Date of Submission Sunday, August 27, 2023.

Digital Object Identifier

Comparative Analysis of Neural Network Architectures on MNIST Dataset: Using Numpy vs PyTorch Modules

PRAJESH SHRESTHA[1], and UJJWAL PAUDEL[1]
[1]Department of Electronics and Computer Engineering, Thapathali Campus, Institute of Engineering, Tribhuvan University, Kathmandu, Nepal (e-mail: prajesh.762417@thc.tu.edu.np, ujjwal.762417@thc.tu.edu.np)

ABSTRACT Artificial Neural Network (ANN) is a powerful computational model inspired by the human brain's neural connections. The study focuses on applying ANNs for multi-class classification, using the MNIST dataset to categorize handwritten digits. ANNs are particularly adept at multi-class classification due to their capability to identify intricate patterns within data, allowing for effective recognition of diverse features across various classes. Through thorough experimentation and model refinement, this study resulted in a remarkable achievement: the best-performing ANN model achieved an accuracy of 91%. By leveraging ANNs' capacity to learn from vast data, this study explores their potential in classifying handwritten digits within the MNIST dataset.

INDEX TERMS Artificial Neural Networks (ANNs), Classification, Handwritten Digits, MNIST

I. INTRODUCTION

ARTIFICIAL Neural Network (ANN) have emerged as a robust computational paradigm inspired by the intricate connectivity of neurons in the human brain. ANNs consist of interconnected nodes, or neurons, organized in layers that process information, enabling them to learn complex patterns and relationships within data. These networks excel in tasks such as pattern recognition, classification, and regression by iteratively adjusting their parameters to minimize prediction errors. ANNs have seen remarkable advancements, particularly with deep learning architectures, allowing them to tackle intricate problems across various domains, from image and speech recognition to natural language processing.

The MNIST dataset, a cornerstone of machine learning benchmarks, encapsulates a collection of handwritten digits. The MNIST dataset serves as an essential playground for evaluating classification algorithms due to its simplicity and relevance. With their ability to capture intricate patterns and hierarchies, ANNs can differentiate unique features within each handwritten digit, enabling accurate categorization.

II. METHODOLOGY

A. ARTIFICIAL NEURAL NETWORKS (ANNS)

1) Biological Neuron

The concept of Artificial Neural Networks (ANNs) takes inspiration from the intricate operations of the human brain. In this biological marvel, neurons form networks that communicate by transmitting signals across synapses. Neurons receive input signals, process them within their cell bodies, and generate output signals that travel through dendrites and dendrons to other neurons. This process of interconnected signaling allows the brain to learn and recognize patterns, making it a fascinating model for computational systems.

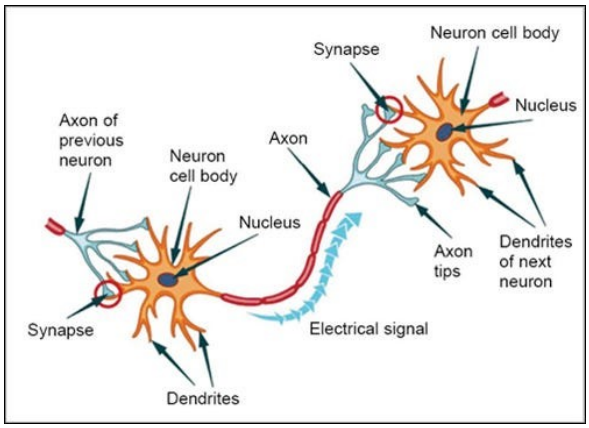

FIGURE 1. Biological Neuron

2) Mathematical Model of a Neuron

A perceptron is a simplified model of a neuron often used as a building block in ANNs. It takes multiple input values, applies weights to them, calculates a weighted sum, adds a bias

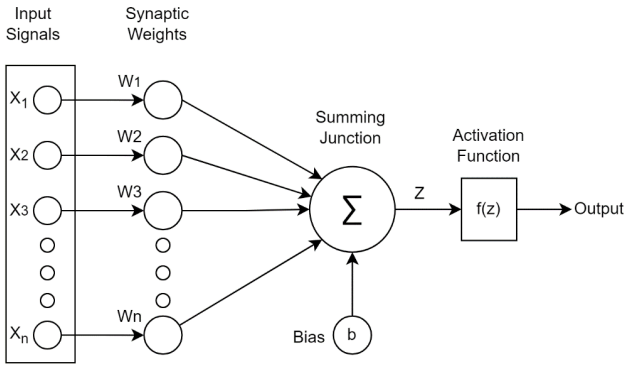

FIGURE 2. Mathematical Model of a Neuron

term, and then passes the result through an activation function to produce an output. Let's denote the input signals as $x_1, x_2, \ldots, x_n$, the corresponding weights as $w_1, w_2, \ldots, w_n$, and the bias as b. The weighted sum of inputs and the bias is calculated as:

$$z = \sum_{i=1}^{n} (x_i \cdot w_i) + b \tag{1}$$

The resulting value of z is passed through an activation function $f(z)$.

3) Activation Functions

Activation functions determine whether a neuron should be activated or not based on the weighted input it receives. Activation functions introduce non-linearity and, bound and define a neuron's output. Common activation functions include:

1. Rectified Linear Unit (ReLU):

$$f(z) = \max(0, z) \tag{2}$$

2. Sigmoid:

$$f(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

3. Hyperbolic Tangent (Tanh):

$$f(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{4}$$

4. Softmax (for multi-class classification):

$$f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{5}$$

4) Feed Forward Neural Network

A multi-layered perceptron (MLP) comprises multiple interconnected perceptron's. The output of one layer serves as input to the next, creating hierarchies that allow ANNs to learn complex patterns. An ANN consists of an input layer, hidden layers, and an output layer.

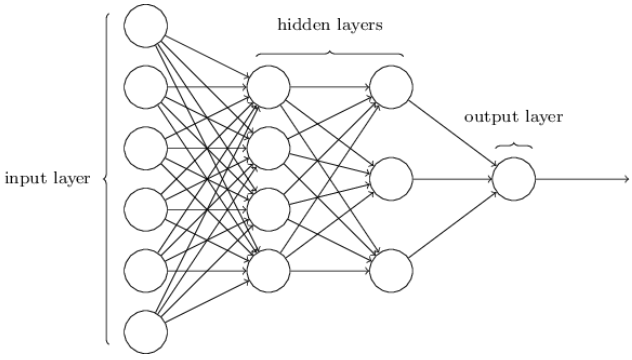

FIGURE 3. Multi-Layered Perceptron

5) Training ANNs

The weight and biases of the ANN must be updated through training mechanism which enables the neural to adapt, optimize, and enhance its predictive capabilities over time. Training an Artificial Neural Network (ANN) involves a two-fold process: forward propagation and backward propagation.

During forward propagation, input data is fed into the network's input layer. Neurons in subsequent layers calculate weighted sums of their inputs and pass the results through activation functions, introducing non-linearity. This sequential processing continues until the output layer generates predictions. Following forward propagation, the predicted output is compared to the actual target output, revealing the error. For multi-class classification, the cost function that is used is categorical cross entropy: The Categorical Cross-Entropy Loss formula is given by:

$$L = -\sum_{i=1}^{K} y_i \cdot \log(p_i) \tag{6}$$

Where:

L represents the Categorical Cross-Entropy Loss.

K is the number of classes in the classification task.

y_i is the true probability of the example belonging to class i.

p_i is the predicted probability of the example belonging to class i.

Backward propagation comes next, where the network learns from its mistakes. Gradients of the error with respect to the weights and biases are computed using the chain rule of calculus, indicating how much each parameter contributed to the error. These gradients guide the adjustment of weights and biases using optimization algorithms like gradient descent, aiming to minimize the error. By iteratively fine-tuning these parameters over multiple epochs, the network refines its ability to make accurate predictions.

In this dynamic process, the network's predictions gradually align with the target outputs, allowing it to learn patterns and relationships within the data. As the iterative adjustments continue, the network's performance improves, ultimately leading to a trained model that can generalize its learning to new, unseen data.

B. SYSTEM BLOCK DIAGRAM

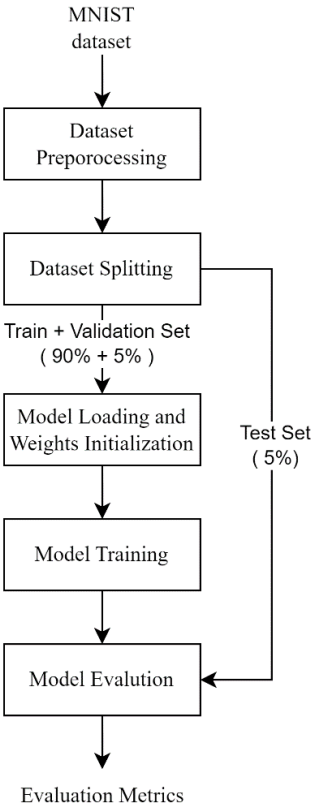

FIGURE 4. System Block Diagram

1) Dataset Preprocessing

In the dataset preprocessing step, the dataset underwent analysis and transformation to ensure it was in a suitable format to be fed as input to the Artificial Neural Network (ANN).

2) Dataset Splitting

In the dataset preprocessing step, the dataset underwent analysis and transformation to ensure it was in a suitable format to be fed as input to the Artificial Neural Network (ANN).

3) Model Loading and Weights Initialization

In the Loading Model (Initialization) phase, the neural network model is loaded and its weights and biases were initialized . Techniques such as Xavier and Kaiming initialization are employed to set the initial weights of the neurons in the network.

- Xavier Initialization: Xavier initialization assigns initial weights to neurons in a way that helps prevent vanishing and exploding gradients during training. It takes into account the number of input and output connections for each neuron and ensures that the variance of the activations remains consistent across layers.
- Kaiming Initialization: Specifically designed for Rectified Linear Unit (ReLU) activations, Kaiming initialization adapts the weight initialization to the properties of the activation function. It aims to maintain appropriate signal flow and gradient propagation in networks using ReLU activations, leading to faster convergence and improved training.

4) Model Training

During this phase, the initialized neural network is trained using the training dataset. The network's weights are iteratively adjusted using backpropagation and optimization algorithms to minimize the loss function. This process allows the model to learn patterns and relationships in the data.

5) Model Evaluation

After training, the model's performance is assessed using the testing dataset. It involves making predictions on the testing data and comparing them to the actual target values. Evaluation metrics such as accuracy, precision, recall, and F1-score help gauge how well the model generalizes to new, unseen data.

C. INSTRUMENTATION TOOLS

1) Pandas and Numpy

- Pandas is employed for data processing, providing data structures like DataFrames that facilitate data manipulation and analysis.
- NumPy serves as the foundation for linear algebra operations, enabling efficient handling of arrays and matrices.

2) Matplotlib and Plotly

- Matplotlib and Plotly are visualization libraries utilized to create informative and visually appealing graphs, charts, and plots, aiding in data exploration and presentation.
- Matplotlib offers various plotting capabilities, while Plotly provides interactive and dynamic visualizations.

3) Seaborn

- Seaborn complements Matplotlib by providing a higher-level interface for statistical graphics, making it easier to create aesthetically pleasing and informative plots.

4) Scikit-learn

- Sklearn offers a comprehensive suite of preprocessing and post-processing modules to streamline data preparation and evaluation processes.
- Preprocessing modules include LabelEncoder, StandardScaler, MinMaxScaler, RobustScaler, and OrdinalEncoder, enabling feature scaling and encoding categorical variables.
- Evaluation metrics like accuracy, precision, recall, F1-score, and confusion matrix are available from metrics module to assess model performance.

5) PyTorch

- PyTorch's 'torchvision' and 'transforms' modules were used for data preprocessing and augmentation, the 'Dat-

aLoader' class facilitated efficient dataset loading, the 'nn' module allowed neural network architecture definition, and the 'optim' module provided optimization algorithms for weight updates during training.

III. RESULTS AND DISCUSSIONS

A. EXPERIMENTS AND RESULTS

The experimental neural architectures designed for training and testing on the MNIST dataset were intentionally diversified by exploring various dimensions. This included altering the number of hidden layers and adjusting the number of neurons within each of these layers. Additionally, we examined the impact of different activation functions, specifically Rectified Linear Unit (ReLU), Softmax, Sigmoid, and Tanh.

In our experimentation, ReLU, Tanh, and Sigmoid activations were applied to the inner hidden layers of the neural network. However, Softmax activation was exclusively utilized at the output layer, reflecting its common use for multi-class classification tasks.

Another crucial aspect of our experiments was the initialization of weights and biases. We employed three different initialization methods: random uniform, Xavier, and Kaiming initialization. These initialization techniques played a pivotal role in influencing the network's convergence and performance.

Furthermore, we investigated the use of dropout layers in some of our experiments. Dropout was introduced not as a corrective measure for overfitting but rather to observe its effects. This allowed us to gain insights into how dropout influences the model's training dynamics and generalization capabilities, shedding light on its role in neural network optimization

TABLE 1. Experiment Details

Exp. No.	Neural Layers	Initializer	Activation Function	Dropout
1	[784, 10, 10]	Random	Relu, Softmax	No
2	[784, 10, 10]	Kaiming	Relu, Softmax	No
3	[784, 10, 10]	Xavier	Relu, Softmax	No
4	[784, 10, 10]	Random	Sigmoid, Softmax	No
5	[784, 10, 10]	Random	Tanh, Softmax	No
6	[784, 20, 10]	Random	Relu, Softmax	No
7	[784, 20, 10]	Kaiming	Relu, Softmax	No
8	[784, 20, 10]	Xavier	Relu, Softmax	No
9	[784, 10, 10]	Random	Sigmoid, Softmax	No
10	[784, 20, 10]	Random	Tanh, Softmax	No
11	[784, 10, 10, 10]	Random	Relu, Softmax	No
12	[784, 20, 10, 10]	Kaiming	Sigmoid, Softmax	No
13	[784, 5, 10, 10]	Xavier	Tanh, Softmax	No
14	[784, 20, 20, 10]	Random	Relu, Softmax	No
15	[784, 10, 20, 10]	Random	Relu, Softmax	No
16	[784, 20, 10, 10]	Xavier	Relu, Softmax	Yes
17	[784, 10, 10, 10]	Xavier	Sigmoid, Softmax	Yes
18	[784, 20, 20, 10]	Kaiming	Relu, Softmax	Yes
19	[784, 20, 10]	Random	Sigmoid, Softmax	Yes
20	[784, 30, 10]	Xavier	Relu, Softmax	Yes

Table 1 presents the results of twenty distinct experiments, each encompassing a unique combination of the aforementioned options. In the 'Neural Layers' column, the array denotes the number of neurons within each layer of the neural network. The first layer represents the input layer, comprising 784 neurons to correspond with the number of pixels in each data instance. Meanwhile, the final layer serves as the output layer, featuring 10 neurons, each responsible for digit classification spanning from 0 to 9.

TABLE 2. Experiment Results

Exp. No.	Last Training Accuracy	Last Validation Accuracy
1	83.86%	84.71%
2	87.08%	87.62%
3	89.63%	89.43%
4	10.69%	10.10%
5	11.28%	10.90%
6	87.42%	86.71%
7	92.00%	90.67%
8	91.49%	89.76%
9	10.93%	10.95%
10	11.72%	12.38%
11	67.07%	72.43%
12	16.40%	15.48%
13	12.46%	11.59%
14	11.18%	10.43%
15	15.26%	13.42%
16	21.52%	23.76%
17	36.43%	37.48%
18	74.02%	73.10%
19	70.70%	70.14%
20	91.48%	90.86%

Table 2 showcases the final training and validation accuracies achieved at the conclusion of each experiment, following 2500 training iterations. These results offer valuable insights into the performance dynamics of diverse neural network configurations.

Foremost among our findings is the remarkable performance of Experiment Number 7, which delivered a remarkable training accuracy of 92%, accompanied by an impressive validation accuracy of 90.67%, positioning it at the pinnacle of our results. In stark contrast, Experiment Number 4 emerged as the least successful, with a mere 10.69% training accuracy.

Upon scrutinizing the results, a conspicuous pattern emerges: the choice of activation function in the hidden layers, particularly sigmoid or tanh, has a pronounced influence on both training and model performance. A notable exception to this trend is evident when comparing Experiment Number 4 and Experiment Number 19. Both employ sigmoid as the activation function, yet the former records the lowest performance metrics while the latter achieves an accuracy of 70.70%. The distinguishing factor between these experiments is the number of neurons in the hidden layers; Experiment Number 19 deploys twice the number of neurons compared to Experiment Number 4. This leads us to infer that sigmoid activation tends to underperform when the number of neurons in the hidden layer is limited, but exhibits superior performance when neurons are abundant.

Expanding our observations to the influence of hidden layer count, we find that for sigmoid activation, increasing

the number of hidden layers, as seen in Experiment Number 12, detrimentally affects model performance.

Conversely, when employing the hyperbolic tangent (tanh) activation function, we observe a consistent drop in performance whether we increase or decrease the number of neurons in the hidden layers.

In the context of weight and bias initializations, both Kaiming and Xavier initialization methods consistently outperform random initialization. Moreover, ReLU activation in the hidden layers emerges as the most effective setting, with every configuration employing ReLU yielding top-notch results. Therefore, a combination of ReLU activation with either Kaiming or Xavier initialization consistently delivers superior performance.

In the final segment of our experiments, spanning from the 16th to the 20th, we introduced dropout layers into our models. While dropout is typically employed as a regularization technique to mitigate overfitting, our usage was primarily for observational purposes, aiming to gain insights into its effects on model behavior.

Experiment 11 with a deep neural network [784, 10, 10, 10] had a high training accuracy (67.07%) but exceptional validation accuracy (72.43%), indicating that depth can benefit the model's performance. In contrast, Experiment 13, with a shallower network [784, 5, 10, 10], had a relatively lower training accuracy (12.46%) and validation accuracy (11.59%), suggesting that a very shallow architecture may not capture complex patterns as effectively. Experiment 14, with wider layers [784, 20, 20, 10], showed a training accuracy of 11.18% and a validation accuracy of 10.43%, indicating that increasing the width of layers may not necessarily lead to better performance.

Neural networks rely on gradient-based optimization techniques, like backpropagation, to learn from data. During training, gradients are computed and propagated backward through the network to adjust the weights. However, the initial values of weights can have a significant impact on this process. Randomly initialized weights may lead to two critical problems: vanishing gradients and exploding gradients. Vanishing gradients occur when gradients become exceedingly small as they propagate backward through the network. This can result in slow convergence or even halt training altogether. Conversely, exploding gradients happen when gradients grow rapidly, leading to instability in training. These issues are primarily caused by inappropriate weight.

Xavier initialization, also known as Glorot initialization, addresses these problems by considering the number of input and output neurons in a layer. When initializing weights, Xavier carefully selects values from a suitable distribution, often Gaussian or uniform. This method ensures that the variance of activations remains relatively consistent across layers. By doing so, Xavier initialization mitigates the vanishing and exploding gradient challenges, promoting smoother convergence during training.

Kaiming initialization is tailored for activation functions like ReLU. It focuses on combating the vanishing gradient problem specifically associated with ReLU units. Kaiming initialization takes into account the fan-in, which represents the number of input neurons. By scaling the initial weights with the square root of the fan-in, Kaiming initialization maintains a balanced distribution of activations, reducing the likelihood of gradients becoming too small during backpropagation.

Proper weight initialization significantly impacts the pace of convergence during training. When weights are initialized closer to suitable values, the neural network can start learning meaningful patterns sooner. This results in faster convergence, meaning that the network reaches an acceptable level of performance with fewer training iterations. Additionally, it often leads to better generalization, where the network performs well on unseen data. In contrast, random initialization lacks the insight into the network's architecture or the characteristics of activation functions. This can lead to inconsistent weight distributions, making it more challenging for the network to learn effectively during training.

The preference for ReLU (Rectified Linear Unit) activation over tanh and sigmoid activations in these experiments can be attributed to several critical factors. Firstly, the vanishing gradient problem plagues tanh and sigmoid functions. These functions tend to saturate, meaning that their derivatives become extremely close to zero when inputs are either very large or very small. In deep neural networks, this leads to vanishing gradients during backpropagation, causing weight updates to become minimal and hindering the training process, especially in the early layers. Secondly, tanh and sigmoid activations can give rise to "dead neurons" in the network. When a neuron consistently outputs values close to zero, it effectively becomes inactive and stops learning because the gradients near zero prevent meaningful weight updates. In deep networks, this phenomenon can result in a substantial portion of neurons contributing very little to the learning process, diminishing the network's capacity to model intricate data patterns. Thirdly, both tanh and sigmoid functions exhibit a limited output range. Sigmoid squashes inputs to the (0, 1) range, while tanh maps inputs to (-1, 1). This limitation can lead to saturation, with most neurons outputting values close to extremes, which restricts the range of representable features. ReLU, in contrast, offers an unbounded output range, enabling it to capture a broader spectrum of features and adapt more effectively to the complexity of the data. Whereas ReLU activation brings advantages in terms of sparsity and non-linearity. It introduces sparsity by only activating for positive inputs, allowing the network to create a piecewise linear approximation to the data. This non-linearity and sparsity can be highly advantageous for modeling complex functions, whereas sigmoid and tanh activations tend to produce smoother, less sparse representations.

The decrease in performance with increasing network depth can be attributed to several key factors. Firstly, as the depth of the neural network grows, the gradients during backpropagation can either vanish or explode, especially when using certain activation functions like sigmoid or tanh. This

phenomenon makes it challenging for the model to effectively update weights, hampering learning. Additionally, deeper networks are more prone to overfitting, as they have a higher capacity to memorize the training data, potentially capturing noise and leading to poor generalization on unseen data.

The increase in performance with wider networks or a higher number of neurons can be attributed to several key reasons. Firstly, wider networks have a greater capacity to capture complex patterns and representations within the data. This increased capacity allows the model to learn more intricate features and relationships, potentially enhancing its ability to discriminate between classes or categories. With more neurons, the network can effectively model both low-level and high-level features, making it more adaptable to the data's inherent complexity.

Secondly, wider networks mitigate the risk of underfitting. Underfitting occurs when the model is too simple to capture the underlying patterns in the data. A network with too few neurons may struggle to represent the complexity of the data adequately, resulting in poor performance. By increasing the width of the network, we provide it with additional expressive power, reducing the likelihood of underfitting and enabling it to fit the training data more effectively.

Figure 5 offers a comprehensive view of the training and validation loss curves across all three distinct experiments. What's particularly intriguing here is the intertwining nature of the training and validation curves in each experiment. This intriguing observation suggests that, in all three models, the complexity of the model architecture played a pivotal role in governing the relationship between the training and validation datasets. The intertwining of these curves is indicative of the model's ability to adapt to the training data as it becomes more complex. Essentially, as the model's complexity increases, it becomes better at fitting the training data, resulting in lower training loss. However, this heightened complexity can also lead to overfitting, where the model becomes excessively tailored to the training data and struggles to generalize to unseen data, causing the validation loss to rise.

Figure 6 displays accuracy plots, mirroring the previously observed trend across most experiments, where increasing model complexity leads to improved training accuracy but potential overfitting. However, a notable exception is the worst-case scenario, characterized by erratic fluctuations in both training and validation accuracy curves, suggesting underlying issues that require thorough investigation. This anomaly underscores the need for a closer examination of convergence, hyperparameters, and data quality to address this instability and enhance model performance.

B. EVALUATION ON TEST SET

The dataset has been meticulously divided into three segments, following a 95:5:5 ratio. This partitioning strategy reserves the final 5% of the data exclusively for the evaluation of the performance metrics associated with three distinct experiments. Table 3 illuminates the classification report of the top-performing experiment. Impressively, this experiment achieved an overall accuracy of 91.69% when assessing the model's proficiency across all digit classes.

TABLE 3. Classification Report: (BEST) Experiment Number 7

Class	Precision	Recall	F1-Score	Support
0	0.97	0.94	0.95	217
1	0.98	0.94	0.96	253
2	0.88	0.90	0.89	218
3	0.90	0.89	0.90	217
4	0.92	0.91	0.92	193
5	0.87	0.93	0.90	198
6	0.95	0.96	0.95	183
7	0.89	0.90	0.89	208
8	0.91	0.89	0.90	192
9	0.88	0.89	0.88	221
Accuracy			0.91	2100
Macro Avg	0.91	0.91	0.91	2100
Weighted Avg	0.92	0.91	0.91	2100

In heatmap depicted in Figure 7, we gain further insights into the model's strengths and areas for improvement. Notably, the model demonstrates exceptional proficiency in recognizing digits 0 and 1. This high accuracy can be attributed to the clear and distinctive structural characteristics inherent to both 0 and 1. Their unambiguous shapes and patterns make them relatively easier for the model to classify accurately.

However, it's crucial to highlight that the model faces a slight challenge when confronted with digit 5. This challenge arises due to the inherently more intricate structural features associated with digit 5 compared to the relatively straightforward shapes of digits 0 and 1. The nuanced variations and complexities within the representation of digit 5 can occasionally pose a hurdle for the model's accurate classification.

TABLE 4. Classification Report: (MIDDLE) Experiment Number 18

Class	Precision	Recall	F1-Score	Support
0	0.82	0.82	0.82	209
1	0.92	0.83	0.87	272
2	0.66	0.83	0.73	179
3	0.76	0.69	0.72	237
4	0.71	0.64	0.67	210
5	0.65	0.62	0.64	220
6	0.84	0.80	0.82	194
7	0.78	0.77	0.78	213
8	0.61	0.66	0.64	173
9	0.64	0.73	0.68	193
Accuracy			0.74	2100
Macro Avg	0.74	0.74	0.74	2100
Weighted Avg	0.75	0.74	0.74	2100

Table 4 provides a snapshot of the classification report from a moderate experiment, yielding a test accuracy of 74%. Notably, the model's performance varies across different digit classes. It excels in accurately classifying digits 0 and 1 but faces challenges with the 5th digit class. Figure 8 illustrates the associated confusion matrix, shedding light on the model's struggles not only with digit 5 but also with digits 2 and 9. In contrast, it performs exceptionally well with the 1st digit. It's worth considering that the high support count for

class 1, totaling 272 instances, could contribute to the model's robust accuracy across all cases. This observation underscores the influence of class distribution on model performance, emphasizing the need for balanced datasets and further investigation into improving classification for the underrepresented classes.

When examining the classification report presented in Table 5, it becomes apparent that the worst-performing experiment stands out with a mere 12% overall accuracy. This dismal performance can be largely attributed to the model's ability to accurately detect only digit 1, where it achieves a modest 21% F1-Score. The corresponding heatmap, depicted in Figure 9, visually represents this underwhelming performance.

TABLE 5. Classification Report: (WORST) Experiment Number 4

Class	Precision	Recall	F1-Score	Support
0	0.00	0.00	0.00	193
1	0.99	0.12	0.21	256
2	0.00	0.00	0.00	217
3	0.00	0.00	0.00	207
4	0.00	0.00	0.00	209
5	0.00	0.00	0.00	197
6	0.02	0.33	0.03	189
7	0.00	0.00	0.00	202
8	0.00	0.00	0.00	220
9	0.00	0.00	0.00	210
Accuracy			0.12	2100
Macro Avg	0.10	0.04	0.02	2100
Weighted Avg	0.99	0.12	0.21	2100

Figures 10, 11, and 12 each depict the Precision-Recall Curves for the three primary models. In the case of the best model, its PR-curve exhibits excellence, though not perfection, across all digits. This suggests strong precision and recall values, but acknowledges some room for improvement in achieving flawless performance.

In the case of moderate model, its PR-curve exhibits a slight departure from the ideal curve, indicating some variability in precision and recall across different digits. However, it remains relatively stable compared to the worst model. Conversely, the worst model's PR-curve demonstrates more erratic behavior. This inconsistency highlights substantial challenges in precision and recall, particularly when distinguishing between different digits.

C. PYTORCH IMPLEMENTATION

In contrast to the previous experiments coded from scratch with the numpy module, we explored the advantages of using established deep learning frameworks like TensorFlow, Keras, and PyTorch for constructing neural networks, including both dense and convolutional architectures. To facilitate a meaningful comparison, we meticulously recreated the architecture of our best model, [784, 20, 10], using PyTorch's sequential modeling APIs. This PyTorch implementation featured Kaiming initialization for weight initialization, Rectified Linear Units (ReLU) as the activation function for hidden layers, and categorical cross-entropy as the loss function.

Figure 13 and Figure 14 offer insights into the performance of these PyTorch-implemented models. The accuracy plot showcases an initial rapid increase followed by a more stable rise, indicating the models' ability to learn effectively. In contrast, the training loss curve exhibits erratic fluctuations, primarily due to the use of batch sampling during training, with a BATCH_SIZE of 16 leading to occasional spikes in losses. Notably, the validation curve appears smooth, as validation evaluations occurred after every 25th iteration rather than in every iteration, contributing to its stability.

TABLE 6. Classification Report: PyTorch Implementation

Class	Precision	Recall	F1-Score	Support
0	0.97	0.96	0.96	193
1	0.96	0.99	0.98	256
2	0.99	0.94	0.96	217
3	0.91	0.96	0.93	207
4	0.94	0.93	0.93	209
5	0.93	0.92	0.92	197
6	0.96	0.98	0.97	189
7	0.98	0.98	0.98	202
8	0.94	0.93	0.94	220
9	0.94	0.91	0.93	210
Accuracy			0.95	2100
Macro Avg	0.95	0.95	0.95	2100
Weighted Avg	0.95	0.95	0.95	2100

Table 6 presents the classification report derived from the PyTorch-implemented model's evaluation on the test set, revealing an outstanding accuracy of 95.14%. This accuracy significantly surpasses the performance achieved by the numpy-based scratch implementation. In Figure 15, the accompanying confusion matrix further underscores the model's prowess, showcasing high prediction scores across all digit classes.

The underperformance of the neural architecture implemented from scratch, in comparison to the PyTorch implementation, can be attributed to several compelling reasons. Firstly, PyTorch is built specifically for deep learning tasks and provides a high-level framework with pre-built functions for neural network components. It offers optimized GPU support, automatic differentiation, and efficient backpropagation, which significantly expedite the training process. In contrast, coding a neural network from scratch using numpy entails manually implementing these critical components, which can be prone to errors and inefficiencies, leading to longer training times and suboptimal results. Secondly, PyTorch leverages hardware acceleration through CUDA, allowing the neural network to harness the full potential of GPUs. This hardware acceleration can result in substantial speedups during training, especially for complex models. In contrast, a numpy-based implementation lacks this level of hardware optimization, making it challenging to achieve comparable training efficiency and, consequently, performance.

Another key factor is the availability of well-optimized loss functions and activation functions in PyTorch, such as categorical cross-entropy and ReLU activation, which are essential for effective training. In contrast, when implementing

from scratch, designing and fine-tuning these functions can be complex and error-prone, potentially leading to suboptimal model behavior and performance. Furthermore, PyTorch offers convenient tools for batch processing and validation, as evident in the discussion where it allowed for smoother training and evaluation on the validation set. The numpy-based implementation may lack such features, leading to erratic training curves and suboptimal convergence during training.

IV. CONCLUSION

In conclusion, this report explores the realm of artificial neural networks (ANNs), delving into their architecture and application. Through hands-on exploration and experimentation, the report focused on training a Feed Forward ANN to tackle multi-class classification using the MNIST dataset. By implementing the ANN, varying hidden layers, activation functions, and weight initialization techniques, we achieved a commendable accuracy of 91%. The study shows intricacies of building, training, and optimizing neural networks, showcasing their strength in finding complex patterns and achieving high-performance classification tasks.

PRAJESH S. is an enthusiastic undergraduate student pursuing Computer Engineering at Thapathali College, Institute of Engineering (IOE), Nepal. Curious about the vast world of Machine Learning and Artificial Intelligence, Prajesh enjoys immersing himself in learning and experimenting with diverse ML/AI models. He is fascinated by the potential applications of these technologies and aspires to utilize them for addressing real-world challenges. For any queries or potential collaborations, you can contact Prajesh at prajesh.762417@thc.tu.edu.np.

UJJWAL P. is a dedicated undergraduate student currently pursuing Computer Engineering at Thapathali College, Institute of Engineering (IOE), Nepal. With a keen interest in Machine Learning and Artificial Intelligence, he embraces the role of a passionate learner, constantly exploring and experimenting with various ML/AI models. Driven by a passion to make meaningful contributions to the AI field, he is committed to continuous growth and learning in this exciting and dynamic domain. For any inquiries or potential collaborations, you can reach out to Ujjwal at ujjwal.762416@thc.tu.edu.np.

APPENDIX A
PLOTS AND FIGURES

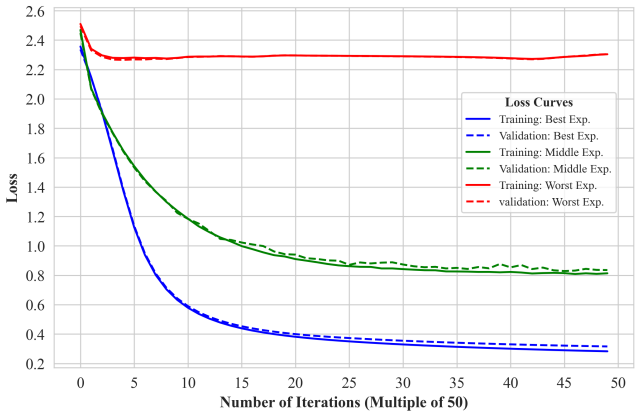

FIGURE 5. Major Experiments Training vs Validation Loss Graph

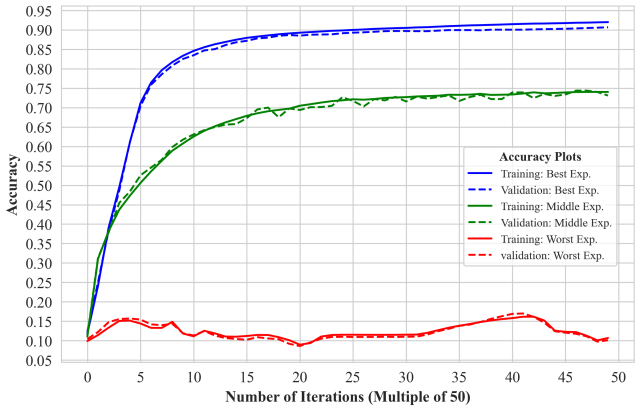

FIGURE 6. Major Experiments Training vs Validation Accuracy Plot

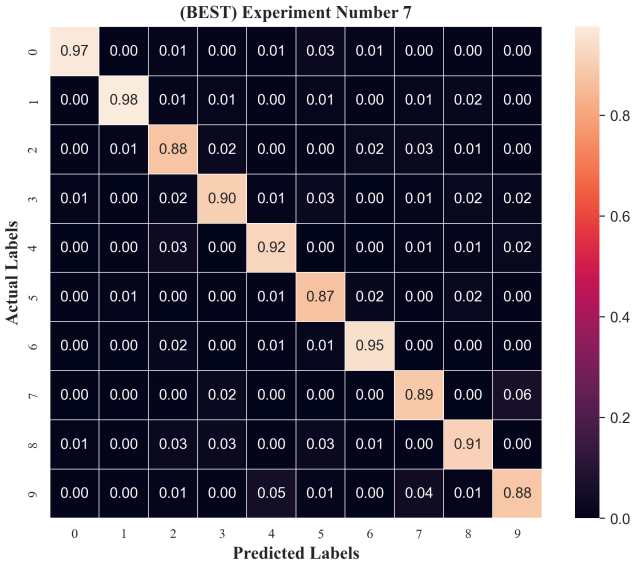

FIGURE 7. Confusion Matrix for Best Model

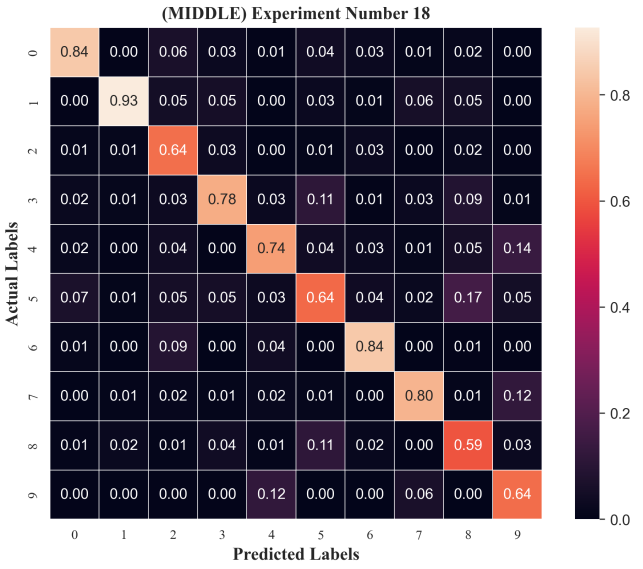

FIGURE 8. Confusion Matrix for Moderate Model

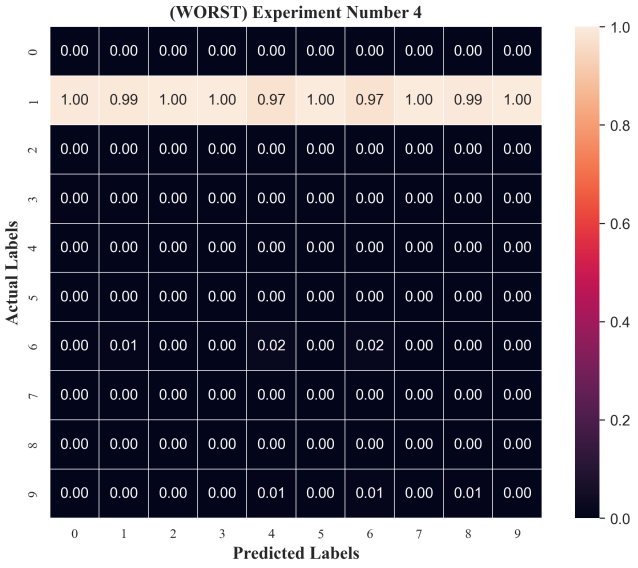

FIGURE 9. Confusion Matrix for Worst Model

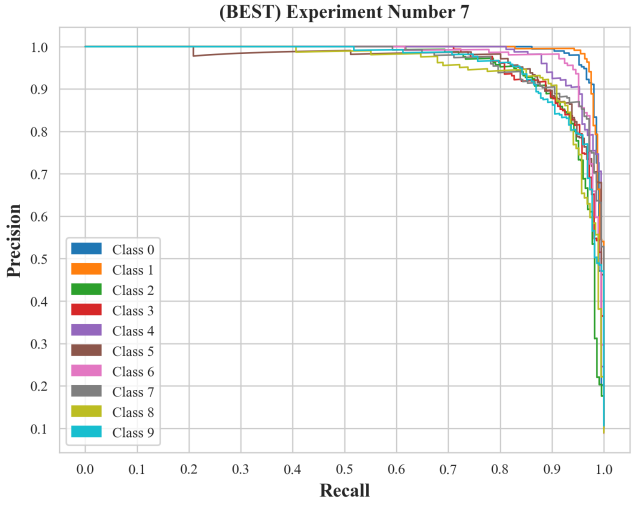

FIGURE 10. Precision-Recall Curve for Best Model

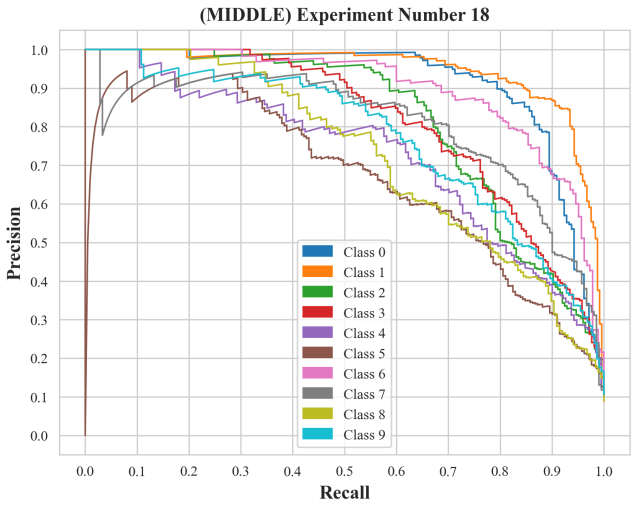

FIGURE 11. Precision-Recall Curve for Moderate Model

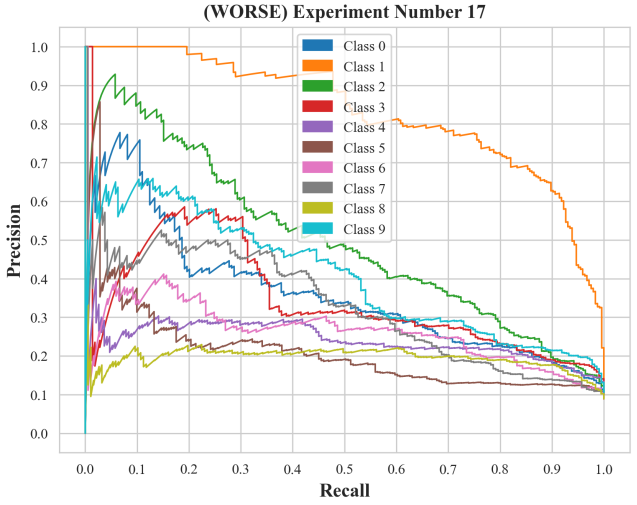

FIGURE 12. Precision-Recall Curve for Worse Model

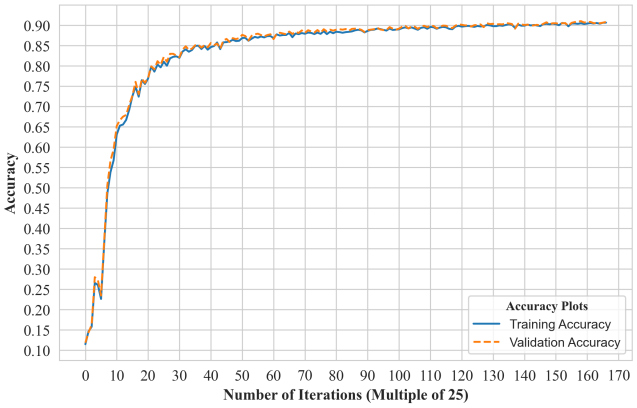

FIGURE 13. Training vs Validation Accuracy Plot from Pytorch Implementation

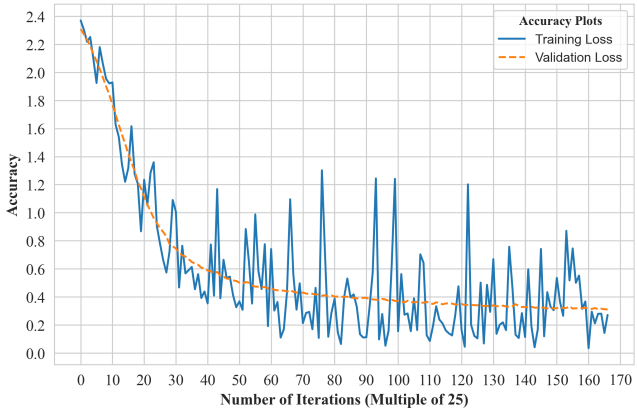

FIGURE 14. Training vs Validation Loss Curve from Pytorch Implementation

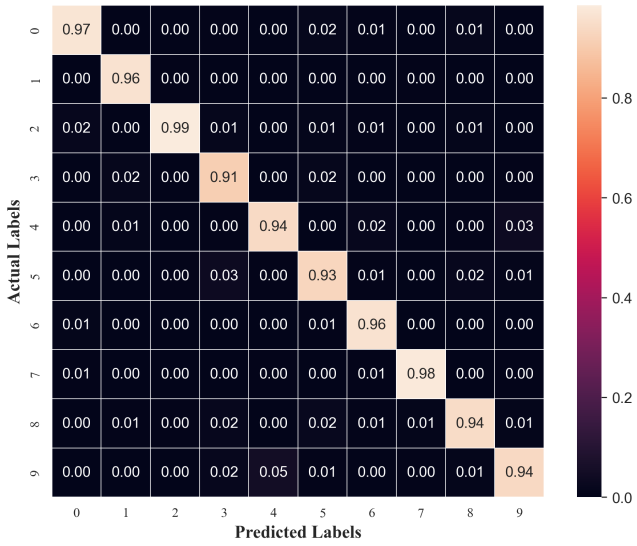

FIGURE 15. Confusion Matrix for Pytorch Implementation

APPENDIX A
CODE

1) Imported Modules and Functions

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
import pickle
from sklearn.metrics import
    precision_recall_curve,
    average_precision_score,
    accuracy_score
from sklearn.metrics import roc_curve,
    auc
from sklearn.metrics import
    classification_report,
    confusion_matrix
import seaborn as sns
from tqdm import tqdm
from matplotlib.font_manager import
    FontProperties
import matplotlib.patches as mpatches
import torchvision
import torchvision.transforms as
    transforms
from torch.utils.data import DataLoader,
     Dataset
import torch
import torch.nn as nn
import torch.optim as optim
```

2) Making Data ML Ready

```
dataset = pd.read_csv('./digit_data.csv'
    )
# converting dataframe 'dataset' into
    numpy array
dataset_np = dataset.values
no_data, no_features = dataset_np.shape

# shuffle the content of numpy array
np.random.shuffle(dataset_np)

test_set = dataset_np[:2100].T
y_test = test_set[0, :]
X_test = test_set[1:, :] / 255

valid_set = dataset_np[2100:4200].T
y_valid = valid_set[0, :]
X_valid = valid_set[1:, :] / 255

train_set = dataset_np[4200:].T
y_train = train_set[0, :]
X_train = train_set[1:, :] / 255

dataset = pd.read_csv('./digit_data.csv'
    )
dataset = dataset.sample(frac = 1,
    random_state = 69)

test_set = dataset.iloc[:2100]
valid_set = dataset.iloc[2100:4200]
train_set = dataset.iloc[4200:]

class CustomDataset(Dataset):
    def __init__(self, set_df):
        self.dataframe = set_df
        self.labels = self.dataframe.
            iloc[:, 0]
        self.data = self.dataframe.iloc
            [:, 1:]
        self.data = self.data / 255

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()
        labels_ = self.labels.iloc[idx]
        sample = {
            'data': torch.Tensor(self.
                data.iloc[idx].values),
            'label': torch.Tensor([int(
                labels_)])
        }
        sample['label'] = sample['label'
            ].to(torch.int64)
        return sample

batch_size = 8
train_dataset = CustomDataset(train_set)
valid_dataset = CustomDataset(valid_set)
test_dataset = CustomDataset(test_set)

train_dataloader = DataLoader(
    train_dataset, batch_size =
    batch_size, shuffle = True)
valid_dataloader = DataLoader(
    valid_dataset, batch_size =
    batch_size, shuffle = True)
test_dataloader = DataLoader(
    test_dataset, batch_size = batch_size
    , shuffle = True)
```

3) Lab Neural Architecture from Scratch

```
def parameter_initialize():
    W1 = np.random.uniform(-0.5, 0.5,
        (10, 784))
```

```
    W2 = np.random.uniform(-0.5, 0.5, (10, 10))
    b1 = np.random.uniform(-0.5, 0.5, (10, 1))
    b2 = np.random.uniform(-0.5, 0.5, (10, 1))
    return W1, b1, W2, b2

def forward_propagation(W1, b1, W2, b2, train_set_features):
    A0 = train_set_features
    Z1 = np.dot(W1, A0) + b1
    A1 = relu(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2

def one_hot_encode_array(digits):
    if not np.all((digits >= 0) & (digits <= 9)):
        raise ValueError("Digits must be in the range 0-9.")
    num_digits = len(digits)
    encoding = np.zeros((num_digits, 10))
    encoding[np.arange(num_digits), digits] = 1
    return encoding.T

def backward_propagation(Z1, A1, Z2, A2, train_set_features, train_set_labels):
    A0 = train_set_features
    n, m = A0.shape
    Y_OHE = one_hot_encode_array(train_set_labels)
    d_Z2 = A2 - Y_OHE
    d_W2 = 1 / m * np.dot(d_Z2, A1.T)
    d_b2 = 1 / m * np.sum(d_Z2, axis = 1)

    d_Z1 = np.dot(W2.T, d_Z2) * relu(Z1)
    d_W1 = 1 / m * np.dot(d_Z1, A0.T)
    d_b1 = 1 / m * np.sum(d_Z1, axis = 1)

    return d_W1, d_b1, d_W2, d_b2

def update_params(W1, b1, W2, b2, d_W1, d_b1, d_W2, d_b2, alpha):
    d_b1 = d_b1.reshape(10, 1)
    d_b2 = d_b2.reshape(10, 1)
    W1 = W1 - alpha * d_W1
    b1 = b1 - alpha * d_b1
    W2 = W2 - alpha * d_W2
    b2 = b2 - alpha * d_b2
    return W1, b1, W2, b2

def prediction(output_vec):
    return np.argmax(output_vec, axis = 0)

def accuracy(prediction, ground_truth):
    return np.sum(prediction == ground_truth) / len(ground_truth)

def gradient_descent(train_set_features, train_set_labels, alpha, no_iters):
    W1, b1, W2, b2 = parameter_initialize()
    A0 = train_set_features
    for i in range(no_iters):
        Z1, A1, Z2, A2 = forward_propagation(W1, b1, W2, b2, A0)
        d_W1, d_b1, d_W2, d_b2 = backward_propagation(Z1, A1, Z2, A2, A0, train_set_labels)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, d_W1, d_b1, d_W2, d_b2, alpha)
        if i \% 100 == 0:
            print(f"Iteration: {i}\
                tAccuracy: {accuracy(prediction(A2), train_set_labels):.2f} %"
                )
```

4) Utility Functions

```
def softmax(x):
    exp_x = np.exp(x - np.max(x))
    return exp_x / exp_x.sum(axis=0)

def relu(x):
    return np.maximum(0, x)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def tanh(x):
    return np.tanh(x)

def leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)

def one_hot_encode_array(digits):
    if not np.all((digits >= 0) & (digits <= 9)):
        raise ValueError("Digits must be in the range 0-9.")
    num_digits = len(digits)
```

```
    encoding = np.zeros((num_digits, 10))
    encoding[np.arange(num_digits), digits] = 1
    return encoding.T
```

5) Dynamic Neural Network

```
class NeuralNetwork:
    def __init__(self, layer_sizes, activation_functions = None, weight_init = "random", dropout_probs=None):
        self.layer_sizes = layer_sizes
        self.num_layers = len(layer_sizes)
        self.dropout_probs = dropout_probs
        self.parameters = self.initialize_parameters(weight_init)
        self.activation_functions = activation_functions

        self.hyperparameters = {
            "layer_sizes": layer_sizes,
            "activation_functions": activation_functions,
            "weight_init": weight_init,
            "dropout_probs": dropout_probs
        }

        saving_statement = f"\n\nlayer_size: {layer_sizes}\ntactivation_functions: {activation_functions}\ntweight_init: {weight_init}\ntdropout_probs: {dropout_probs}\n\n"


    def initialize_parameters(self, weight_init):
        parameters = {}
        for layer in range(1, self.num_layers):
            if weight_init == "random":
                parameters[f'W{layer}'] = np.random.uniform(-0.5, 0.5, (self.layer_sizes[layer], self.layer_sizes[layer-1]))
            elif weight_init == "kaiming":
                parameters[f'W{layer}'] = np.random.randn(self.layer_sizes[layer], self.layer_sizes[layer-1]) * np.sqrt(2.0 / self.layer_sizes[layer-1])
            elif weight_init == "xavier":
                parameters[f'W{layer}'] = np.random.randn(self.layer_sizes[layer], self.layer_sizes[layer-1]) * np.sqrt(1.0 / self.layer_sizes[layer-1])
            else:
                raise ValueError("Invalid weight initialization method.")

            parameters[f'b{layer}'] = np.random.uniform(-0.5, 0.5, (self.layer_sizes[layer], 1))
        return parameters

    def forward_propagation(self, X):
        A = X
        cache = {}
        for layer in range(1, self.num_layers):
            Z = np.dot(self.parameters[f'W{layer}'], A) + self.parameters[f'b{layer}']
            if self.dropout_probs and layer < self.num_layers - 1:
                # Apply dropout if specified (excluding the output layer)
                D = (np.random.rand(*Z.shape) < (1 - self.dropout_probs[layer-1])).astype(int)
                A = self.apply_activation(Z, self.activation_functions[layer - 1]) * D / (1 - self.dropout_probs[layer-1])
                cache[f'D{layer}'] = D
```

```
            else:
                A = self.apply_activation(Z, self.activation_functions[layer - 1])
            cache[f'Z{layer}'] = Z
            cache[f'Z{layer}'] = Z
            cache[f'A{layer}'] = A
        return cache

    def backward_propagation(self, cache, X, Y):
        m = X.shape[1]
        gradients = {}
        cache['A0'] = X
        for layer in range(self.num_layers - 1, 0, -1):
            if layer == self.num_layers - 1:
                dZ = cache['A' + str(layer)] - Y
            else:
                if self.dropout_probs:
                    dZ = np.dot(self.parameters[f'W{layer+1}'].T, gradients[f'dZ{layer+1}']) * (cache[f'D{layer}'] / (1 - self.dropout_probs[layer-1]))
                else:
                    dZ = np.dot(self.parameters[f'W{layer+1}'].T, gradients[f'dZ{layer+1}']) * relu(cache[f'Z{layer}'])
            dW = np.dot(dZ, cache['A' + str(layer-1)].T) / m
            db = np.sum(dZ, axis=1, keepdims=True) / m
            gradients[f'dZ{layer}'] = dZ
            gradients[f'dW{layer}'] = dW
            gradients[f'db{layer}'] = db
        return gradients

    def update_parameters(self, gradients, learning_rate):
        for layer in range(1, self.num_layers):
            self.parameters[f'W{layer}'] -= learning_rate * gradients[f'dW{layer}']
            self.parameters[f'b{layer}'] -= learning_rate * gradients[f'db{layer}']

    def train(self, X_train, y_train, X_valid, y_valid, learning_rate, num_iterations):
        y_train = one_hot_encode_array(y_train)
        if y_train.shape != (self.layer_sizes[-1], X_train.shape[1]):
            raise ValueError(f"Shape mismatch: Y should have shape {(self.layer_sizes[-1], X_train.shape[1])} when using one-hot encoding.")

        precision_list, recall_list, avg_precision_list = [], [], []
        for i in range(num_iterations):
            cache = self.forward_propagation(X_train)
            gradients = self.backward_propagation(cache, X_train, y_train)
            self.update_parameters(gradients, learning_rate)

            if i % 50 == 0:
                A2 = cache['A' + str(self.num_layers - 1)]

                train_loss = self.categorical_cross_entropy(A2, y_train)
                test_loss = self.categorical_cross_entropy(self.predict(X_valid), one_hot_encode_array(y_valid))

                train_accuracy = self.compute_accuracy(A2, y_train)
                test_accuracy = self.compute_accuracy(self.predict(X_valid), one_hot_encode_array(y_valid))
```

```
                print(f"Iteration: {i}\tTraining Loss: {train_loss:.4f}\tTraining Accuracy: {train_accuracy:.2f}%\tValidation Loss: {test_loss:.4f}\tValidation Accuracy: {test_accuracy:.2f}%")


    def categorical_cross_entropy(self, A, Y):
        m = Y.shape[1]
        loss = -np.sum(Y * np.log(A + 1e-15)) / m
        return loss

    def test(self, X_test, Y_test):
        cache = self.forward_propagation(X_test)
        A2 = cache['A' + str(self.num_layers - 1)]
        accuracy = self.compute_accuracy(A2, one_hot_encode_array(Y_test))
        print(f"Test Accuracy: {accuracy:.2f} %")
        return accuracy


    def predict(self, X):
        cache = self.forward_propagation(X)
        A2 = cache['A' + str(self.num_layers - 1)]
        return A2

    def compute_accuracy(self, A, Y):
        predictions = np.argmax(A, axis=0)
        ground_truth = np.argmax(Y, axis=0)
        return np.sum(predictions == ground_truth) / Y.shape[1]

    def apply_activation(self, Z, activation_function):
        if activation_function == "sigmoid":
            return sigmoid(Z)
        elif activation_function == "tanh":
            return tanh(Z)
        elif activation_function == "relu":
            return relu(Z)
        elif activation_function == 'softmax':
            return softmax(Z)
        else:
            raise ValueError("Invalid activation function.")

    def save_model(self, filename):
        model_data = {
            "hyperparameters": self.hyperparameters,
            "parameters": self.parameters
        }
        with open(filename, 'wb') as file:
            pickle.dump(model_data, file)

    def compute_precision_recall(self, X_test, Y_test):
        cache = self.forward_propagation(X_test)
        A2 = cache['A' + str(self.num_layers - 1)]

        # You need to calculate precision and recall for each class separately
        precision_list, recall_list, avg_precision_list = [], [], []

        for class_idx in range(self.layer_sizes[-1]):
            y_true = (Y_test == class_idx).astype(int)
            y_scores = A2.T[:, class_idx]
            precision, recall, _ = precision_recall_curve(y_true, y_scores)
            avg_precision = average_precision_score(y_true, y_scores)

            precision_list.append(precision)
            recall_list.append(recall)
            avg_precision_list.append(avg_precision)
```

```
        return precision_list, recall_list, avg_precision_list

    @classmethod
    def load_model(cls, filename):
        with open(filename, 'rb') as file:
            model_data = pickle.load(file)
        hyperparameters = model_data["hyperparameters"]
        parameters = model_data["parameters"]
        model = cls(**hyperparameters) # Reconstruct the model with the same hyperparameters
        model.parameters = parameters # Set the loaded parameters
        return model
```

6) Evaluation and Visualizations

```
outputs = {
    "BEST_data": [],
    "MIDDLE_data": [],
    "WORST_data": []
}
placements = ['BEST', 'MIDDLE', 'WORST']
p_index = 0
for line in lines:
    if placements[p_index] in line:
        record_data = True
        continue
    elif '****' in line:
        record_data = False
        p_index += 1
        continue

    if record_data:
        t_ls, t_ac, v_ls, v_ac = line.split('\t')
        t_ls = float(t_ls.split(' ')[1])
        t_ac = float(t_ac)
        v_ls = float(v_ls)
        v_ac = float(v_ac.strip('\n'))
        outputs[f"{placements[p_index]}_data"].append([t_ls, t_ac, v_ls, v_ac])

for key in outputs.keys():
    outputs[key] = np.array(outputs[key])

    sole_data = outputs[key]
    training_loss = sole_data[:, 0]
    training_accuracy = sole_data[:, 1]
    valid_loss = sole_data[:, 2]
    valid_accuracy = sole_data[:, 3]
    temp_dict = {
        't_loss': training_loss,
        't_accuracy': training_accuracy,
        'v_loss': valid_loss,
        'v_accuracy': valid_accuracy
    }
    outputs[key] = temp_dict

# Training Loss vs all the performances
plt.figure(figsize = (8, 5))
sns.set_style('whitegrid')
plt.plot(outputs['BEST_data']['t_loss'], label = 'Training: Best Exp.', color = 'b')
plt.plot(outputs['BEST_data']['v_loss'], label = 'Validation: Best Exp.', color = 'b', linestyle = '--')

plt.plot(outputs['MIDDLE_data']['t_loss'], label = 'Training: Middle Exp.', color = 'g')
plt.plot(outputs['MIDDLE_data']['v_loss'], label = 'Validation: Middle Exp.', color = 'g', linestyle = '--')

plt.plot(outputs['WORST_data']['t_loss'], label = 'Training: Worst Exp.', color = 'r')
plt.plot(outputs['WORST_data']['v_loss'], label = 'validation: Worst Exp.', color = 'r', linestyle = '--')

plt.xlabel("Number of Iterations (Multiple of 50)", font = font_label)
plt.ylabel("Loss", font = font_label)

plt.locator_params(axis='x', nbins = 20)
plt.locator_params(axis='y', nbins = 20)

plt.xticks(font = font_xticks)
plt.yticks(font = font_yticks)
legend = plt.legend(loc = 'best', prop = font, bbox_to_anchor = (0.98, 0.78))
legend.set_title("Loss Curves", prop = {'size': 10, 'weight': 'bold', 'family': 'Times New Roman'})
plt.show()

plt.figure(figsize = (8, 5))
sns.set_style('whitegrid')
plt.plot(outputs['BEST_data']['t_accuracy'], label = 'Training: Best Exp.', color = 'b')
```

```
plt.plot(outputs['BEST_data']['v_accuracy'], label = 'Validation: Best Exp.', color = 'b', linestyle = '--')

plt.plot(outputs['MIDDLE_data']['t_accuracy'], label = 'Training: Middle Exp.', color = 'g')
plt.plot(outputs['MIDDLE_data']['v_accuracy'], label = 'Validation: Middle Exp.', color = 'g', linestyle = '--')

plt.plot(outputs['WORST_data']['t_accuracy'], label = 'Training: Worst Exp.', color = 'r')
plt.plot(outputs['WORST_data']['v_accuracy'], label = 'validation: Worst Exp.', color = 'r', linestyle = '--')

plt.xlabel("Number of Iterations ( Multiple of 50)", font = font_label)
plt.ylabel("Accuracy", font = font_label)

plt.locator_params(axis='x', nbins = 20)
plt.locator_params(axis='y', nbins = 20)

plt.xticks(font = font_xticks)
plt.yticks(font = font_yticks)
legend = plt.legend(loc = 'best', prop = font, bbox_to_anchor = (0.98, 0.62))
legend.set_title("Accuracy Plots", prop = {'size': 10, 'weight': 'bold', 'family': 'Times New Roman'})
plt.show()

loaded_model = NeuralNetwork.load_model("./Models/saved_model_18.pkl")
y_pred = np.argmax(loaded_model.predict(X_test), axis = 0)

print(classification_report(y_pred, y_test))

loaded_model = NeuralNetwork.load_model("./Models/saved_model_7.pkl")
y_pred = np.argmax(loaded_model.predict(X_test), axis = 0)

plt.figure(figsize = (8, 6))
sns.heatmap(confusion_matrix(y_pred, y_test, normalize = 'pred'), annot = True, fmt = "1.2f", linewidth = 0.01, square = True)
plt.xlabel("Predicted Labels", font = font_label)
plt.ylabel("Actual Labels", font = font_label)
plt.xticks(font = font)
plt.yticks(font = font)
plt.title("(BEST) Experiment Number 7", font = font_label)
plt.show()

loaded_model = NeuralNetwork.load_model("./Models/saved_model_18.pkl")
y_pred = np.argmax(loaded_model.predict(X_test), axis = 0)

plt.figure(figsize = (8, 6))
sns.heatmap(confusion_matrix(y_pred, y_test, normalize = 'pred'), annot = True, fmt = "1.2f", linewidth = 0.01, square = True)
plt.xlabel("Predicted Labels", font = font_label)
plt.ylabel("Actual Labels", font = font_label)
plt.xticks(font = font)
plt.yticks(font = font)
plt.title("(MIDDLE) Experiment Number 18", font = font_label)
plt.show()

loaded_model = NeuralNetwork.load_model("./Models/saved_model_4.pkl")
y_pred = np.argmax(loaded_model.predict(X_test), axis = 0)

plt.figure(figsize = (8, 6))
sns.heatmap(confusion_matrix(y_pred, y_test, normalize = 'pred'), annot = True, fmt = "1.2f", linewidth = 0.01, square = True)
plt.xlabel("Predicted Labels", font = font_label)
plt.ylabel("Actual Labels", font = font_label)
plt.xticks(font = font)
plt.yticks(font = font)
plt.title("(WORST) Experiment Number 4", font = font_label)
plt.savefig("./Figures/5_worst_CM.png", dpi = 300, bbox_inches = 'tight')
plt.show()

loaded_model = NeuralNetwork.load_model("./Models/saved_model_7.pkl")
precision, recall, avg_precision = loaded_model.compute_precision_recall
```

```
    (X_test, y_test)

colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf']
for class_idx in range(10):
    plt.plot(recall[class_idx], precision[class_idx], color=colors[class_idx], lw=1, label=f'Class {class_idx}')

plt.xlabel('Recall', font = font_label)
plt.ylabel('Precision', font = font_label)
plt.title('(BEST) Experiment Number 7', font = font_label)

legend_patches = [mpatches.Patch(color=colors[class_idx], label=f'Class {class_idx}') for class_idx in range(10)]
plt.locator_params(axis='x', nbins = 20)
plt.locator_params(axis='y', nbins = 20)
plt.legend(handles=legend_patches, loc='best', prop = font)
plt.xticks(font = font)
plt.yticks(font = font)
plt.grid(True)
plt.show()

loaded_model = NeuralNetwork.load_model("./Models/saved_model_18.pkl")
precision, recall, avg_precision = loaded_model.compute_precision_recall(X_test, y_test)

colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf']
for class_idx in range(10):
    plt.plot(recall[class_idx], precision[class_idx], color=colors[class_idx], lw=1, label=f'Class {class_idx}')

plt.xlabel('Recall', font = font_label)
plt.ylabel('Precision', font = font_label)
plt.title('(MIDDLE) Experiment Number 18', font = font_label)

legend_patches = [mpatches.Patch(color=colors[class_idx], label=f'Class {class_idx}') for class_idx in range(10)]
plt.locator_params(axis='x', nbins = 20)
plt.locator_params(axis='y', nbins = 20)
plt.legend(handles=legend_patches, loc='best', prop = font)
plt.xticks(font = font)
plt.yticks(font = font)
plt.grid(True)
plt.savefig("./Figures/7_middle_PR-curve.png", dpi = 300, bbox_inches = 'tight')
plt.show()

loaded_model = NeuralNetwork.load_model("./Models/saved_model_17.pkl")
precision, recall, avg_precision = loaded_model.compute_precision_recall(X_test, y_test)

colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf']
for class_idx in range(10):
    plt.plot(recall[class_idx], precision[class_idx], color=colors[class_idx], lw=1, label=f'Class {class_idx}')

plt.xlabel('Recall', font = font_label)
plt.ylabel('Precision', font = font_label)
plt.title('(WORSE) Experiment Number 17', font = font_label)

legend_patches = [mpatches.Patch(color=colors[class_idx], label=f'Class {class_idx}') for class_idx in range(10)]
plt.locator_params(axis='x', nbins = 20)
plt.locator_params(axis='y', nbins = 20)
plt.legend(handles=legend_patches, loc='best', prop = font)
plt.xticks(font = font)
plt.yticks(font = font)
plt.grid(True)
plt.savefig("./Figures/8_worse_PR-curve.png", dpi = 300, bbox_inches = 'tight')
plt.show()
```

7) PyTorch Implementation

```
class SimpleNN(nn.Module):
    def __init__(self, input_size, output_size, hidden_layers=[]):
```

```
        super(SimpleNN, self).__init__()

        layers = []
        layers.append(nn.Linear(input_size, hidden_layers[0] if hidden_layers else output_size))
        layers.append(nn.ReLU())

        for i in range(len(hidden_layers) - 1):
            layers.append(nn.Linear(hidden_layers[i], hidden_layers[i+1]))
            layers.append(nn.ReLU())

        layers.append(nn.Linear(hidden_layers[-1] if hidden_layers else input_size, output_size))
        layers.append(nn.LogSoftmax(dim=1))

        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)

input_size = 784
output_size = 10

hidden_layers = [20, 10]

model = SimpleNN(input_size, output_size, hidden_layers)

criterion = nn.NLLLoss()  # Negative Log Likelihood Loss for classification
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training loop (you'll need to prepare your dataset and dataloaders)

for batch_inputs, batch_targets in tqdm(train_loader):
    optimizer.zero_grad()
    batch_inputs = batch_inputs.view(batch_inputs.size(0), -1)
    outputs = model(batch_inputs)
    loss = criterion(outputs, batch_targets)
    loss.backward()  # Backpropagation
    optimizer.step()  # Update weights

model.eval()

predictions = []
true_labels = []

with torch.no_grad():
    for batch_inputs, batch_targets in test_loader:
        batch_inputs = batch_inputs.view(batch_inputs.size(0), -1)

        outputs = model(batch_inputs)
        _, predicted = torch.max(outputs, 1)
        predictions.extend(predicted.tolist())
        true_labels.extend(batch_targets.tolist())

predictions = np.array(predictions)
true_labels = np.array(true_labels)
accuracy_score(true_labels, predictions)

plt.figure(figsize = (8, 5))
sns.set_style('whitegrid')
plt.plot(accuracy_list[:, 0], label = 'Training Accuracy')
plt.plot(accuracy_list[:, 1], label = 'Validation Accuracy', linestyle = '--')

plt.xlabel("Number of Iterations (Multiple of 25)", font = font_label)
plt.ylabel("Accuracy", font = font_label)

plt.locator_params(axis='x', nbins = 20)
plt.locator_params(axis='y', nbins = 20)

plt.xticks(font = font_xticks)
plt.yticks(font = font_yticks)
legend = plt.legend(loc = 'best')
legend.set_title("Accuracy Plots", prop = {'size': 10, 'weight': 'bold', 'family': 'Times New Roman'})
plt.savefig("./Figures/9_training_accuracies.png", dpi = 300, bbox_inches = 'tight')
plt.show()

accuracy_list = np.array(output_dict['loss_list'])
font = FontProperties(family='Times New Roman', size = 9)
font_label = FontProperties(family='Times New Roman', size = 12, weight =
```

```
    "bold")
font_xticks = FontProperties(family='Times New Roman', size = 12)
font_yticks = FontProperties(family='Times New Roman', size = 12)

plt.figure(figsize = (8, 5))
sns.set_style('whitegrid')
plt.plot(accuracy_list[:, 0], label = 'Training Loss')
plt.plot(accuracy_list[:, 1], label = 'Validation Loss', linestyle = '--')

plt.xlabel("Number of Iterations (Multiple of 25)", font = font_label)
plt.ylabel("Accuracy", font = font_label)

plt.locator_params(axis='x', nbins = 20)
plt.locator_params(axis='y', nbins = 20)

plt.xticks(font = font_xticks)
plt.yticks(font = font_yticks)
legend = plt.legend(loc = 'best')
legend.set_title("Accuracy Plots", prop = {'size': 10, 'weight': 'bold', 'family': 'Times New Roman'})
plt.savefig("./Figures/10_train_vs_validation_losses.png", dpi = 300, bbox_inches = 'tight')
plt.show()

plt.figure(figsize = (8, 6))
sns.heatmap(confusion_matrix(true_labels, predictions, normalize = 'pred'), annot = True, fmt = "1.2f", linewidth = 0.01, square = True)
plt.xlabel("Predicted Labels", font = font_label)
plt.ylabel("Actual Labels", font = font_label)
plt.xticks(font = font)
plt.yticks(font = font)
plt.show()
```