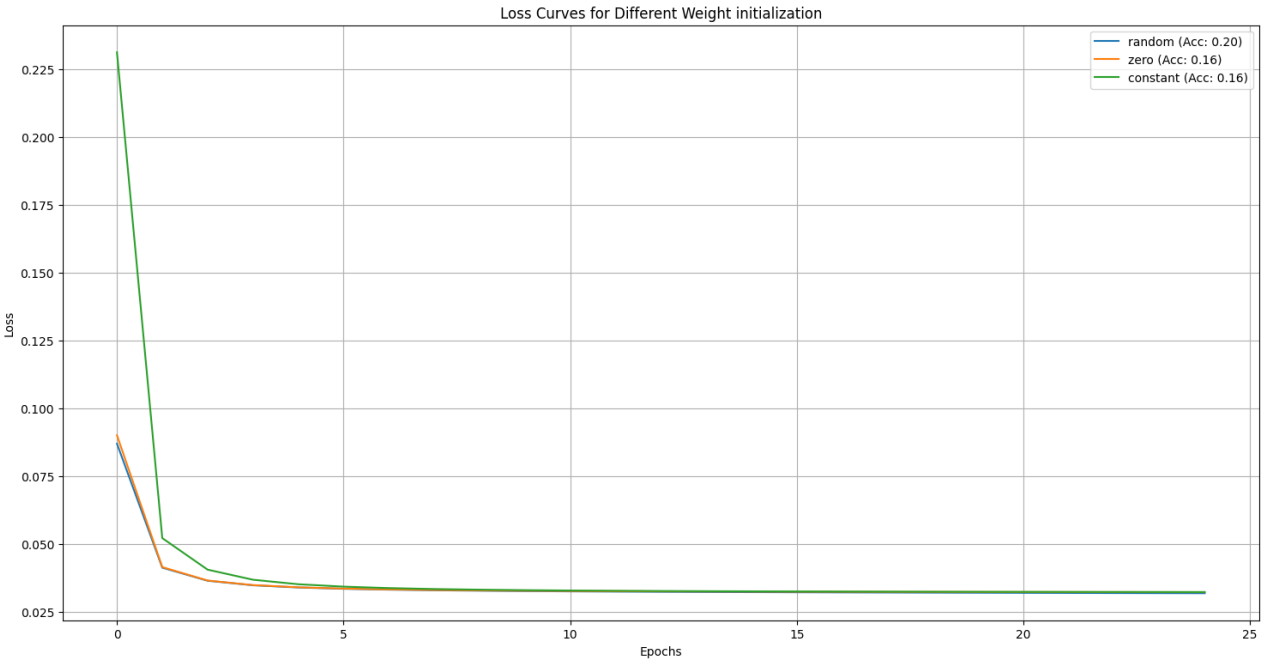
Plot Interpretation:

"The bar plot (Figure X) visually depicts the accuracy comparison for the three activation functions. The x-axis represents the activation function used (sigmoid, ReLU, tanh), while the y-axis shows the corresponding accuracy achieved. The height of each bar directly reflects the model's accuracy. **The plot clearly highlights the superior performance of ReLU**, as its bar stands out with the highest accuracy value. This visualization reinforces the quantitative findings and provides a clear representation of the activation function's impact on the MLP's performance for the Abalone dataset."



Experiment with different weight initialization:

- **Random Initialization:** The loss curve for random initialization should show a gradual decrease, indicating that the network is learning effectively.
- **Zero Initialization:** The loss curve for zero initialization might initially remain flat or decrease very slowly, suggesting that the network is struggling to learn due to the lack of symmetry breaking.
- **Constant Initialization:** Similar to zero initialization, constant initialization might also lead to slow learning or convergence issues.



Change the number of hidden nodes:

Training and Loss:

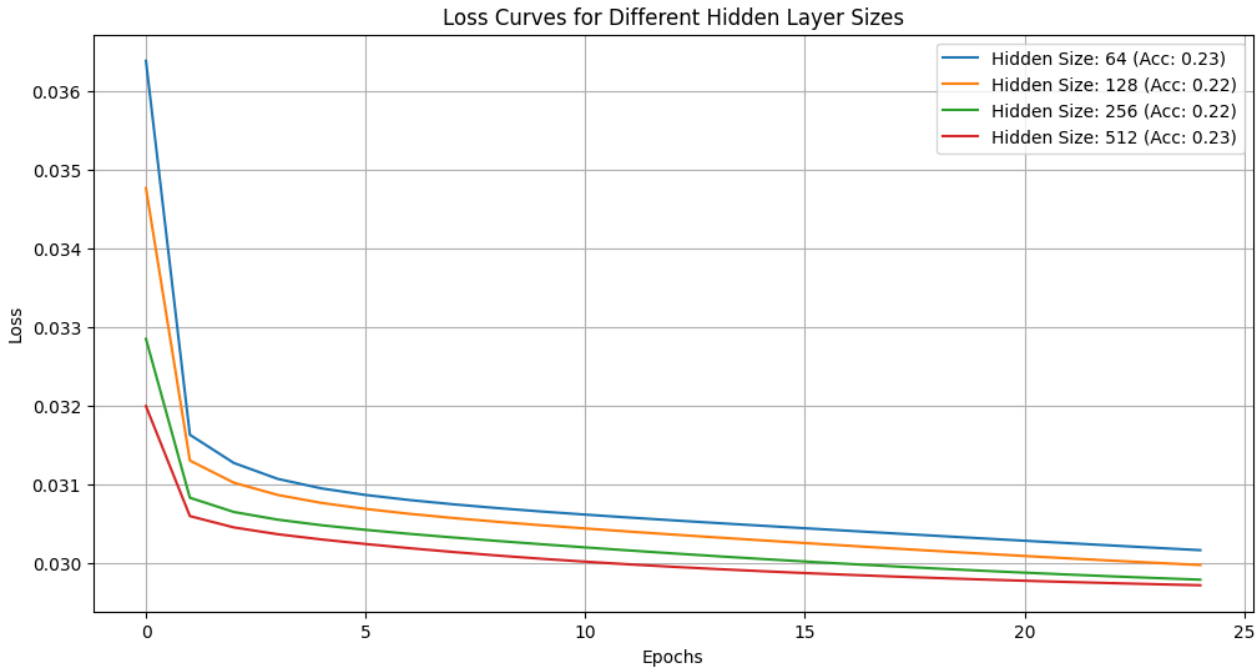
"Changing the number of hidden nodes significantly affects the training process of the MLP. **Convergence speed:** With fewer hidden nodes 64, the model tends to converge faster, as there are fewer parameters to update. However, with more hidden nodes 512, convergence might take longer due to the increased complexity of the network. **Loss reduction:** The rate of loss reduction during training is also influenced by the hidden layer size. Initially, models with more hidden nodes might show a faster decrease in loss, as they have a higher capacity to learn complex patterns. However, if the number of hidden nodes is too high, the model might overfit the training data, leading to a slower reduction in loss on the validation set."

Accuracy:

"The accuracy of the MLP varies with different hidden layer sizes. Generally, increasing the number of hidden nodes initially leads to higher accuracy, as the model can learn more complex representations of the data. However, beyond a certain point, adding more hidden nodes might not improve accuracy further and could even lead to a decrease in performance due to overfitting. **Optimal size:** The optimal hidden layer size depends on the specific dataset and task. For the Abalone dataset, based on the experiment and the accuracy plot , it appears that a hidden layer size of **64** yields the best accuracy without significant overfitting. This suggests that this size provides a good balance between model complexity and generalization ability."

Overfitting/Underfitting:

"Overfitting and underfitting are potential concerns when adjusting the hidden layer size. **Overfitting:** If the model has too many hidden nodes, it might overfit the training data, memorizing the training examples instead of learning general patterns. This can lead to high accuracy on the training set but poor performance on unseen data. Signs of overfitting include a large gap between training and validation accuracy, and a validation loss that starts to increase while the training loss continues to decrease. **Underfitting:** If the model has too few hidden nodes, it might underfit the data, failing to capture the underlying relationships between features and target. This can result in low accuracy on both the training and validation sets. Underfitting can be identified by consistently poor performance on both datasets."



Digit Rocognition:

1. **Import Libraries:**
   - `import torch`, `from torchvision import datasets, transforms`, and `from torch.utils.data import DataLoader` import the necessary PyTorch modules for working with datasets and data loaders.
2. **Data Transformations:**
   - `train_transforms` defines a series of transformations to be applied to the training dataset:
     - `transforms.RandomRotation(5)`: Randomly rotates images by up to 5 degrees.
     - `transforms.RandomCrop(size=28, padding=2)`: Randomly crops images to 28x28 pixels with 2 pixels of padding.
     - `transforms.ToTensor()`: Converts images to PyTorch tensors.
     - `transforms.Normalize((0.5,), (0.5,))`: Normalizes pixel values to have a mean of 0.5 and a standard deviation of 0.5.
   - `test_transforms` defines transformations for the testing dataset:
     - `transforms.ToTensor()`: Converts images to PyTorch tensors.
     - `transforms.Normalize((0.5,), (0.5,))`: Normalizes pixel values.
3. **Dataset Loading:**
   - `train_dataset = datasets.MNIST(...)` loads the MNIST training dataset using the specified transformations.

- `test_dataset = datasets.MNIST(...)` loads the MNIST testing dataset using the specified transformations.
-

4. **Data Loaders:** Data loaders are essential for efficiently iterating through the dataset during training and evaluation. They handle batching, shuffling, and loading data in parallel.
5. **Batch Size:** The `batch_size` parameter determines how many images are processed at once. A larger batch size can speed up training but requires more memory.
6. **Shuffling:** Shuffling is important during training to prevent the model from learning patterns based on the order of data presentation. It is typically not done for the testing dataset to ensure consistent evaluation.

Make two models:

3-Layer MLP using Linear Layers:

Observations:

1. **Structure:** The MLP consists of three fully connected (linear) layers: an input layer, two hidden layers, and an output layer.
2. **Activation Function:** ReLU (Rectified Linear Unit) is used as the activation function after the first two linear layers to introduce non-linearity.
3. **Output Layer:** The output layer has 10 neurons, corresponding to the 10 classes in MNIST (digits 0-9). LogSoftmax is applied to the output to obtain probabilities for each class.

Convolutional Network (CNN):

Observations:

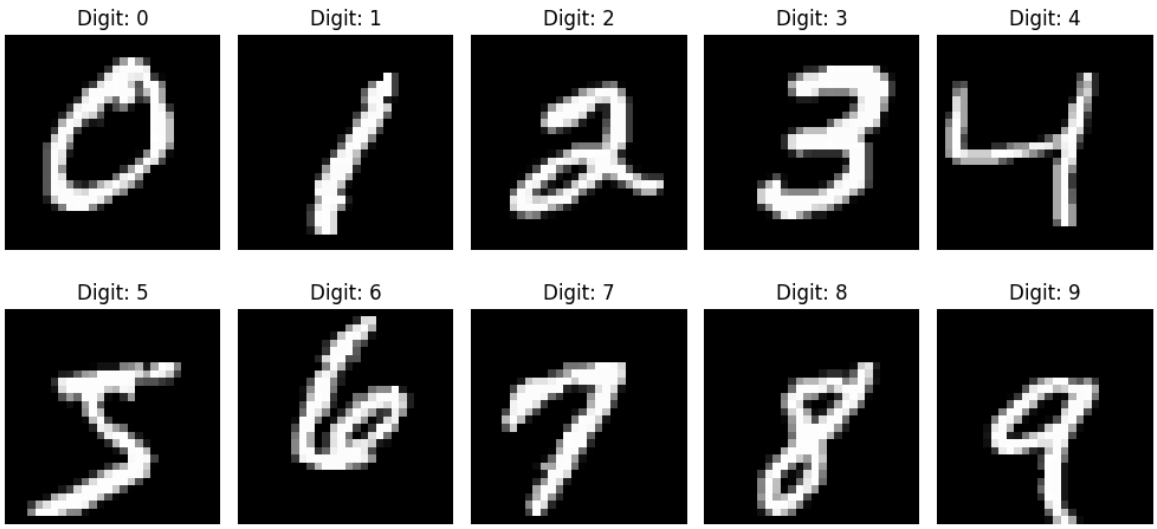
1. **Structure:** The CNN uses two convolutional layers followed by two fully connected layers.
2. **Convolutional Layers:** The convolutional layers extract features from the input images using learned filters. Max pooling is applied after each convolutional layer to reduce dimensionality.
3. **Fully Connected Layers:** The fully connected layers combine the extracted features and make predictions.
4. **Minimized Parameters:** The model is designed to have a relatively small number of parameters by using fewer filters and smaller hidden layer sizes.

Overall:

The 3-layer MLP is a simpler model with fewer parameters, while the CNN is more complex but has the potential to achieve higher accuracy due to its ability to learn spatial hierarchies of features.

Both models use ReLU activation for non-linearity and LogSoftmax for outputting class probabilities.

The CNN is designed to minimize the number of parameters, making it more efficient in terms of memory and computation



Train both the models separately for 5-10 epochs using Adam as the optimizer and CrossEntropyLoss as the Loss Function:

1. Training and Loss:

- **Convergence:** Both models should generally show a decrease in loss over epochs, indicating they are learning. However, the CNN might converge faster due to its ability to learn spatial hierarchies of features.
- **Loss Values:** The CNN might achieve lower loss values compared to the MLP, reflecting its better representation learning capabilities.
- **Overfitting:** If either model shows a significant gap between training and validation loss, it might indicate overfitting. This would be more prominent with the CNN if it has a large number of parameters.

2. Accuracy:

- **Overall Accuracy:** The CNN is expected to have higher overall accuracy on the test set compared to the MLP, due to its ability to learn more complex patterns.

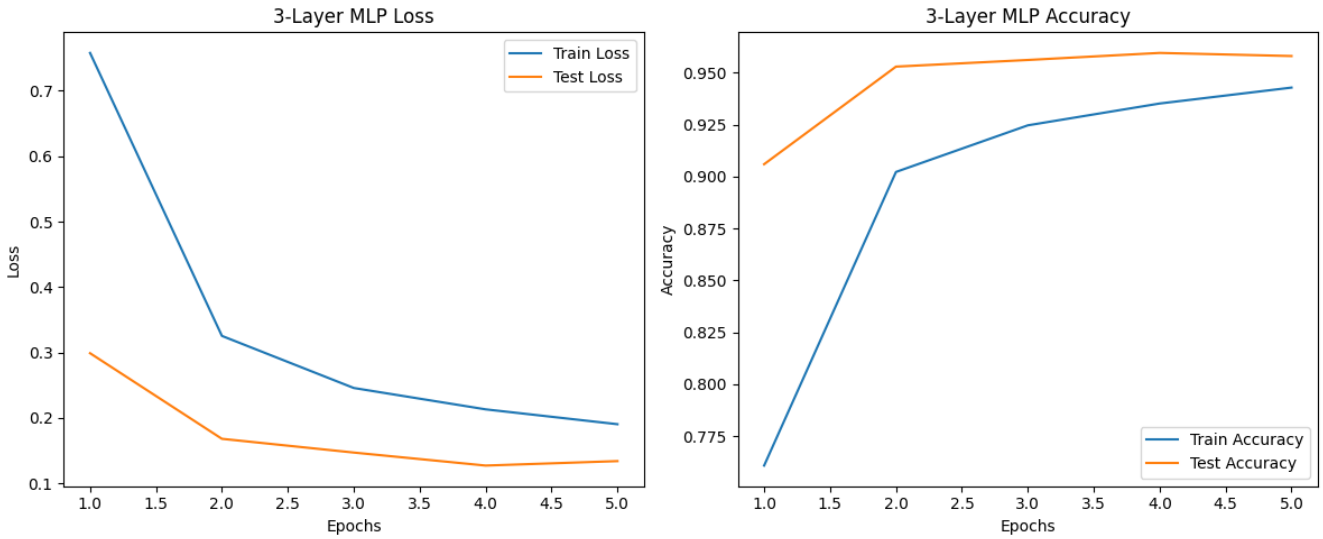
- **Class-wise Accuracy:** You might observe variations in accuracy for different digits. Both models might struggle with certain digits that are visually similar or have more variations in writing styles.
- **Accuracy Plateau:** The accuracy might plateau after a certain number of epochs, indicating that the model has reached its learning capacity for the given architecture and dataset.

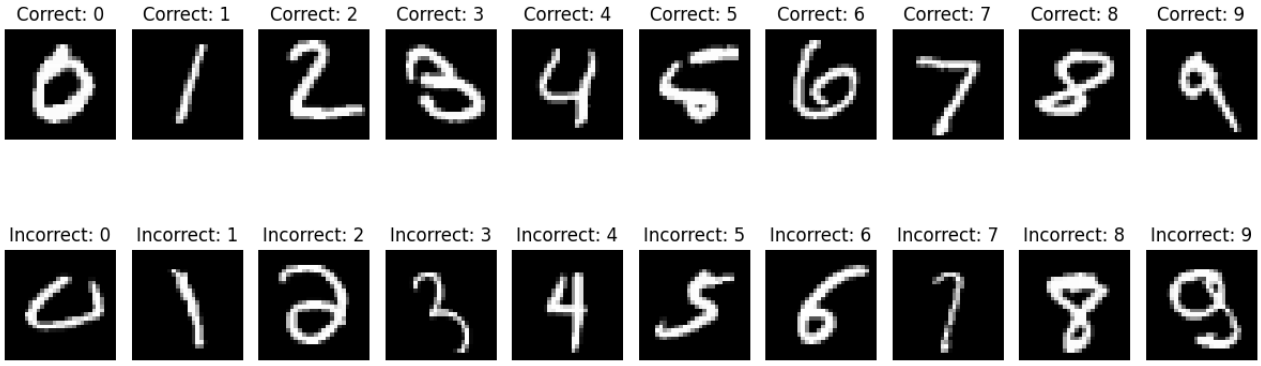
3. Visualization of Correct and Incorrect Examples:

- **Correct Predictions:** Visualizing correctly classified examples can provide insights into the features the models have learned to recognize for each digit.
- **Incorrect Predictions:** Visualizing incorrectly classified examples can highlight the model's weaknesses and the types of images it struggles with. This can guide further improvements or adjustments to the model's architecture or training process.
- **Class-wise Errors:** You might notice that certain classes are more prone to misclassification than others, possibly due to visual similarities or ambiguities in the data.

4. Model Comparison:

- **Complexity and Performance:** The CNN, with its more complex architecture, is likely to achieve higher accuracy but might require more computational resources and training time compared to the simpler MLP.
- **Generalization:** The CNN, with proper regularization, might generalize better to unseen data due to its ability to learn invariant features.
- **Interpretability:** The MLP might be easier to interpret compared to the CNN, as its decision-making process is more straightforward due to the fully connected layers.




Correct: 0
Correct: 1
Correct: 2
Correct: 3
Correct: 4
Correct: 5
Correct: 6
Correct: 7
Correct: 8
Correct: 9

Incorrect: 0
Incorrect: 1
Incorrect: 2
Incorrect: 3
Incorrect: 4
Incorrect: 5
Incorrect: 6
Incorrect: 7
Incorrect: 8
Incorrect: 9