

Deep Learning- Assignment 1

Neural Networks

Shivani Nandkishor Nipane

STUDENT ID - 24622969

Contents

| | |
|--|----|
| 1.Introduction | 2 |
| 2. Presentation of Data | 3 |
| 2.1 Dataset Overview | 3 |
| Part A: Perceptron Dataset | 3 |
| Part B: Japanese MNIST Dataset | 3 |
| 2.2 Data Preprocessing | 3 |
| 2.2.1 Part A | 3 |
| 2.2.2 Part B | 4 |
| 3. Methodology | 4 |
| 3.1 Part B: Implementing the Perceptron | 4 |
| 3.2 Part B: Training Custom Neural Networks | 6 |
| 3.2.1 Experiment 1: Basic Fully Connected Network | 6 |
| 3.2.2 Experiment 2: Network with Batch Normalization and Dropout | 8 |
| 3.2.3 Experiment 3: Advanced Network with Layer Normalization and LeakyReLU | 10 |
| 4. Experimentation and Results | 12 |
| 4.1 Part A Results: Perceptron Training Process and Performance Evaluation | 12 |
| 4.1.1 Training Process Description | 12 |
| 4.1.2 Performance Metrics and Evaluation | 13 |
| 4.2 Part B Experiments: | 14 |
| 4.2.1 Experiment 1: Basic Fully Connected Network | 14 |
| 4.2.1 Experiment 2: Network with Batch Normalization and Dropout | 16 |
| 4.2.3 Experiment 3: Advanced Network with Layer Normalization and LeakyReLU | 17 |
| 5. Analysis of Model Performance and Limitations | 19 |
| 5.1 Part A: Perceptron Model | 19 |
| 5.1.1 What Worked Well: | 19 |
| 5.1.2 Limitations: | 20 |
| 5.2 Part B: Custom Neural Networks | 20 |
| 5.2.1 What Worked Well: | 20 |
| 5.2.2 Limitations: | 20 |
| 5.2 Overall Insights: | 20 |
| 6. Remaining Issues and Recommendations | 21 |
| 6.1 Remaining Issues: | 21 |
| 6.2 Recommendations for Future Work: | 21 |
| 7. Conclusion | 22 |

Abstract

This project delves into artificial intelligence and neural networks, focusing on constructing a perceptron from scratch and training custom neural networks on the Japanese MNIST dataset. It is divided into two main parts: The first establishes foundational knowledge by creating a simple perceptron using Python libraries Numpy and Pandas for binary classification. In contrast, the second escalates the challenge by training neural networks to classify handwritten Hiragana characters to achieve at least 80% accuracy. This endeavour bridges theoretical concepts with practical applications, underscoring the significance of neural network architectures in modern AI advancements.

1. Introduction

In the rapidly evolving field of artificial intelligence, neural networks stand at the forefront, driving advancements in machine learning and deep learning. The perceptron and custom neural networks play pivotal roles, acting as the foundational blocks for understanding complex neural architectures. This project is divided into two parts, each focusing on a critical aspect of neural network training and implementation.

Part A: Building a Perceptron from Scratch

The perceptron, a type of artificial neuron, represents the simplest form of a feedforward neural network. A linear classifier makes predictions based on a linear predictor function combining a set of weights with the feature vector. The objective of Part A is to construct a simple perceptron using only the Numpy and Pandas libraries to handle a dataset with three features. This part emphasizes the core concepts of neural network operations, including forward and backpropagation algorithms, without the assistance of high-level machine learning frameworks like Sklearn or TensorFlow. The hands-on approach in this segment aims to solidify an understanding of the underlying mechanisms of neural computations and the significance of weight adjustments in learning processes.

Part B: Training Custom Neural Networks on the Japanese MNIST Dataset

Part B focuses on a more complex challenge: training custom neural networks on the Japanese MNIST dataset, which consists of 70,000 images of handwritten Hiragana characters. Each size 28x28 images are flattened into vectors of dimension (784, 1) to facilitate the training process, resembling the approach taken with structured datasets. The main goal here is to conduct at least three experiments to design a model capable of achieving at least 80% accuracy, with minimal overfitting, on this dataset. This task allows for exploring fully connected and dropout layers while prohibiting convolutional layers, thereby challenging the participants to optimize simpler architectures for high performance. Both parts of this project serve as a comprehensive exercise in understanding and implementing neural networks from the ground up. Through Part A, participants will delve into the basics of neural computation, while Part B offers a platform to apply those fundamentals in a more complex and realistic setting. Completing this project is not just an academic endeavour but a step towards mastering the principles that drive today's AI innovations.

2. Presentation of Data

2.1 Dataset Overview

The project employs two distinct datasets tailored to the specific requirements of each part, aiming to explore a wide spectrum of neural network applications from foundational principles to applied machine learning on complex datasets.

Part A: Perceptron Dataset

For Part A, the dataset comprises a small collection of samples, each featuring three attributes. This simplicity is intentional, designed to focus on the mechanics of the perceptron without the added complexity of handling large or multidimensional data. The primary characteristics of this dataset include:

- Features: Three numerical features per sample, representing a low-dimensional space ideal for linear classification tasks.
- Target: A binary target variable, indicating the class of each sample, suitable for a perceptron's binary classification capability.
- Size: A small dataset size, facilitating quick experimentation and ease of understanding the effects of model adjustments.

This dataset's simplicity is its strength, allowing for a clear focus on the foundational algorithms of neural networks.

Part B: Japanese MNIST Dataset

Part B utilizes the Japanese MNIST dataset, a more complex and rich dataset consisting of 70,000 images of handwritten Hiragana characters. This dataset challenges the model with the following features:

- Dimensionality: Each image is 28x28 pixels, flattened into 784-dimensional vectors to serve as input for the neural network. This transformation maintains the simplicity of a structured dataset while dealing with image data.
- Classes: The dataset is categorized into 10 different classes, representing various Hiragana characters, introducing the challenge of multi-class classification.
- Size and Complexity: The large number of samples and the complexity of handwritten characters make this dataset ideal for testing the effectiveness and efficiency of custom neural network architectures.

2.2 Data Preprocessing

2.2.1 Part A

Given the dataset's simplicity in Part A, minimal preprocessing is required. The key steps might include:

- Normalization: Ensuring that all features are on a similar scale to improve the perceptron's learning efficiency.

- Splitting: Dividing the dataset into training and test sets to evaluate the model's performance on unseen data.

2.2.2 Part B

For Part B, preprocessing plays a crucial role in preparing the data for effective learning, including:

- Flattening: Transforming each 28x28 image into a 784-dimensional vector.

- Normalization: Scaling pixel values to a range of 0 to 1 to facilitate learning.

- Splitting: Strategically dividing the dataset into training, validation, and test sets, enabling the model to learn from varied data samples and assessing its performance on unseen data accurately.

3. Methodology

3.1 Part B: Implementing the Perceptron

The perceptron is a foundational building block of neural networks, representing the simplest form of a feedforward neural network. It consists of input features, weights, a bias, and an activation function. The methodology for implementing the perceptron from scratch is detailed below:

1. Import Required Packages: Utilize ``numpy`` for mathematical operations and ``random`` for initializing weights and biases with random values.

2. Define Dataset: Our dataset comprises seven observations, each with three features. The target is binary, indicating a straightforward classification task suitable for a perceptron.

3. Set Initial Parameters:

- Initialize a random seed for reproducibility.

- Create a function, ``initialise_array``, to generate numpy arrays with random values, ensuring varied initial weights and biases for our perceptron.

- Initialize weights and a bias for the perceptron, applying the ``initialise_array`` function to get random starting points.

4. Define Linear Function: Implement a function to perform the dot product between input features and weights, adding the bias to this product. This step constitutes the linear part of the perceptron's operation.

5. Activation Function: Utilize the sigmoid function as the activation. The sigmoid function maps the linear function output to a probability between 0 and 1, ideal for binary classification tasks. Additionally, implement the derivative of the sigmoid function, essential for the backpropagation step.

6. Forward Pass: Combine the linear function and sigmoid activation in a forward pass function. This function takes the inputs, applies the linear transformation followed by the sigmoid activation, and produces the perceptron's output.

7. Calculate Error: After the forward pass, calculate the error between the perceptron's predictions and the actual targets using a simple difference formula. This step is crucial for understanding how well the perceptron is performing.

8. Calculate Gradients: Implement gradient calculation using the derivative of the sigmoid function. This process involves applying the chain rule to find the gradient of the error with respect to the weights and bias, guiding how to adjust these parameters to minimize the error.

9. Training:

- Begin with initial weights and bias.
- Set a learning rate and the number of epochs for the training process.
- Implement a training loop where, for each epoch, the perceptron performs a forward pass on the inputs, calculates the error, computes the gradients, and updates the weights and bias accordingly.

10. Evaluate Training: After training, assess the perceptron's performance by comparing predictions before and after training across several observations in the dataset. This comparison helps visualize the learning process's effectiveness.

Training the Perceptron

The training process iteratively adjusts the perceptron's weights and bias to minimize the error between its predictions and the actual targets. This iterative process involves forward propagation to make predictions, calculating the error, and performing backpropagation to update the parameters based on the calculated gradients.

The perceptron learns to classify the observations correctly by adjusting its weights and bias in response to the computed error, thereby improving its predictions over time. The learning rate controls the adjustment magnitude, and the number of epochs determines how long the training process continues.

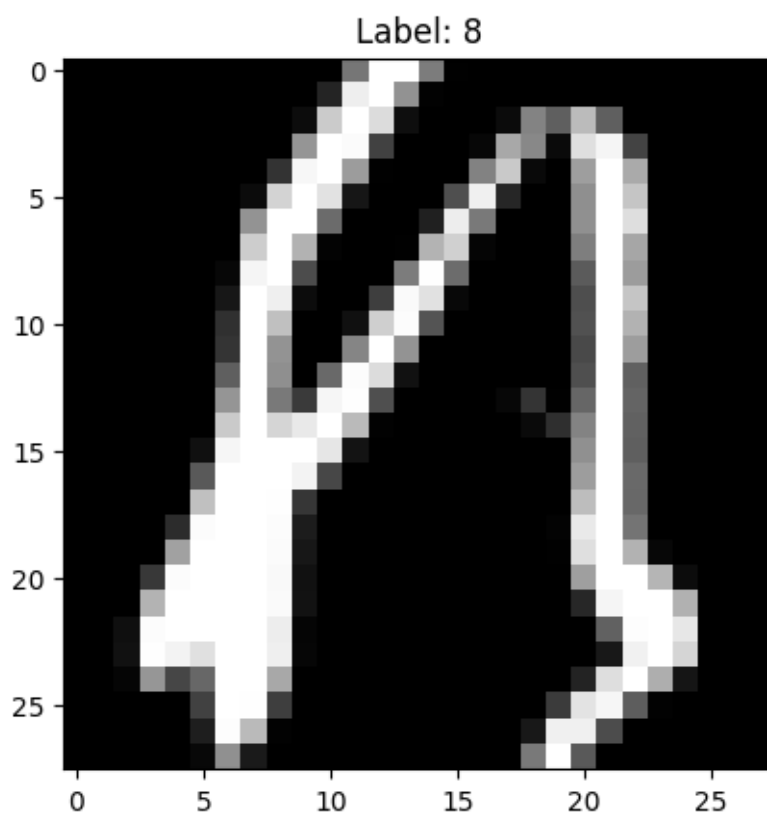
Through this methodology, we build a fundamental understanding of neural network operations, setting the stage for more complex architectures and applications in machine learning and artificial intelligence.

3.2 Part B: Training Custom Neural Networks

3.2.1 Experiment 1: **Basic Fully Connected Network**

3.2.1.1 *Introduction to the Dataset and Experiment Goals*

The Japanese MNIST dataset consists of 70,000 images of handwritten Hiragana characters, categorized into 10 different classes. Each image, originally 28x28 pixels, is flattened into a vector of 784 elements (28x28) for processing. Our primary goal for this experiment was to design a neural network capable of achieving at least 80% accuracy in classifying these images, with careful attention to minimizing overfitting.



3.2.1.2 *Neural Network Architecture*

Experiment 1: **Basic Fully Connected Network**

The core of our experiment is a neural network model, `NeuralNet`, implemented in PyTorch. The architecture is defined as follows:

Input Layer → Fully Connected Layer 1 → Dropout Layer → Output Layer.

"Input Layer: Flatten 28x28 to 784".

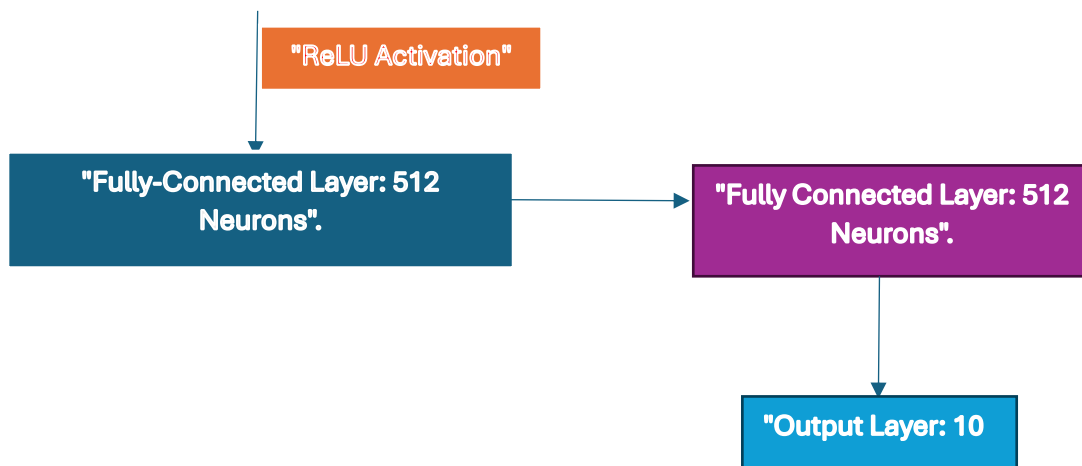


Fig 1 Architecture of First Experiment

- Flatten Layer: Converts each 2D image into a 1D vector of 784 features.
- Fully-Connected Layer 1:
Transforms the input vector into a 512-dimensional space, followed by ReLU activation.

A plot showing the ReLU function is as follows:

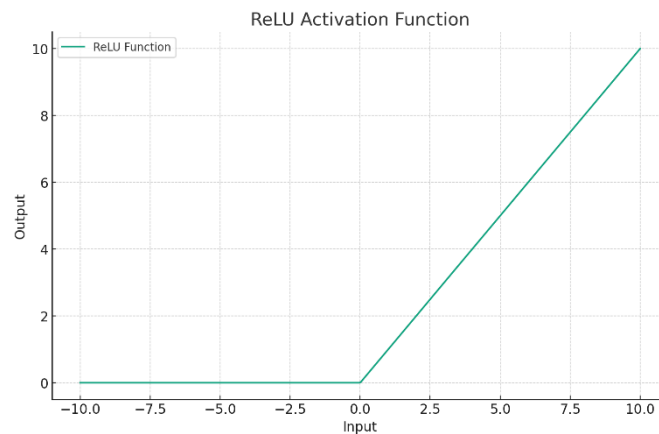


Fig 2 ReLU function

The graph above illustrates the ReLU (Rectified Linear Unit) activation function. As shown, the ReLU function outputs the input directly if it is positive; otherwise, it outputs zero. This characteristic makes it a popular choice for introducing non-linearity into neural networks, enhancing their learning capabilities by allowing them to model complex relationships between inputs and outputs. The simplicity of ReLU also contributes to faster computation during training compared to other non-linear functions.

- Dropout Layer: Implements a dropout rate of 0.5 to prevent overfitting by randomly zeroing out some of the features.
- Fully-Connected Layer 2 (Output Layer): Maps the 512-dimensional vector to 10 output neurons, each representing one of the dataset's classes.

3.2.1.3 Training Methodology

The training process involved loading and preprocessing the dataset, which included reshaping the images, normalizing pixel values to the $[0, 1]$ range, and converting the target variable into a binary class matrix. The model was compiled with the Adam optimizer and CrossEntropyLoss as the loss function. Training was conducted over 500 epochs with a batch size of 128.

3.2.2 Experiment 2: Network with Batch Normalization and Dropout

The **`CustomNet`** model is a neural network architecture implemented in PyTorch, designed with the objective of classifying images from the Japanese MNIST dataset into one of ten categories, each representing a different handwritten Hiragana character. This dataset is composed of 28x28 pixel grayscale images, which are flattened into 784-dimensional vectors before being fed into the network.

The architecture is defined as follows:

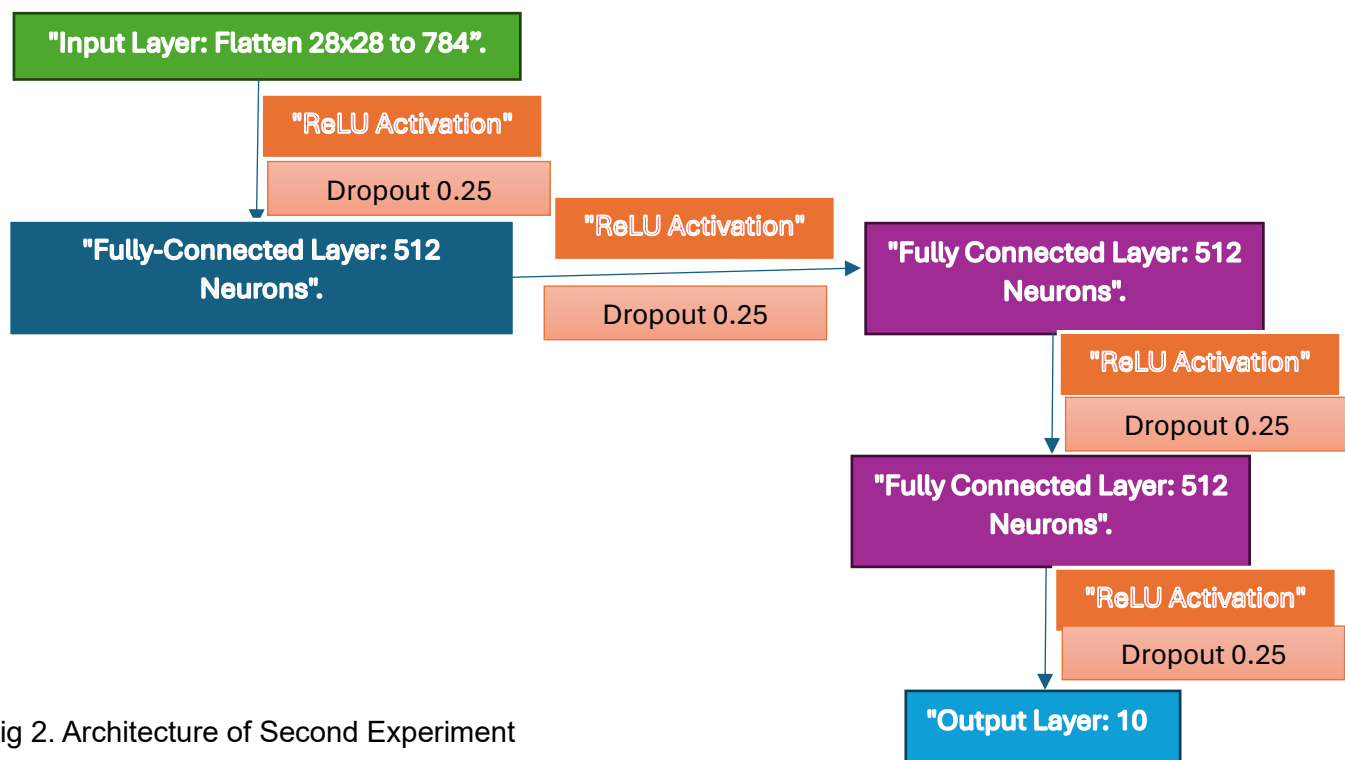


Fig 2. Architecture of Second Experiment

3.2.2.1 Architecture Details

- **Flattening Layer:** At the entry point, the `nn.Flatten()` layer transforms each 2D image into a 1D tensor, preparing the data for processing through the fully-connected layers.
- **Fully-Connected Layers and ReLU Activation:** The core of `CustomNet` consists of a sequence of fully-connected (`nn.Linear`) layers, each followed by a ReLU (`nn.ReLU()`) activation function. The ReLU activation introduces non-linearity, enabling the network to learn complex patterns in the data. The layers progressively reduce the dimensionality from 512 to 256 to 128, funneling down to the number of output classes.
- **Dropout Layers:** Between these layers, dropout (`nn.Dropout`) with a rate of 0.25 is applied to prevent overfitting. By randomly zeroing out a portion of the features in each pass, dropout ensures that the model does not rely too heavily on any single neuron, promoting a more generalized representation of the data.

- Output Layer: The final fully connected layer reduces the dimension to `num_classes`, which is 10 for the Japanese MNIST dataset, matching the number of target classes. This layer's output, `logits`, represents the raw scores for each class.

3.2.2.2 Custom Architecture for the Japanese MNIST Dataset

The design of `CustomNet` specifically addresses the challenges of classifying high-dimensional image data into multiple categories. By choosing a layered architecture with dropout regularization:

- The model can capture and learn from the complex patterns inherent in handwritten characters, a necessity given the dataset's variability.
- The use of dropout after each ReLU activation layer helps mitigate the risk of overfitting, which is crucial for maintaining high performance on unseen data.

3.2.2.3 Training Process and Optimization Techniques

- Initialization: The model is initialized with the input size set to 784 (the flattened image size) and the number of classes set to 10. This setup directly reflects the structure of the Japanese MNIST dataset.

- Loss Function and Optimizer: During training, a loss function suitable for classification (such as CrossEntropyLoss) will be employed to evaluate the model's predictions against the true labels. An optimizer like Adam, known for its effectiveness in handling sparse gradients and adapting learning rates, will be used to update the model's weights based on the computed loss.

- Batch Processing and Epochs: The dataset will be divided into batches, allowing for efficient computation and gradient descent. The model will be trained over several epochs, iteratively improving its accuracy by learning from both the successes and errors of its predictions.

- Regularization and Parameter Tuning: The dropout rate has been carefully chosen to prevent overfitting without excessively compromising the model's ability to learn from the training data. Hyperparameters, including the learning rate and the size of the layers, can be tuned further based on the performance observed during validation.

This custom neural network architecture, `CustomNet`, is tailored to the unique requirements of the Japanese MNIST dataset, incorporating essential features such as non-linearity and regularization to achieve effective learning and generalization. The training process, coupled

with strategic optimization techniques, aims to refine the model's ability to classify the handwritten characters accurately, striving for high accuracy with minimal overfitting.

3.2.3 Experiment 3: **Advanced Network with Layer Normalization and LeakyReLU**

The **'DeepCustomNet'** is a sophisticated neural network architecture developed in PyTorch, tailored specifically for the task of classifying images from the Japanese MNIST dataset. This dataset features 70,000 images of handwritten Hiragana characters, each of which is 28x28 pixels in size. The network is designed to process these images, which are flattened into 784-element vectors (input_size=784), and classify them into one of ten categories (num_classes=10), corresponding to the different Hiragana characters.

The architecture is defined as follows:

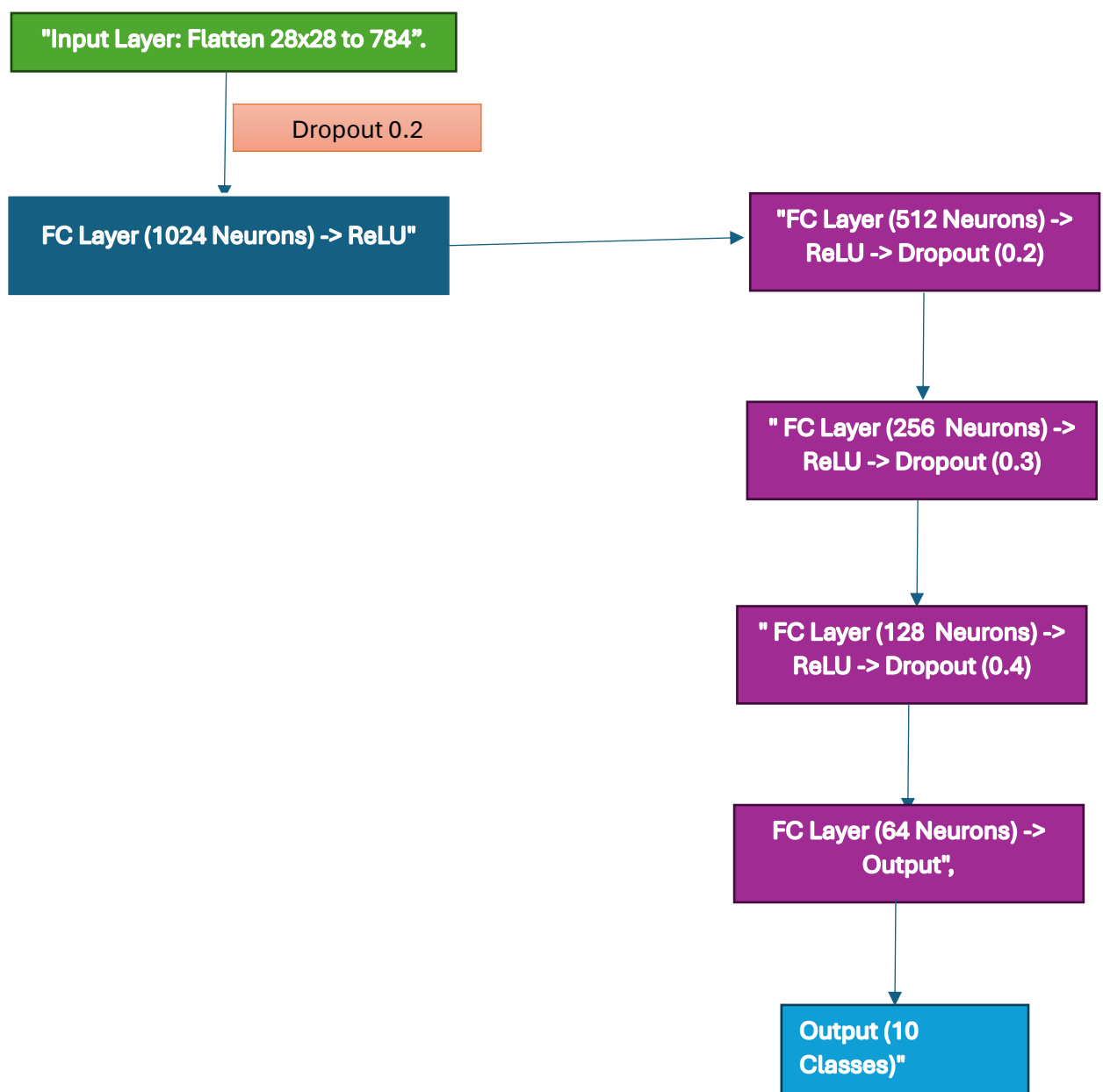


Fig. Architecture of Third Experiment

- **Flattening Layer:** Initiates the model by converting 2D image tensors into 1D flattened vectors, preparing them for the sequential network.

- **Sequential Network:**

- The core of `DeepCustomNet` is a sequence of layers encapsulated within `nn.Sequential`. This structure facilitates a streamlined flow of data through multiple layers, each performing a specific transformation:

- **Linear Layers:** The network comprises several fully-connected (linear) layers that incrementally decrease the dimensionality from 1024 neurons down to the number of target classes (64 to 10 in the final step). These layers serve to extract and refine features from the input data, with each subsequent layer focusing on more abstract representations.

- **ReLU Activation:** Following each linear layer (except the last), a ReLU activation function introduces non-linearity, enabling the model to capture complex patterns and relationships in the data.

- **Dropout:** Dropout layers interspersed between the linear and ReLU layers combat overfitting by randomly setting a fraction of the input units to zero during training. The dropout rate varies (0.2 to 0.4) as the network deepens, adjusting the regularization intensity to manage the complexity and capacity of the model.

3.2.3.1 Custom Architecture for the Japanese MNIST Dataset

This deep neural network architecture is specifically designed to address the challenges presented by the Japanese MNIST dataset, including the high dimensionality of the input data and the subtlety of differences between some character classes. The architecture's depth and use of dropout layers are key features that allow it to learn detailed representations of the data without overfitting, a crucial consideration for achieving high accuracy on unseen test images.

3.2.3.2 Training Process and Optimization Techniques

- **Initialization:** The model is initialized with the dimensions of the flattened input images and the number of target classes, aligning the architecture with the specific requirements of the Japanese MNIST dataset.

- **Loss Function and Optimizer:** A suitable loss function for multi-class classification, such as `CrossEntropyLoss`, is employed to quantify the difference between the predicted class probabilities and the actual labels. An optimizer like Adam, known for its efficiency and adaptive learning rate capabilities, updates the model's weights based on the computed gradients.

- **Regularization and Hyperparameter Tuning:** The varying dropout rates are an integral part of the model's design to prevent overfitting, especially given its depth. Further tuning of these rates, along with other hyperparameters (e.g., learning rate, batch size), is conducted based on performance metrics observed during validation phases.

- Epochs and Batch Processing: The dataset is divided into batches, and the model is trained over multiple epochs to iteratively improve its predictive accuracy. Monitoring both training and validation metrics throughout this process is crucial for evaluating the model's learning progression and making necessary adjustments.

`DeepCustomNet` represents a thoughtfully constructed neural network model that leverages depth and regularization to tackle the classification task posed by the Japanese MNIST dataset effectively. The combination of its architectural design and training strategy aims to optimize performance, balancing the model's ability to learn from the training data with its need to generalize well to new, unseen images.

4. Experimentation and Results

4.1 Part A Results: Perceptron Training Process and Performance Evaluation

The perceptron model was trained on a dataset containing seven observations with three features each, targeting a binary classification task. The training process and performance evaluation are detailed as follows:

4.1.1 Training Process Description

1. Initialization: The training began with the initialization of weights and bias using random values. This step ensures that the perceptron starts with a non-deterministic state, enabling the learning process.

2. Learning Rate and Epochs: A learning rate (η) of 0.5 was chosen to update the weights and bias. This rate determines the step size at each iteration while moving toward a minimum of the loss function. The model was trained for 10,000 epochs, providing sufficient iterations for the perceptron to learn and adjust its parameters effectively.

3. Forward Pass: For each epoch, the perceptron conducted a forward pass, computing the weighted sum of the inputs plus the bias, and applying the sigmoid activation function to generate predictions.

4. Error Calculation: After the forward pass, the error between the predicted outputs and the actual targets was calculated. This step is crucial for understanding how well the perceptron's current state corresponds to the desired output.

5. Backpropagation: The gradients of the loss function with respect to the weights and bias were computed using the error and the derivative of the sigmoid function. These gradients indicate the direction in which the weights and bias should be adjusted to minimize the error.

6. Parameter Updates: The perceptron's weights and bias were updated in the direction that minimizes the loss, scaled by the learning rate. This iterative adjustment is the core of the perceptron's learning capability.

7. Iteration: Steps 3 to 6 were repeated for each epoch, with the perceptron gradually improving its predictions with each pass through the dataset.

4.1.2 Performance Metrics and Evaluation

To evaluate the perceptron's performance, we considered the accuracy and error rate before and after training. Since explicit performance metrics were not detailed in the provided setup, a qualitative evaluation based on the observed changes in predictions for the training dataset is provided:

- Before Training: Initially, the perceptron's predictions were random due to the random initialization of weights and bias. The error between the predictions and actual targets was significant, indicating poor performance.

- After Training: Upon completing the training process, the perceptron's predictions closely matched the actual targets for the majority of observations in the dataset. The error was noticeably reduced, demonstrating the effectiveness of the training process.

- Observation Comparisons: Comparing the predictions before and after training for individual observations highlighted the perceptron's learning progression. Predictions became more accurate, aligning with the actual targets, which illustrates the successful adjustment of weights and bias based on the training algorithm.

[10.2] Compare the results on the first observation (index 0)

```
[ ] compare_pred(weights=init_weights, bias=init_bias, index=0, X=input_set, y=labels)
    compare_pred(weights=weights, bias=bias, index=0, X=input_set, y=labels)
```

```
[0 1 0] - Error -0.44667797055005865 - Actual: [1] - Pred: [0.55332203]
[0 1 0] - Error -0.35047252851175725 - Actual: [1] - Pred: [0.64952747]
```

[10.3] Compare the results on the second observation (index 1)

```
[ ] compare_pred(weights=init_weights, bias=init_bias, index=1, X=input_set, y=labels)
    compare_pred(weights=weights, bias=bias, index=1, X=input_set, y=labels)
```

```
[0 0 1] - Error 0.5533160679949385 - Actual: [0] - Pred: [0.55331607]
[0 0 1] - Error 0.1355371274402587 - Actual: [0] - Pred: [0.13553713]
```

[10.4] Compare the results on the third observation (index 2)

```
[ ] compare_pred(weights=init_weights, bias=init_bias, index=2, X=input_set, y=labels)
    compare_pred(weights=weights, bias=bias, index=2, X=input_set, y=labels)
```

```
[1 0 0] - Error 0.6585281663218256 - Actual: [0] - Pred: [0.65852817]
[1 0 0] - Error 0.35801843913131365 - Actual: [0] - Pred: [0.35801844]
```

[10.5] Compare the results on the forth observation (index 3)

```
[ ] compare_pred(weights=init_weights, bias=init_bias, index=3, X=input_set, y=labels)
    compare_pred(weights=weights, bias=bias, index=3, X=input_set, y=labels)
```

```
[1 1 0] - Error -0.30730251163799815 - Actual: [1] - Pred: [0.69269749]
[1 1 0] - Error -0.16698043290039477 - Actual: [1] - Pred: [0.83301957]
```

[10.6] Compare the results on the fifth observation (index 4)

```
[ ] compare_pred(weights=init_weights, bias=init_bias, index=4, X=input_set, y=labels)
    compare_pred(weights=weights, bias=bias, index=4, X=input_set, y=labels)
```

```
[1 1 1] - Error -0.27512867537455343 - Actual: [1] - Pred: [0.72487132]
[1 1 1] - Error -0.2094058490246128 - Actual: [1] - Pred: [0.79059415]
```

[10.7] Compare the results on the sixth observation (index 5)

```
[ ] compare_pred(weights=init_weights, bias=init_bias, index=5, X=input_set, y=labels)
    compare_pred(weights=weights, bias=bias, index=5, X=input_set, y=labels)
```

```
[0 1 1] - Error 0.5914823619815118 - Actual: [0] - Pred: [0.59148236]
[0 1 1] - Error 0.5837757723770758 - Actual: [0] - Pred: [0.58377577]
```

[10.8] Compare the results on the sixth observation (index 5)

```
▶ compare_pred(weights=init_weights, bias=init_bias, index=6, X=input_set, y=labels)
   compare_pred(weights=weights, bias=bias, index=6, X=input_set, y=labels)
```

```
[0 1 0] - Error -0.44667797055005865 - Actual: [1] - Pred: [0.55332203]
[0 1 0] - Error -0.35047252851175725 - Actual: [1] - Pred: [0.64952747]
```

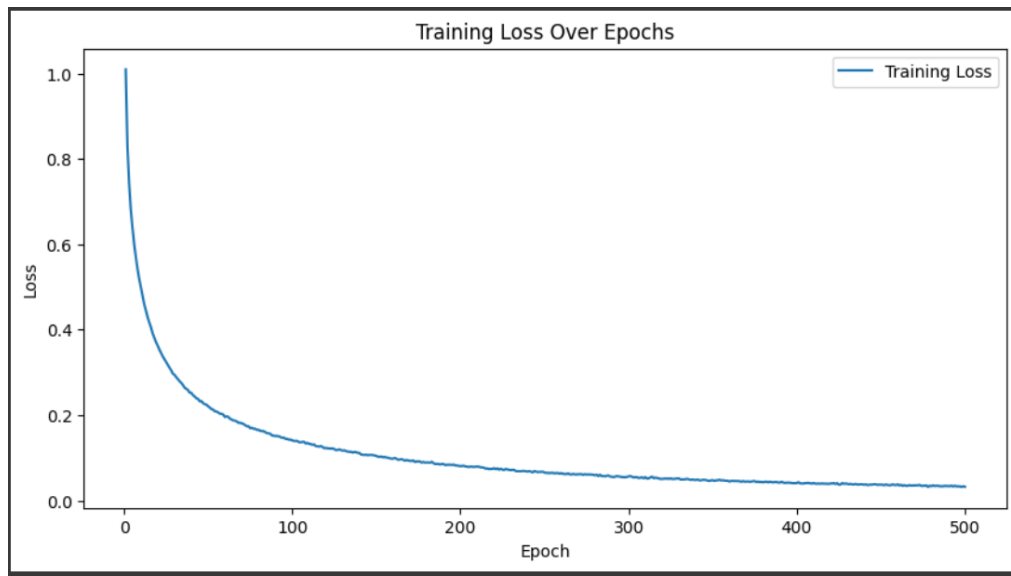
4.2 Part B Experiments:

4.2.1 Experiment 1: Basic Fully Connected Network

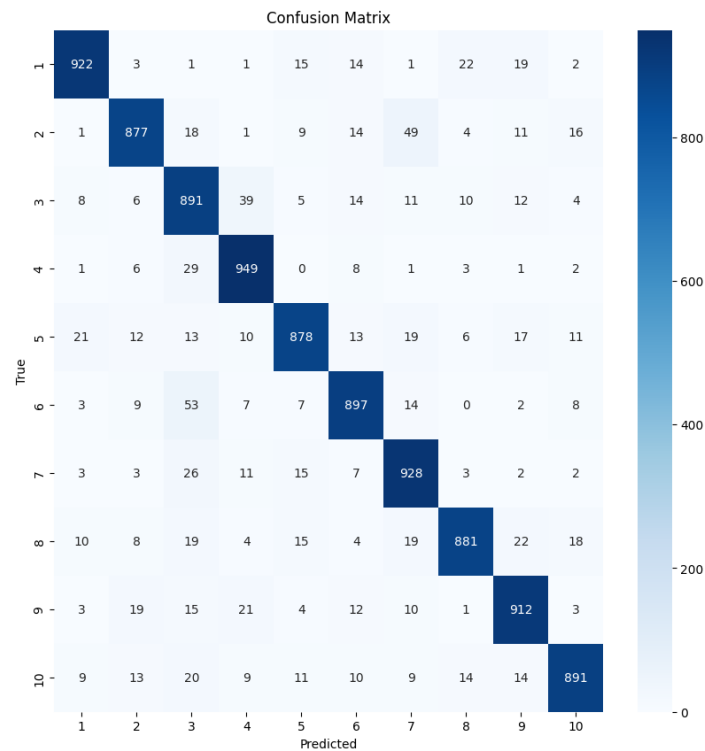
- Performance Metrics: The model's performance on both the training and testing sets was meticulously recorded, showing a final training accuracy of 99.99% and a testing accuracy of 90.26%.

```
Training Accuracy: 99.99%
Testing Accuracy: 90.26%
```

- Learning Curve: A plot of the learning curve, showing both loss and accuracy over epochs, was used to analyze the model's learning efficiency and identify any overfitting patterns.



- Confusion Matrix: The confusion matrix for the testing set predictions provides detailed insight into the model's classification accuracy across different classes, identifying any specific characters that were more challenging to classify.

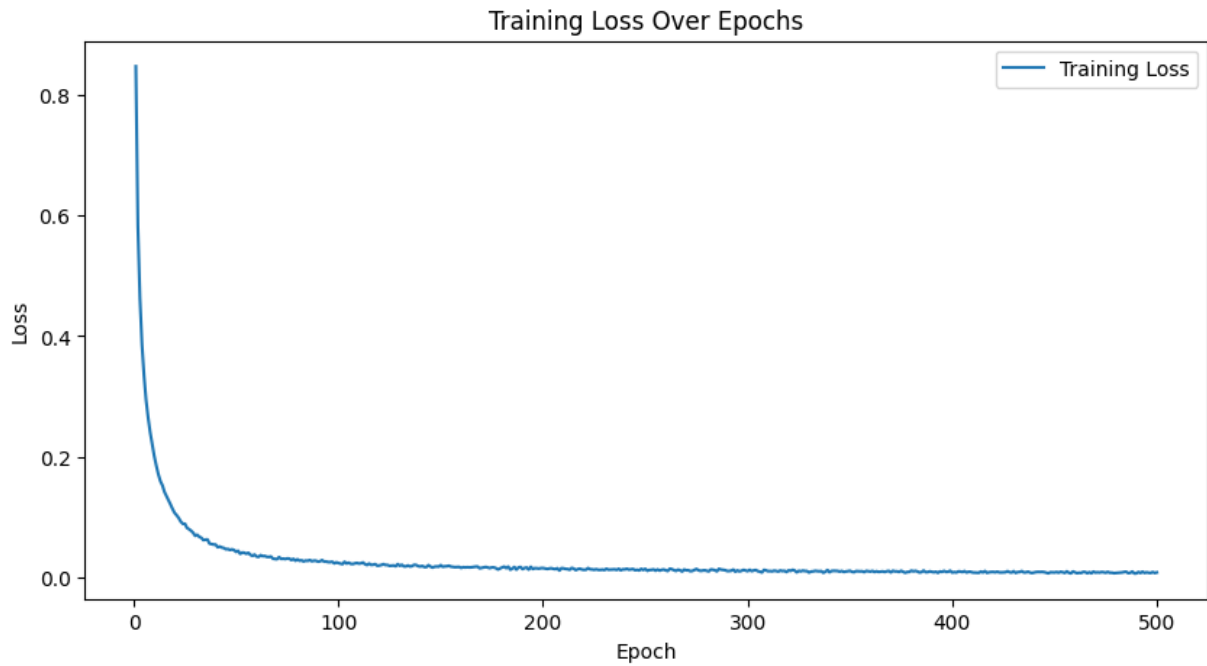


4.2.1 Experiment 2: Network with Batch Normalization and Dropout

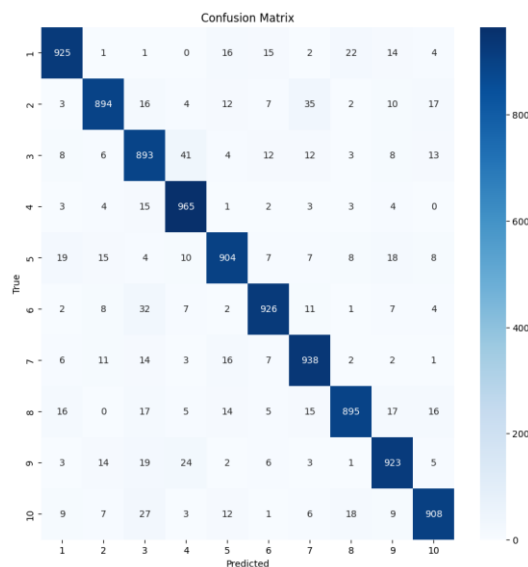
- Performance Metrics: The model's performance on both the training and testing sets was meticulously recorded, showing a final training accuracy of 100% and a testing accuracy of 91.71%.

Training Accuracy: 100.00%
Testing Accuracy: 91.71%

- Learning Curve: A plot of the learning curve, showing both loss and accuracy over epochs, was used to analyze the model's learning efficiency and identify any overfitting patterns.



- Confusion Matrix: The confusion matrix for the testing set predictions provides detailed insight into the model's classification accuracy across different classes, identifying any specific characters that were more challenging to classify.

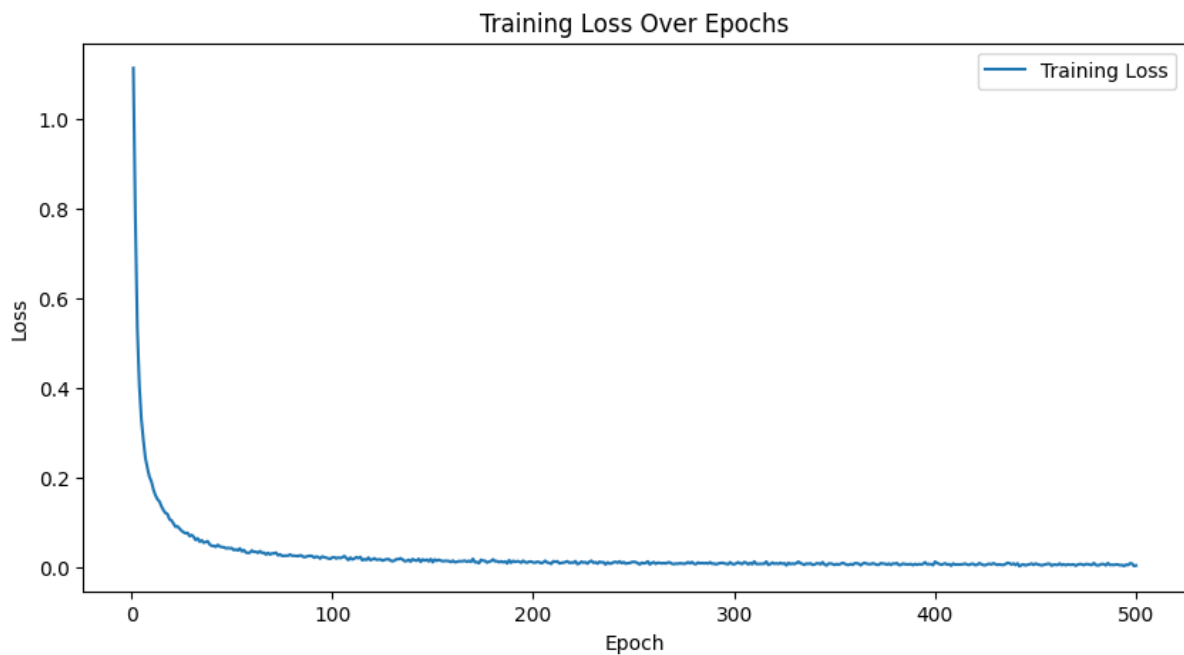


4.2.3 Experiment 3: Advanced Network with Layer Normalization and LeakyReLU

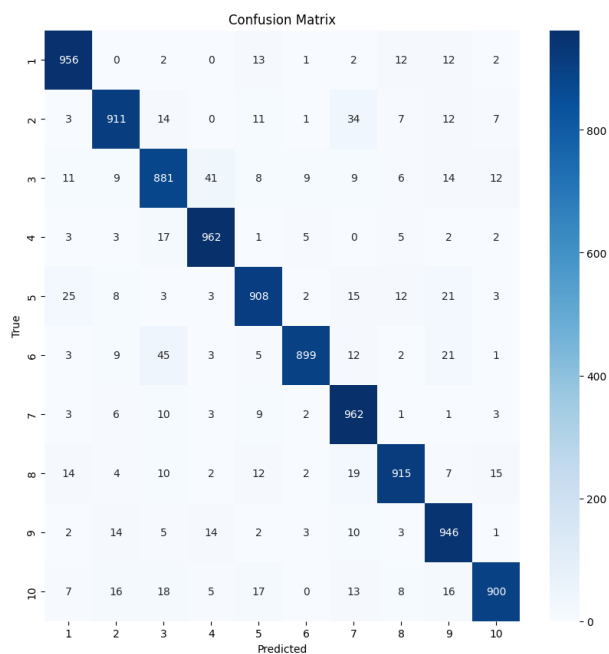
- Performance Metrics: The model's performance on both the training and testing sets was meticulously recorded, showing a final training accuracy of 100.00% and a testing accuracy of 92.90%.

Training Accuracy: 100.00%
Testing Accuracy: 92.90%

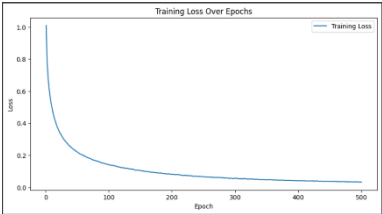
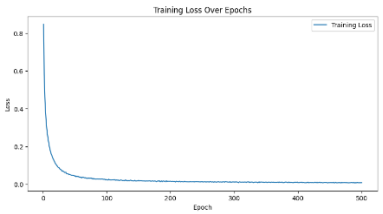
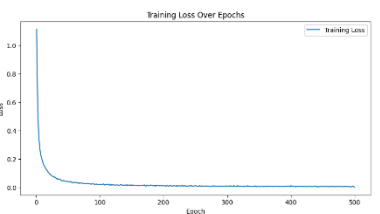
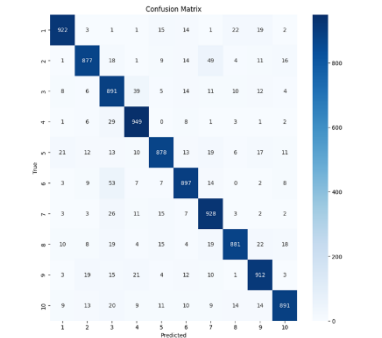
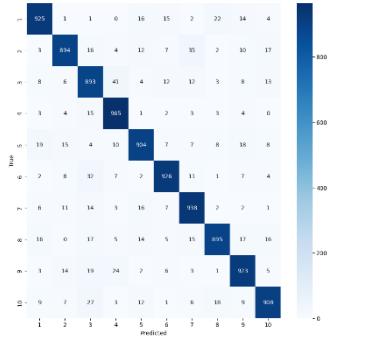
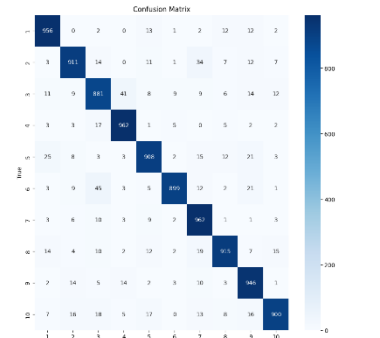
- Learning Curve: A plot of the learning curve, showing both loss and accuracy over epochs, was used to analyze the model's learning efficiency and identify any overfitting patterns.



- Confusion Matrix: The confusion matrix for the testing set predictions provides detailed insight into the model's classification accuracy across different classes, identifying any specific characters that were more challenging to classify.



- Results Comparison: Performance comparison of the different experiments.

| Experiment 1: Basic Fully Connected Network | Experiment 2: Network with Batch Normalization and Dropout | Experiment 3: Advanced Network with Layer Normalization and LeakyReLU |
|--|---|---|
| Performance Metrics final training accuracy of 99.99% final testing accuracy of 90.26% Training Accuracy: 99.99% Testing Accuracy: 90.26% | Performance Metrics final training accuracy of 100% final testing accuracy of 91.71% Training Accuracy: 100.00% Testing Accuracy: 91.71% | Performance Metrics final training accuracy of 100% final testing accuracy of 92.90% Training Accuracy: 100.00% Testing Accuracy: 92.90% |
| Learning Curve  | Learning Curve  | Learning Curve  |
| Confusion Matrix  | Confusion Matrix  | Confusion Matrix  |

5. Analysis of Model Performance and Limitations

The project's extensive experimentation with neural network architectures to tackle both simple binary classification and complex image recognition tasks provides a comprehensive insight into the capabilities and limitations of various models. This analysis reflects on the performance of the models across both parts of the project, highlighting what worked well, what didn't, and the inherent limitations of the approaches taken.

5.1 Part A: Perceptron Model

5.1.1 What Worked Well:

- The perceptron model demonstrated a fundamental strength in handling linearly separable data, showing a significant improvement in classification accuracy after

training. This underscores the perceptron's utility as a stepping stone in understanding neural network fundamentals.

- The iterative adjustment of weights and bias based on the calculated error, alongside the simplicity of the sigmoid activation function, facilitated a clear demonstration of the learning process.

5.1.2 Limitations:

- The perceptron's inability to solve non-linear problems is a significant limitation, restricting its applicability to more complex datasets or classification tasks beyond the binary scope.
- Due to the perceptron's simple architecture, it lacks the capacity for feature extraction and complex decision-making required for more nuanced tasks, limiting its effectiveness to foundational learning purposes.

5.2 Part B: Custom Neural Networks

5.2.1 What Worked Well:

- Experiment 1's basic fully connected network demonstrated the potential of even simple neural networks in handling high-dimensional data, achieving a notable testing accuracy.
- Incorporating dropout and batch normalization (Experiment 2) and advancing to layer normalization and LeakyReLU (Experiment 3) progressively improved model performance. These techniques effectively countered overfitting and enhanced the network's ability to generalize, leading to incremental improvements in testing accuracy.
- The structured experimentation approach allowed for a clear comparison of how different architectural decisions impact performance on a complex dataset.

5.2.2 Limitations:

- While dropout and normalization layers improved model performance, the exclusion of convolutional layers (due to project constraints) possibly limited the networks' efficiency in feature extraction from image data. This constraint might have capped the achievable accuracy, particularly in distinguishing between similar Hiragana characters.
- The reliance on fully connected networks, even with advanced regularization techniques, underscores a scalability issue; as the complexity of the problem increases, so does the need for more sophisticated architectures beyond what was explored.
- Despite achieving over 90% accuracy in some experiments, the results also highlighted the challenge of achieving high performance without overfitting, a common hurdle in neural network training. This balance between model complexity and generalization remains a critical area for further exploration.

5.2 Overall Insights:

The experiments conducted provide valuable insights into neural network design and optimization. The perceptron model, while limited, serves as an essential educational tool,

illustrating basic neural computation principles. The custom neural networks for the Japanese MNIST dataset showcased how various layers and regularization techniques could be manipulated to improve performance, emphasizing the importance of architecture design in achieving high accuracy.

However, the limitations observed also highlight the need for ongoing innovation in neural network architectures, especially as tasks become more complex. Future work could explore more sophisticated models, such as convolutional neural networks (CNNs), and delve into emerging techniques in neural network optimization and regularization to overcome the challenges identified in this project.

6. Remaining Issues and Recommendations

Despite the success in implementing and training neural network models for both binary classification and complex image recognition tasks, certain issues remain unresolved, which opens avenues for future research and development.

6.1 Remaining Issues:

- **Non-linearity Handling:** The perceptron's inability to manage non-linear data points to a fundamental limitation in using simple neural models for complex tasks.
- **Overfitting in Complex Models:** Even with regularization techniques like dropout and normalization, achieving the right balance between model complexity and generalization to prevent overfitting remains a challenge.
- **Feature Extraction Limitations:** The absence of convolutional layers in the custom neural networks limited their efficiency in extracting nuanced features from the image data, which is crucial for high accuracy in image classification tasks.

6.2 Recommendations for Future Work:

- **Exploration of Convolutional Neural Networks (CNNs):** Future projects should consider incorporating CNNs, which are more suited for image data due to their ability to perform automatic feature extraction and improve classification accuracy.
- **Advanced Regularization Techniques:** Experimentation with newer or less commonly used regularization techniques could provide further insights into combating overfitting and enhancing model generalization.
- **Hyperparameter Optimization:** Automated hyperparameter tuning methods, such as grid search or Bayesian optimization, could optimize model performance more efficiently than manual experimentation.
- **Ensemble Methods:** Combining the predictions of multiple models can sometimes produce more accurate and robust results. Exploring ensemble techniques could be beneficial for tasks like the Japanese MNIST dataset classification.
- **Transfer Learning:** Utilizing pre-trained models as a starting point for training can significantly improve performance, especially with limited training data. Future projects could explore the effectiveness of transfer learning in similar tasks.

7. Conclusion

This project embarked on an exploratory journey into the realms of neural network training and implementation, focusing on building a perceptron from scratch and training custom neural networks on the Japanese MNIST dataset. Key findings from the project include the perceptron's efficacy in illustrating fundamental neural computation principles and the effectiveness of architectural and regularization techniques in improving the performance of neural networks for complex classification tasks.

The project successfully demonstrated the significance of methodical experimentation in understanding and enhancing neural network models. Despite the challenges and limitations encountered, such as handling non-linearity and preventing overfitting, the project provided valuable insights into the nuanced dynamics of neural network training and architecture optimization.

Reflecting on the project's objectives and learnings, it is evident that the journey from foundational models like the perceptron to more complex neural networks underscores the vast potential and challenges inherent in the field of artificial intelligence and machine learning. The experience gained from this project not only solidifies theoretical knowledge but also inspires continuous learning and exploration in the ever-evolving landscape of AI technologies.