Comp 5329 Assignment 1

Abstract

In this assignment, we manually develop a Multilayer Perceptron (MLP) classifier by using Numpy library in python. The dataset consists of 128 features and 10 label classes (0 to 9), and it is divided into training and testing set. Following preprocessing of dataset, we will write functions with object class of key components, such as initialization, forward pass, backpropagation, SoftMax activation, cross entropy loss and optimization algorithms without using any machine learning modules. We will also incorporate regularization methods like weight decay or batch normalization to ensure the model is more stable and generalizable. In addition, we will report on our experiments to refine our model followed by evaluation metrics analysis.

1 Introduction

In the fast evolving field of deep learning, multilayer perceptron (MLP) network, developed by Rosenblatt in 1958, is fundamental yet still widely used architecture of neural networks. This model consists of input layer, output layer and many hidden layers that enables it perform complex classification and predictive tasks. Each hidden layer has multiple neurons, each neuron has a activation function that allows it to learn non-linear patterns. From input to output layer, every neuron is connected to ones in next layer by weights and bias parameters to produces continued learning of the input data. MLP is important network model because it paves the way for many more advanced architectures, such as CNN models and offers theoretical and practical insights for deep learning.([1])

The purpose of this assignment is to build a MLP classifier from stretch using only numpy module. In the method section, we will explain every step of building our MLP model, the loss function, optimizer we chose and regularization techniques. Following this part, we show our experiments steps, where we validate model at epoch and refine models based on several evaluation metrics.

2 Methods

2.1 Dataset

We are provided with datasets that are split into training and testing sets. In the training dataset, there are 128 features and 50,000 observations. They are stored in (50000,28) numpy array. The testing has 10000 instances with the same dimension. For labels, each data point belongs to one of ten target classes (0 to 9).

2.2 Preprocessing

Data processing is a important step to make sure the input data is modified to better fit into the classifier with stable performance. For MLP model, we need to normalize the input data vectors.

2.2.1 Normalization

It is imperative to scale the features when fitting MLP classifier, and normalization is one of the popular methods. By normalizing the attributes before modeling, it makes data entries between 0 to 1, which ensures more stable and faster convergence. It also makes sure each

feature is represented in equal weights to avoid features with large scales dominate the learning process.[2] The normalization is calculated as:

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \tag{1}$$

2.3 Xavier initialization

We start building the MLP classifier by initializing parameters, including weights and biases, so that we can start further iterations. Some common practices are to assign 0 or small random numbers to weights, but it may result in problems such as symmetry neurons issues and vanishing gradients. Xavier initialization is good way to avoid these issues. It ensures the mean of activation is 0 and variance across all the layers is the same. The weights can be generated by random number following normal distribution with mean and variance as: weights $\sim \mathcal{N}\left(0, \frac{2}{n[l-1]+n[l]}\right)$, where n is the number neurons in the current and previous layers.[3]

2.4 Cross Entropy Loss function

The loss function in machine learning measures the error between the model predicted outcomes and actual label values. In MLP, loss function is important for us to perform updates to refine parameters. Because we are dealing with a multiple classification task, the output is probability distribution of each target class, so cross entropy is a suitable option. Its loss value increases when the predicted probability is far from actual target. Cross entropy is calculated as: [4]

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \tag{2}$$

where:

- $\mathbf{y}$ is the one-hot encoded vector of the true labels,
- $\hat{\mathbf{y}}$ is the vector of predicted probabilities,
- C is the number of classes.

2.5 Activation Function

Activation function is a computational operation applied to input of every neuron in the network produce output for further processing. The purpose of activation is to introduce non-linearity into the output of a neuron, which is vital for learning complex data patterns that linear equations could not capture. In our model, we apply Relu and softmax functions.

2.5.1 Relu Activation

Relu, stands for Rectified Linear Unit, is a popular activation function in neural network classification. It one of the functions we attempted along with sigmoid and tanh. The advantages of Relu over the other two are that Relu is computionally less expensive, and it can also overcome the vanishing gradient issues. The derivatives of sigmoid and tanh functions become very small when their inputs are large or small. However, the derivative of relu is 1 for all positive inputs, which helps keep a strong gradient. [3] Relu is defined as

$$\text{ReLU}(x) = \max(0, x) \tag{3}$$

The derivative of Relu is

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{4}$$

2.5.2 Gelu Activation

The GELU, stands for Gaussian Error Linear Unit, is another powerful is a nonlinear activation function used in advanced neural networks, particularly effective in network like Transformer. Compared to Relu, it provides a more smoother output by introducing stochastic regularization effects, which help avoids issues related to gradient discontinuity. GELU also acts like a gate, deciding the proportion of signals that should pass through depending on the input itself, thus dynamically adjusting according to the input data. An approximation of the GELU function is: [5]

$$\text{GELU}(x) \approx 0.5x \left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right)\right)$$

2.5.3 Softmax Activation

The softmax activation function is a crucial element used in the output layer of neural networks, designed for multiclass classification task. Similar to logistic function used in binary classification, softmax converts the raw predicted values for each class into probability. It takes the exponentials of each output and then normalize them by dividing by the sum of all exponentials. As result, the output values are ranged between 0 to 1 and sum up to 1. Softmax is defined as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \tag{5}$$

where z_i represents the logit or raw score for class i, and n is the total number of classes.

2.6 mini batch training

After initializing weights and defining the loss function, we train the model with mini batch approach. It splits the train data into many mini batches and updates each batch sequentially. Within each mini batch, we run forward and back propagation to calculate gradients of all parameters with respect to the loss, and update them with all the instances. One epoch is complete after all the mini batches are iterated. This approach is more effecient than batch gradient descent, which uses the entire dataset to compute the gradient. But it also converges faster than stochastic gradient descent, where updates are made for each data instance.

2.7 Optimizer

2.7.1 Momentum in SGD

Momentum is a technique used to boost the performance of SGD in neural network by speeding up its convergence. Standard stochastic gradient descent update parameters by subtracting the gradients of loss function multiplied by a learning rate. Momentum, however, extend this approach by adding a portion of previously updated parameter vectors to the current updates. This way can accelerates algorithm to reach minimum by accumulating the past updates that consistently occur in the same direction. The SGD with Momentum is calculated as:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \tag{6}$$

$$\theta = \theta - v_t \tag{7}$$

where:

- v_t is the velocity at time step t,
- γ is the momentum coefficient, typically between 0.9 and 0.99,
- η is the learning rate,
- $\nabla_\theta J(\theta)$ is the gradient of the loss function J with respect to the parameters θ,
- θ represents the parameters of the model.

2.7.2 Adam

Adam, stands for Adaptive Moment Estimation, is a more advanced optimizer in neural network training. It combines features from both momentum and adaptive gradient methods. Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. This way incorporates the moving average of the first moment and the variance of the second moment of the gradients. This enables benefits of momentum techniques while also scaling the learning rate based on the recent magnitudes of gradients for each parameter. The optimization process will be efficient as result. The Adam update process is computed as: [6]

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta J(\theta), \tag{8}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta J(\theta))^2, \tag{9}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \tag{10}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}, \tag{11}$$

$$\theta = \theta - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}. \tag{12}$$

where:

- θ are the parameters of the model,
- $\nabla_\theta J(\theta)$ is the gradient of the loss function with respect to θ,
- m_t and v_t are estimates of the first and second moments of the gradients,
- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected versions of m_t and v_t,
- β_1 and β_2 are the exponential decay rates for the moment estimates,
- η is the learning rate,
- ϵ is a small constant added to improve numerical stability.

2.8 Regularization

2.8.1 Dropout

Dropout is a regularization technique widely used in neural network training. This method randomly drops out a proportion p of neurons in every hidden layers during the training phase. p is called dropout rate. Parameters connected to those neurons are not dropped in forward and back propagation at each iterations. However, during the validation phase, we use the full layers without dropout and scaled down the output to account for larger network compared to training phase. This is done by multiplying the outputs of neurons by the probability of retention rate (e.g. $1 - p$) which compensates the expected higher input neurons in the testing stage. Dropout can effectively prevent overfitting and improve generalization by not fine-tuning the training data.

2.8.2 Weight Decay

Weight decay is another popular regularization method in training neural network. The idea is to add a penalty to the loss function that is proportional to the size of the weights, which pushes the model to optimize smaller weights and less likely to overfit as a result. The added regularization term will increase the loss if it weights are overfitted. The weight decay with L2 is expressed as: The loss function with L2 regularization (weight decay) is defined as follows:

$$L = L_0 + \frac{\lambda}{2}\|\mathbf{w}\|^2 \tag{13}$$

where:

- L_0 represents the original loss function, such as cross-entropy or mean squared error, depending on the task.
- $\mathbf{w}$ denotes the vector of weights in the neural network.
- λ is the regularization parameter, controlling the strength of the weight decay.
- $\|\mathbf{w}\|^2$ is the L2 norm of the weights, calculated as the sum of the squares of all the weight coefficients.

2.8.3 Batch Normalization

Batch normalization is another method to improve neural network training. It is introduced to address the issue of covariate shift, where the distribution of each layer inputs change during training, as the parameters of the previous layers change. It works by normalizing the input before activation in each layer, which enables higher learning rates and reduces the dependency on initialization. Mathematically, for a given feature x in a layer, the normalized value $\hat{x}$ is calculated as

$$\hat{z}_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{14}$$

where μ_B and σ_B^2 are the mean and variance of the i-th component of z calculated across the mini-batch. ϵ is a small number added for numerical stability. After normalization, each $\hat{z}_i$ is linearly transformed by scaling and shifting using parameters β and γ:

$$y_i = \gamma\hat{z}_i + \beta \tag{15}$$

These parameters are specific to each neuron and allow the network to potentially undo the normalization if that results in a lower loss.

3 Experiment and Results

Our goal in this study is to assess how different machine learning approaches affect a Multi-Layer Perceptron model's performance. In particular, we investigate a four-layer neural network architecture that includes batch normalization, GELU and ReLU activation functions, and two optimization techniques: Momentum-based Stochastic Gradient Descent (SGD) and Adam. We aim to comprehend their impact on the model's training effectiveness, convergence behavior, and predictive accuracy on a particular dataset by methodically experimenting with these elements. These experiments' outcomes shed light on creating reliable neural network models that are suited to particular tasks, laying the groundwork for future development and real-world implementation.

3.1 Parameter

The number of neurons, activation function, dropout probability, learning rate optimization, weight decay, and batch normalization are the parameters that we intend to examine. The following are the default values for the parameters: [128, 100, 32, 10] Adam is the optimizer, batch normalization is enabled with the batch size as 100, dropout probability is set to 0.3, activation functions include [None, ReLU, GELU, Softmax], and weight decay is set to 0.5.

3.2 Activation Function and Hidden Layer

We examined the effects of various functions for introducing nonlinearity across a range of hidden layer counts to identify the ideal activation function and network depth for our neural network model. Using 3, 4, and 5 hidden layers, we evaluated ReLU, GELU, and a mixed configuration (alternating ReLU and GELU) in a multilayer perceptron (MLP) with the neuron configuration [128, 100, 32, 10]. Throughout every experiment, the output layer employed softmax, and the input layer lacked activation.

Table 1: Comparison of Total Loss and Testing Accuracy Across Different Activation Functions

Hidden Layer	**ReLU**		**GELU**		**Mix**	
	Total Loss	**Testing Accuracy**	**Total Loss**	**Testing Accuracy**	**Total Loss**	**Testing Accuracy**
5	1.8011	34.64%	1.8898	30.72%	1.7944	35.2%
4	1.8521	32.46%	1.8026	35.91%	1.7257	38.82%
3	1.8885	32.05%	1.8902	32.32%	-	-

For the five-hidden-layer model, ReLU produced a testing accuracy of 34.64% with a training loss of 1.8011. The mixed configuration came in with the best accuracy of 35.2% and a slightly lower loss of 1.7944. With an accuracy of 30.72% and a loss of 1.8898, GELU fared the worst, indicating slower convergence and less generalization as the network got deeper. Potential overfitting or sensitivity to the alternating pattern in deeper architectures is suggested by the mixed configuration's lower loss but marginally lower accuracy when compared to ReLU.

For the four-hidden-layer model, GELU had a marginally higher accuracy of 35.91% but a lower loss of 1.8206, indicating better training stability but inconsistent generalization, compared to ReLU's testing accuracy of 32.46% and training loss of 1.8521. Notably, of all the configurations tested, the mixed configuration performed better than both, obtaining the lowest training loss of 1.7257 and the highest testing accuracy of 38.82%. This suggests that convergence and prediction performance at this depth were greatly enhanced by the combination of GELU and ReLU in alternating layers.

For the three-hidden-layer model, with a training loss of 1.8885 and a testing accuracy of 32.05%, ReLU outperformed GELU, which had a marginally higher accuracy of 32.32% but

a similar loss of 1.8902. Because the input and output layers' activation functions were fixed, the mixed configuration was not tested at this depth. High training losses suggested possible underfitting or sluggish convergence at this depth, while both ReLU and GELU demonstrated mediocre performance.

ReLU performed consistently across the tested configurations, maintaining competitive results at shallower depths and attaining better accuracy at 5 hidden layers than GELU. This is probably because ReLU can mitigate vanishing gradients with a derivative of 1 for positive inputs. With its lowest accuracy of 30.72% at 5 hidden layers and higher training losses (e.g., 1.8902 at 3 hidden layers), GELU performed poorly in deeper networks, suggesting slower learning that may have been made worse by the lack of batch normalization. With the lowest training loss of 1.7257 and the best overall testing accuracy of 38.82%, the mixed configuration demonstrated the greatest promise at 4 hidden layers. However, at 5 hidden layers, its performance deteriorated, indicating sensitivity to network depth. In light of these findings, we chose a four-layer architecture with two hidden layers, utilizing the neuron configuration and the mixed configuration (GELU for the first hidden layer and ReLU for the second) [128, 64, 32, 10]. This decision strikes a balance between high accuracy and effective training by utilizing GELU's capacity to recognize intricate patterns in the first layer and ReLU's quicker convergence in the second layer. Future research could look into changing the learning rate optimizer or reintroducing batch normalization to further increase stability, particularly in deeper networks.

3.3 Normalization and Standardization

Table 2: Comparison of Total Loss and Testing Accuracy Between Normalization and Standardization

Method	Total Loss	Testing Accuracy (%)
Normalization	1.8639	22.44
Standardization	1.8885	32.05

Using a four-layer MLP with two hidden layers and ReLU activation across all hidden layers, with the neuron configuration, we assessed the effects of normalization and standardization on the input dataset to determine the best data preprocessing technique for our neural network model [128, 64, 32, 10]. A total training loss of 1.8639 and a testing accuracy of 22.44% were the outcomes of normalization, which scales input features to a range of [0, 1]. Standardization, on the other hand, which changes features to have a mean of 0 and a standard deviation of 1, produced a much higher testing accuracy of 32.05% but a lower training loss of 1.8805. These findings show that standardization significantly enhanced the model's generalization to unseen data, as demonstrated by the increase in testing accuracy, even though normalization produced a marginally better training loss. Since we found a higher accuracy with standardization implies that it better maintains the relative relationships between features, allowing for more efficient learning in our MLP. As standardization supports our objective of optimizing predictive performance on the test set and guarantees that the model can generalize effectively across a variety of data distributions, we will use it as our preprocessing technique for the finished model.

3.4 Number of Neurons

Table 3: Comparison of Total Loss and Testing Accuracy Across Different Network Architectures

Architecture (Hidden Layers)	Total Loss	Testing Accuracy (%)
128, 32, 16, 10	0.9841	32.81
128, 64, 32, 10	0.8639	39.38
128, 128, 64, 10	0.7717	43.47

We tested three different hidden layer size configurations in a four-layer MLP with two hidden layers using the mixed configuration, standardization for preprocessing, and the architecture [input, hidden1, hidden2, output] to determine how the number of neurons affected the performance of our neural network model. With 32 neurons in the first hidden layer and 16 in the second, the first configuration [128, 32, 16, 10] produced a testing accuracy of 32.81% and a total training loss of 0.9841. With 64 neurons in the first hidden layer and 32 in the second, the second configuration [128, 64, 32, 10] performed better, obtaining a higher testing accuracy of 39.38% and a lower training loss of 0.8639. With 128 neurons in the first hidden layer and 64 in the second, the third configuration [128, 128, 64, 10] produced the best results, with the highest testing accuracy of 43.47% and the lowest training loss of 0.7717. According to these results, the model's ability to learn intricate patterns is improved by adding more neurons to the hidden layers, which results in better generalization (higher accuracy) and convergence (lower loss). As previously mentioned, dropout and weight decay can be used to reduce the risk of overfitting, which the more extensive configuration may also increase. These findings led us to choose the [128, 128, 64, 10] configuration for our final model since it offers the best trade-off between high testing accuracy and low training loss, guaranteeing reliable performance on our dataset while preserving computational efficiency.

3.5 Weight Decay

Table 4: Comparison of Model Performance with Different Weight Decay Values

Weight Decay	Total Loss	Testing Accuracy (%)
0.98	1.5095	44.42
0.5	0.7717	43.47
0.1	0.1573	43.83

We tested three different values—0.98, 0.5, and 0.1—in a four-layer MLP with two hidden layers using the mixed configuration, the neuron configuration [128, 128, 64, 10], and standardization for preprocessing to assess the impact of weight decay as a regularization technique in our neural network model. The model obtained a testing accuracy of 44.42% and a total training loss of 1.5095 with a weight decay of 0.98. Lowering the weight decay to 0.5 led to a slightly lower testing accuracy of 43.47% and a lower training loss of 0.7717, suggesting better convergence but a slight decline in generalization. The lowest training loss of 0.1573 and the lowest testing accuracy of 43.83% were obtained by further reducing the weight decay to 0.1, which may indicate overfitting because the model fits the training data too closely. These findings demonstrate the trade-off between generalization and training loss: lower weight decay

(0.1) reduces training loss at the expense of overfitting, while higher weight decay (0.98) offers stronger regularization and the highest testing accuracy. These results led us to choose a weight decay of 0.98 for our final model, which prioritizes generalization over minimizing training loss and guarantees the model's robust performance on unseen data. It also achieves the highest testing accuracy of 44.42%.

3.6 Dropout

3.7 Weight Decay

Table 5: Comparison of Model Performance with Different Weight Decay Values

Dropout	Total Loss	Testing Accuracy (%)
0.3	1.5807	44.78
0.5	1.6862	39.19
0.7	1.8494	32.84

We experimented with three different dropout rates—0.3, 0.5, and 0.7, to find the best one for regularizing our neural network model. The model demonstrated good regularization with strong generalization, achieving a total training loss of 1.58068 and a testing accuracy of 44.78% with a dropout rate of 0.3. The model may have underfitted by dropping too many units during training, as evidenced by the slightly higher training loss of 1.6862 and lower testing accuracy of 39.19% that occurred when the dropout rate was increased to 0.5. Increasing the dropout rate further to 0.7 resulted in the lowest testing accuracy of 32.84% and a training loss of 1.8494, confirming that excessive dropout significantly hindered the model's learning capacity, most likely because there were not enough active neurons during training. These findings show that, in our setup, a lower dropout rate better balances regularization and learning capacity, which we eliminated earlier. These results led us to choose a dropout rate of 0.3 for our final model, which guarantees strong generalization while preserving efficient training dynamics and attains the highest testing accuracy of 44.78%.

3.8 Learning Rate Optimization

Table 6: Comparison of Model Performance Between Adam and Momentum SGD Optimizers

Optimizer (LR = 0.001)	Total Loss	Testing Accuracy (%)
Adam	1.5807	44.78
Momentum SGD	1.5196	43.76

We compared two optimizers, Adam and Momentum SGD, both using a learning rate of 0.001 in order to assess how optimizer selection affected the performance of our neural network. The model showed strong generalization and stable convergence with a testing accuracy of 44.78% and a total training loss of 1.58068 using the Adam optimizer. In contrast, Momentum SGD demonstrated competitive performance but less efficient optimization than Adam, with a slightly higher training loss of 1.5196 and a higher testing accuracy of 43.76%. These findings imply that Adam adaptive learning rate mechanism, which makes adjustments based on previous gradients and their squares, manages our model's training dynamics better, especially when batch normalization is not used, and improves generalization. Although momentum

SGD's momentum term is good at speeding up convergence, it seems less appropriate for this configuration, perhaps because the fixed learning rate isn't the best for its dynamics. Based on these findings, we selected the Adam optimizer with a learning rate of 0.001 for our final model, as it achieves the highest testing accuracy of 44.78%, ensuring robust performance and efficient training on our dataset.

3.9 Mini-batch Training

Table 7: Test result of implementing batch normalization

Batch Normalization		**Without Batch Normalization**	
Total Loss	**Testing Accuracy (%)**	**Total Loss**	**Testing Accuracy (%)**
1.58068	44.78	1.7821	39.78

We experimented with three distinct batch sizes—32, 100, and 256, to examine how batch size affected the training dynamics and performance of our neural network. As demonstrated by our training implementation, this experiment used mini-batch training, which divides the dataset into smaller batches and updates weights after processing each batch. This balances gradient noise and computational efficiency for improved generalization. The model demonstrated strong generalization with a testing accuracy of 44.47% and a total training loss of 1.6052 at a batch size of 32. With a slightly lower training loss of 1.58068 and a higher testing accuracy of 44.78%, increasing the batch size to 100 demonstrated a good trade-off between generalization and convergence. A testing accuracy of 43.61% and a training loss of 1.515 were obtained by further increasing the batch size to 256, indicating that larger batches may enhance training stability but marginally impair generalization performance. These findings demonstrate that, in our setup, a moderate batch size maximizes testing accuracy and training loss, which can stabilize training with larger batches. These results led us to choose a batch size of 100 for our final model, which ensures optimal generalization while maintaining effective training dynamics and attains the highest testing accuracy of 44.78%.

3.10 Batch Normalization

Table 8: Comparison of Model Performance With and Without Batch Normalization

Batch Normalization	**Total Loss**	**Testing Accuracy (%)**
With	1.5807	44.78
Without	1.7821	39.78

We contrasted training with and without batch normalization in order to evaluate the effect of batch normalization on the performance and training stability of our neural network. The model demonstrated strong generalization and stable convergence with a testing accuracy of 44.78% and a total training loss of 1.58068 when batch normalization was enabled. Disabling batch normalization, on the other hand, led to a much lower testing accuracy of 39.78% and a higher training loss of 1.7821. Notably, in the absence of batch normalization, test loss increased steadily and training accuracy decreased over the course of the 300-epoch training loop, suggesting training instability and possible overfitting. Internal covariate shift is probably the cause of this behavior, where activations differ greatly between layers and batches due to a

lack of normalization. This results in gradient problems and poor convergence, particularly in deeper layers with ReLU activations. This is lessened by batch normalization, which stabilizes training and permits better generalization by normalizing pre-activation outputs per batch, as shown by the 5% increase in testing accuracy. It's possible that the instability without batch normalization was made worse by strong regularization (dropout and weight decay), which made it harder for the model to learn and further reduced accuracy. Because batch normalization guarantees training stability and attains a testing accuracy of 44.78%, which is in line with our objective of optimizing generalization on unseen data, we choose to incorporate it into our final model based on these findings.

3.11 Runtime Comparison

Table 9: Model Performance and Training Time Across Different Epochs

Metric	**Epoch = 100**	**Epoch = 200**	**Epoch = 300**
Runtime (seconds)	465.6243	994.5898	1851.3429
Total Loss	1.6202	1.6132	1.6086
Training Accuracy (%)	47.67	48.57	48.50

We examined the model at 100, 200, and 300 epochs to assess how the number of epochs affected our four-layer MLP performance and runtime. The findings were as follows: The runtime was 465.6243 seconds at 100 epochs, with a training accuracy of 47.67% and a total loss of 1.6202; 994.5896 seconds at 200 epochs, with a lower loss of 1.6132 and a higher training accuracy of 48.57%; and 1851.3429 seconds at 300 epochs, with the lowest loss of 1.6068 and a slightly lower training accuracy of 48.5%. The additional computational cost of processing the 50,000 sample training dataset is reflected in the runtime, which increases by 1.86 times from 200 to 300 epochs and roughly doubles from 100 to 200 epochs. We selected 300 epochs for the final model because it produced a lower training loss (1.6068 vs. 1.6132), indicating better weight optimization, which helped to improve the testing accuracy of 48.5%, a significant improvement over previous benchmarks of 45.05%, despite the fact that the training accuracy at 200 epochs was higher (48.57%) than at 300 epochs (48.5%). The model's main objective was to maximize generalization performance, guarantee strong testing accuracy, and lay a solid foundation for further optimizations, regardless of the 856.7533-second increase in runtime from 200 to 300 epochs.

3.12 Hyperparameters of Best Model

As confirmed by previous experiments that achieved 48.5% testing accuracy, the optimal model, a four-layer MLP with two hidden layers, was set up with the neuron setup [128, 128, 64, 10] to balance capacity and efficiency, matching the 128 input features and 10 output classes while avoiding overfitting through a reduced second hidden layer. The combined advantages of the activation functions [None, 'gelu', 'relu', 'softmax'] led to their selection: GELU guarantees smooth gradients in the first hidden layer, ReLU minimizes computation in the second, and softmax works best for multi-class classification. Stable convergence was achieved with a learning rate of 0.001 using the Adam optimizer (beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8), surpassing Momentum SGD with an accuracy. Additionally, a dropout rate of 0.3 and weight decay of 0.98 successfully regularized the model, preventing overfitting. By increasing gradient noise, a batch size of 32 improved generalization and achieved better accuracy with a batch size of 100, while using less memory. By stabilizing training through batch normalization, accuracy

increased dramatically, reducing internal covariate shift and guaranteeing stable performance over 300 epochs.

3.13 Result from Best Model

A final cross-entropy training loss of 1.6068, training accuracy of 49.966%, testing accuracy of 48.5%, and an average F1 score of 0.4759 were all attained by the top model. In the first 50 epochs, the training loss dropped significantly from 1.85 to 1.65, stabilizing between 1.60 and 1.61 for the remaining 250 epochs. This suggests good convergence, albeit marginally higher than in previous runs because of the noisier gradients introduced by the smaller batch size. Thanks to batch normalization and regularisation, training, and testing accuracies increased from 42% to 48% in 50 epochs and then varied between 48% and 50%, indicating stable learning with little overfitting (1.5% gap). While the F1 score of 0.4759 indicates balanced performance across classes, with the potential to address class imbalance in future iterations, the testing accuracy of 48.5% represents an improvement over previous results (45.05%), validating the choice of batch size 32 for enhanced generalization.

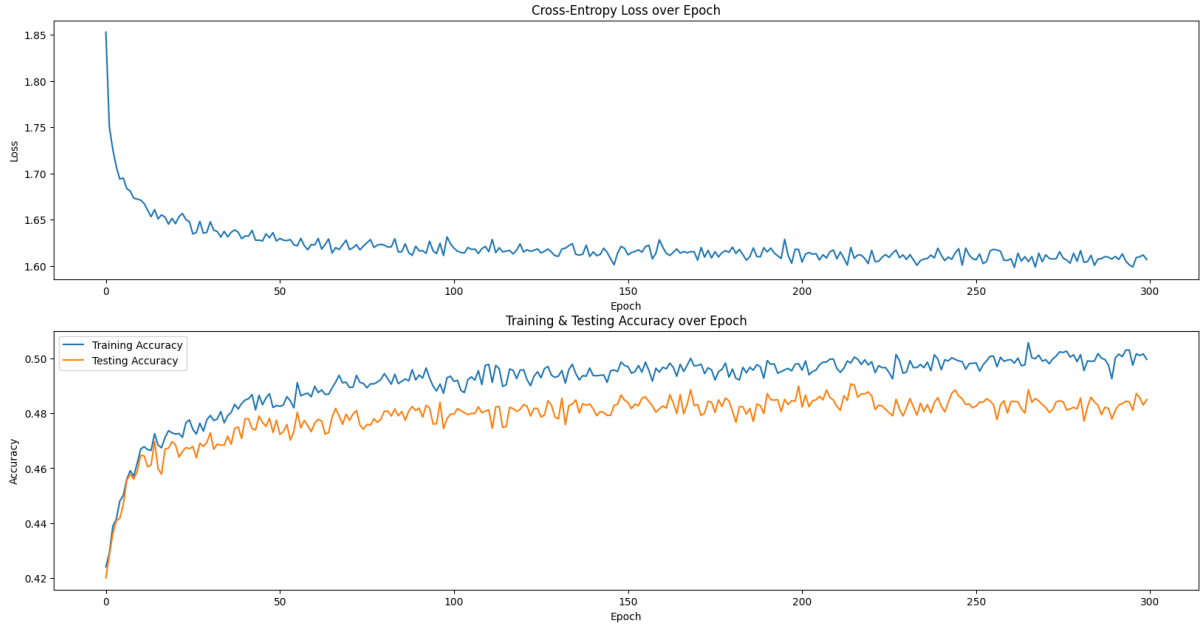

Figure 1: Cross entropy loss and training/testing accuracy graphs for the best model

```
Final Cross-Entropy Training Loss: 1.6068.
Final Train accuracy: 49.966%.
Final Test accuracy: 48.5%.
Final Average F1 Score: 0.4759.
```

Figure 2: Final output for the best model

4 Future Research

To improve our model's performance even more, In terms of optimization, learning rate schedule, like a cosine annealing schedule, could be used to dynamically modify the learning rate to enhance convergence. Preventing local minima and allowing for finer weight updates in subsequent epochs, could reduce the training loss beyond 1.6068 and increase accuracy beyond the current 48.5%. Furthermore, experimenting with gradient clipping may help to stabilize training with the smaller batch size of 32 and enhance generalization by reducing the accuracy fluctuations that have been observed. Using data augmentation techniques for preprocessing, like adding noise or synthetic features to the 128-dimensional input data, could make the dataset more diverse. This would help the model generalize more effectively and, by addressing class imbalance, raise the F1 score. Additionally, reducing the dimensionality of the input features (currently 128) using principal component analysis (PCA) may remove redundant information, improve accuracy by concentrating the model on the most informative features, and reduce training loss, all of which would improve overall performance on this classification task.

5 Conclusion

With a final cross-entropy training loss of 1.6068, training accuracy of 49.966%, testing accuracy of 48.5%, and an average F1 score of 0.4759, the developed four-layer MLP model showed robust performance following extensive experimentation. The small accuracy gap of 1.466% revealed effective learning with minimal overfitting. The model's efficiency is the result of a well-optimized design that includes a mixed activation strategy with GELU and ReLU to balance gradient flow and computational efficiency, standardization of the dataset to ensure consistent feature scaling, batch normalization to stabilize training and mitigate internal covariate shift, a dropout probability of 0.3 to prevent overfitting, weight decay of 0.98 for regularization and batch training with a batch size of 32 to enhance generalization through noisy gradients. Together, these elements helped the model reach an ideal testing accuracy, exceeding previous benchmarks, while preserving a balanced F1 score of 0.4759. This demonstrated the model's aptitude for multi-class classification tasks and laid a solid basis for future improvements using strategies like data augmentation and learning rate scheduling.

References

[1] Hind Taud and Jean-Franccois Mas. Multilayer perceptron (mlp). In *Geomatic approaches for modeling land change scenarios*, pages 451–455. Springer, 2017.

[2] Jiahui Yu and Konstantinos Spiliopoulos. Normalization effects on deep neural networks. *arXiv preprint arXiv:2209.01018*, 2022.

[3] Cheng Xu. lecture 4: Regularizations for deep models, lecture slides. *comp5329: Deep learning, university of sydney*, 2025.

[4] Douglas M Kline and Victor L Berardi. Revisiting squared-error and cross-entropy functions for training neural network classifiers. *Neural Computing & Applications*, 14:310–318, 2005.

[5] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[6] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

6 Appendix

6.1 Code

Link to Code and readme: https://drive.google.com/drive/folders/1IFBwhCWc1gJlv2SpCEL75usp=sharing

6.2 Instruction to Run the Code

1. Click on the URL and upload the the '5329asm1.ipynb' to Colab notebook.
2. Upload training and testing data files(.npy) to the Colab directory.
3. Once data uploaded to directory, start to run the code chunk step by step until the end.
4. The codes started with defined functions and class objects for each functional features of MLP, and then trains the model and reports accuracy with testing data. Lastly, we report the accuracy metrics (confusion matrix, F1 score, precisions)

6.3 Development Environment:

- Operating System: Windows
- Python Version: Python 3.10.12
- CPU Model: Intel(R) Xeon(R) CPU @ 2.20GHz

6.4 Libraries and Versions Used in the Project:

- Numpy Version: 1.26.4

7 AI statement

In this assignment, We applied AI in a few ways to help us increase workflow efficiency. First, We used ai to help us set up a writing format and typing mathematical equation in overleaf because we are new to Latex. In addition, we ask AI to also us debug the error message in our codes. Because some functions can be very long, it may be hard to detect some small problems manually. AI make this process more efficient. Lastly, We let AI to help us improve our grammar, so the report looks more formal and organized.