THE UNIVERSITY OF SYDNEY

The University of Sydney

School of Comupter Science

Comp5329 Assignment1

Author:
Haixu Liu hliu2490
Zerui Tao ztao0063

Supervisor:
Chang Xu

An Assignment report submitted for the Comp5329 Assignment1:

April 12, 2024

Contents

Abstract

This paper explores how various functions, optimization strategies, and functional modules can be leveraged to enhance the performance of deep learning models in multiclass classification tasks. Additionally, the paper employs multiple evaluation metrics to assess the performance of these models, with the best-performing model being identified and elucidated based on a composite of these metrics. GPT is used to check the correctness of code and to polish text.

1 Introduction

1.1 Important of Study

Since real-world classification issues often extend beyond binary classification, necessitating categorization into multiple specific classes, the investigation of multiclass models is crucial. These models enhance data analysis and interpretation, substantially supporting real-world applications. Furthermore, real-world classification tasks often entail handling large datasets, where multiclass models are more adaptable and practical. Additionally, deep learning techniques facilitate automatic feature extraction, significantly aiding the construction of multiclass models. The significance of multiclass classification research is underscored by these models' pivotal role across various industry sectors.

1.2 Aim of Study

This paper aims to develop a deep learning model capable of executing multiclass classification tasks, incorporating at least two hidden layers. The model will undergo modifications through various training and optimization strategies to achieve optimal performance. Its effectiveness will be gauged using diverse evaluation metrics to determine the best overall performing model.

1.3 Introduction of Dataset

The training set in the dataset employed for this study comprises 50,000 samples evenly distributed across ten distinct labels, with each label represented by 5,000 samples. Each sample possesses 128 unique features and is devoid of missing values. The test dataset contains 10,000 samples, each also characterized by 128 features.

2 Methodology and Principle of technology

2.1 Pre-processing the data

Through initial data exploration, as shown in Figure 1, we identified that the dataset consists of 50,000 training and 10,000 test entries, covering 128 feature variables and

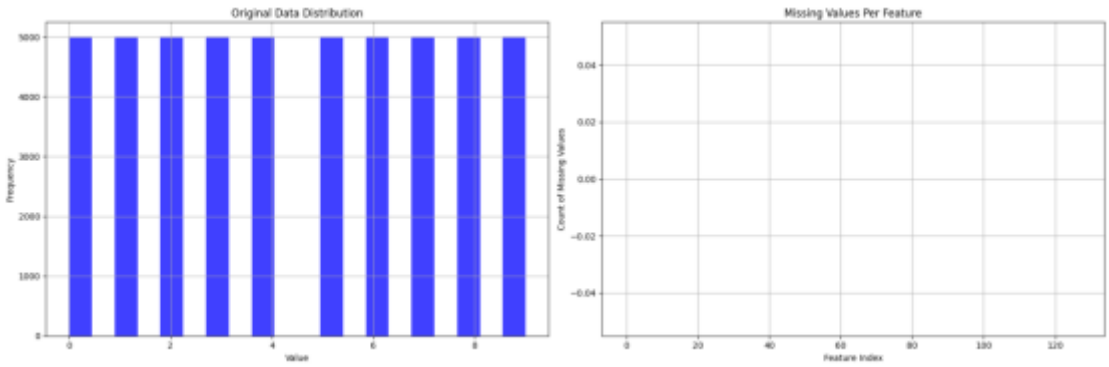


Figure 1: Deatils of dataset

ten classification categories. It exhibits no missing values and maintains balance with each category uniformly represented by 5,000 data points, thus negating data imbalance concerns. Further analysis of feature distribution, detailed in Appendix Figure 1, reveals that all features conform to a normal distribution with no significant scale variation.
Consequently, our preprocessing involves normalizing the data to a (0,1) range, converting labels into one-hot encoded format, and shuffling the labels to eliminate any inherent order, thereby randomizing their sequence.

2.1.1 Data Normalization

As previously stated, we normalized the dataset to the range [0,1] using the following formula (see Formula 1). This scaling method guarantees that every feature in the dataset is proportionately adjusted between 0 and 1, which is essential for ensuring the uniform contribution of all features to the model's training process.
Following this, we applied one-hot encoding to the labels, transforming each label into a binary vector of length equal to the number of classes. In this vector, the position corresponding to the label number is assigned a value of 1, while all other positions are set to 0.

$$X_{normalization} = \frac{X - X_{min}}{max(X) - min(X)} \tag{1}$$

2.1.2 Data Standardization

This paper employs data standardization as a processing method to normalize feature values to a standard scale, which is particularly beneficial for algorithms sensitive to input data scales. The normalization formula is presented in Equation 2, where x represents the original data point, μ is the mean of the data points for that feature, σ is the standard deviation, and z denotes the normalized data point.

$$z = \frac{(x - \mu)}{\sigma} \tag{2}$$

Standardization accelerates gradient descent and diminishes feature interdependence in the model, reducing the disproportionate impact of any single feature on model performance. Consequently, this ensures a more equitable consideration of all features in the model.

2.1.3 Label One-hot Encoding

In this paper, the one-hot encoding method is utilized to transform the sample labels in the dataset into one-hot vector form. This method ensures a distinct, non-overlapping representation for each category, eliminating potential biases due to value magnitudes. One-hot encoding allows the model to more accurately interpret category information, as it positions each category at an equal distance in the feature space, thereby enhancing model performance. Furthermore, one-hot encoding distinctly differentiates each category, enabling the model to accurately identify and distinguish among all potential output categories. Therefore, one-hot encoding is highly suited to the task addressed in this paper.

2.2 Multiple Hidden Layer Class Module

The hidden layer is divided into two components for analysis: forward propagation and backpropagation. In forward propagation, each neuron in the hidden layer computes the weighted sum of its inputs. Then, each neuron is processed through a nonlinear activation function, which is mathematically represented by Formulas 3 and 4.

$$y_j = \sum_{i=1}^{n} w_{ji} x_i + b_j \tag{3}$$

$$a_j = f(y_j) \tag{4}$$

In this context, x_i denotes the inputs of the hidden layer, w_{ji} represents the weights in the weight matrix, $b_j$is the bias, and f signifies the activation function.

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial W} = \frac{\partial L}{\partial Z}X^T \tag{5}$$

$$\frac{\partial L}{\partial b} = \sum \frac{\partial L}{\partial Z} \tag{6}$$

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial A}\frac{\partial A}{\partial Y}\frac{\partial Y}{\partial X} = \frac{\partial L}{\partial A}\frac{\partial A}{\partial Y}W^T \tag{7}$$

The backpropagation section accounts for weight decay and momentum, with the gradient detailed in Formulas 5, 6, and 7. Formulas 8 and 9 depict the velocity of the weights and bias. Finally, Formula 10 and Formula 11 illustrate the update process.

$$v_w = \mu v_w - \eta\left(\frac{\partial L}{\partial W} + \lambda W\right) \tag{8}$$

$$v_b = \mu v_b - \eta\frac{\partial L}{\partial b} \tag{9}$$

$$W = W - v_w \tag{10}$$

$$b = b - v_b \tag{11}$$

2.3 Activation Function

2.3.1 Sigmoid Activation Function

The sigmoid function[1] is one of the earliest activation functions, as defined by Formula 12. From Formula 11, we can derive the derivative of the sigmoid function, which is shown in Formula 13. This function maps all inputs to a range between 0 and 1, making it ideal for binary classification. Characterized by an S-shaped curve, the sigmoid function exhibits rapid value changes around zero. However, it saturates towards the function's extremities, causing the gradient to approach zero, leading to the vanishing gradient problem. This issue can significantly slow the learning process or halt further network training.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{12}$$

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x)) = \frac{e^{-x}}{(1+e^{-x})^2} \tag{13}$$

2.3.2 Tanh Activation Function

The hyperbolic tangent function (tanh)[1] is a frequently used activation function in deep neural networks. Its mathematical definition is provided in Formula 14, and the derivative of the tanh function is given in Formula 15. Unlike the sigmoid function, tanh is zero-centered, with an output range from -1 to 1. This characteristic makes Tanh particularly effective when dealing with data samples that exhibit significant feature variance, amplifying the effect of features as the training progresses. However, tanh is susceptible to the vanishing gradient problem similar to the sigmoid function.

$$tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{14}$$

$$tanh^{'}(x) = 1 - tanh^{2}(x) \tag{15}$$

2.3.3 Relu Activiation Function

The Rectified Linear Unit (ReLU)[1] is an activation function widely employed in neural networks to introduce non-linearity, especially in the hidden layers of deep learning models. It is defined mathematically in Formula 16, while its derivative is presented in Formula 17.

$$f(x) = max(0,x) \tag{16}$$

$$\frac{df(x)}{dx} = \begin{cases} 1, \text{ if } x > 0 \\ 0, \text{ if } x \leq 0 \end{cases} \tag{17}$$

ReLU facilitates the approximation of complex functions by adding non-linearity to the model. Unlike traditional activation functions such as the sigmoid function or Tanh function, where gradients diminish significantly as the input's absolute value increases (a phenomenon known as gradient vanishing), ReLU alleviates this issue with its gradient being consistently 0 for negative inputs and 1 for positive inputs. This attribute

ensures a robust gradient flow during backpropagation. Moreover, ReLU is computationally less intensive than the exponential functions used in sigmoid or tanh activations, enhancing computational efficiency. However, a drawback of ReLU is that neurons outputting negative values will have a constant gradient of 0, preventing updates based on the loss. It can result in a substantial portion of the network becoming inactive, thus potentially diminishing the model's capacity.

2.3.4 Leakly Relu Activation Function

Leaky ReLU[1] was introduced to address the drawbacks of the ReLU function. This variant permits a small, nonzero, constant gradient for inactive units when the input is negative. It is defined in Formula 18, and its derivative is presented in Formula 19.

$$f(x) = \begin{cases} x, \text{ if } x > 0 \\ ax, \text{ if } x \leq 0 \end{cases} \tag{18}$$

$$\frac{df(x)}{dx} = \begin{cases} 1, \text{ if } x > 0 \\ a, \text{ if } x \leq 0 \end{cases} \tag{19}$$

Leaky ReLU rectifies the dying neuron issue inherent in ReLU while retaining all its benefits. With a slight positive slope in its negative domain, Leaky ReLU also facilitates backpropagation for negative inputs. However, the inconsistency in function intervals means Leaky ReLU cannot uniformly predict relationships for both positive and negative inputs.

2.3.5 GELU Activiation Function(Advanced)

ReLU, dropout, and zoneout inspire the Gaussian Error Linear Unit (GELU) activation function[4]. ReLU deterministically multiplies the input by zero or one, while dropout and zoneout randomly multiply the inputs by zero and one, respectively. GELU similarly modulates the inputs by zero or one but with a masking mechanism that is randomly determined and also input-dependent. It is defined in Formulas 20, formula 21 and formula 22, and its derivative is presented in Formula 23.

$$GELU(x) = x \cdot \phi(x) = x \cdot \frac{1}{2}[1 + erf(\frac{x}{\sqrt{2}})] \tag{20}$$

$$erf(x) = \frac{2}{\sqrt{(\pi)}} \int_0^x e^{-x^2} dx \tag{21}$$

$$\phi(x) = \frac{1}{2}[1 + erf(\frac{x}{\sqrt{2}})] \tag{22}$$

$$GELU^{'}(x) = \frac{1}{2}[1 + erf(\frac{x}{\sqrt{2}})] + x \cdot \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \tag{23}$$

GELU is a smooth function, ensuring that its derivatives are continuous. This smoothness is advantageous for gradient-based optimization, providing more stable and consistent gradients throughout the input range. GELU features an adaptive gating mechanism that dynamically adjusts neuron activation based on the input. Consequently, large inputs lead to activation, whereas small inputs suppress activation. This adaptability enables the model to manage variations in the input data more effectively.

2.3.6 Softmax Activiation Function

The softmax activation function[1] extends the sigmoid function, which can work for multi-class classification. It maps a K-dimensional vector to another vector with values between 0 and 1 that sum to 1. It is mathematically defined in Formula 24.
In formula 24, z_j is the jth element in the vector x, representing the original predicted value for the jth category.$e^{z_j}$the exponent of z_j is computed. The exponential function ensures that all outputs are positive, which is necessary to interpret these values as probabilities. K is the total number of categories. The denominator of the Softmax function is summed over all k categories, ensuring that the probabilities of all categories sum to 1.

$$P(class_j|x) = \frac{e^{z_j}}{\sum_{j=1}^{k} e^{z_j}} \tag{24}$$

The Softmax output is the predicted probability for each class. It is typically utilized as the final layer in a neural network. Softmax amplifies the differences between outputs, employing the exponential function to enhance the classification's confidence.

2.4 Weight Decay Strategy

This paper explores using a weight decay strategy[3] to mitigate model overfitting and enhance model generalization. The typical form of this regularization involves adding an L2 term, defined in Formula 25, with its derivative presented in Formula 26

$$\min_{\theta} \hat{L}_R(\theta) = \widehat{L}(\theta) + \frac{\alpha}{2}\|\theta\|_2^2 \tag{25}$$

$$\bigtriangledown \hat{L_R}(\theta) = \bigtriangledown \hat{L}(\theta + \alpha\theta) \tag{26}$$

α is the hyperparameter controlling the strength of the regularization penalty; a low α value might result in overfitting, while a α lambda can prevent it. $\hat{L_r}(\theta$ is the loss function after regularisation and $\hat{L}(\theta$ is the original loss function.
Analysis of Formulas 25 and 26 indicates that the gradient descent update rule is depicted in Formula 27.

$$\theta \leftarrow (1 - \eta\alpha)\theta - \eta \bigtriangledown \hat{L}(\theta) \tag{27}$$

In Equation 27, θ represents the model's weight parameter, which is learned and updated throughout the training process. η denotes the learning rate, governing the step size of the weight update during each iteration. α is the regularization coefficient that regulates the intensity of the regularization.
Large weights can destabilize training, as these weights excessively magnify minor input changes, leading to disproportionately large outputs after several layers. In backpropagation, larger weights produce more significant gradients, which can escalate through the chain rule, causing the exploding gradients problem. Therefore, weight decay helps maintain weights within a suitable range, contributing to a more stable training process.

2.5 Momentum in SGD

Stochastic Gradient Descent (SGD)[2] often struggles with navigating narrow valleys, leading to the introduction of momentum to expedite SGD's convergence in the desired direction and mitigate oscillations. Momentum increases updates in dimensions where the gradient direction remains consistent and decreases updates in dimensions with changing gradients. This process is mathematically represented by Formulas 28 and 29.

$$v_t = \alpha v_t + \eta \bigtriangledown L(\theta) \tag{28}$$

$$\theta_t = \theta_{t-1} - v_t \tag{29}$$

v_t denotes the velocity vector in formula 28 and formula 29, which is initially set to zero. α is the momentum coefficient, which dictates the degree to which the previous velocity is retained, enhancing convergence towards directions, consistently reducing the loss and diminishing the impact of noisy gradients. Momentum SGD mitigates oscillations by computing a weighted average of previous gradients, thus smoothing the update trajectory. Additionally, the momentum term aids the model in escaping local minima and saddle points during optimization. This effect is due to its retention of directional persistence in gradient accumulation, facilitating traversal over minor depressions and flat areas. Momentum facilitates smoother navigation through local minima but adds complexity to the model and requires additional computational resources.

2.6 Dropout Module

Dropout[5] is a regularization technique employed in neural networks to prevent overfitting. In its basic form, during model training, each neuron is randomly deactivated with a predefined probability, allowing only the remaining neurons to be updated through backpropagation. During prediction, however, all neurons are utilized, and their weights are adjusted by the same predefined probability. The activity of unit i in layer h can be mathematically represented as Formula 30 and Formula 31.

$$S_i^h = \sum_{l<h} \sum_j w_{ij}^{hl} \delta_j^l S_j^l \tag{30}$$

$$S_j^0 = I_j \tag{31}$$

In formula 30 and formula 31, S_i^h denotes the ith neuron's activity (or output value) in layer h. $\sum_{l<h}$ represents a summation over all layers l less than h, indicating that the activity of the current neuron accumulates the contributions from neurons in all preceding layers. $\sum_j$ signifies a summation over all neurons j in layer l, showing that the activity of the current neuron is affected by the activities of all neurons within that layer.w_{ij}^{hl}denotes the weight between the jth neuron in layer l and the ith neuron in layer h. This weight influences the magnitude of the contribution from the neuron in the previous layer to the activity of the current neuron. δ_j^lrepresents an indicator variable determining whether the jth neuron of layer l is subjected to dropout during training. If $\delta_j^l = 1$, the neuron is active in both forward and backward propagation; if $\delta_j^l = 0$, it is excluded from the current training step. S_j^l denotes the activity of the jth neuron of layer l.
Dropout mitigates overfitting risk by randomly deactivating neurons during training,

reducing the model's reliance on specific training data samples. This enhances performance on unseen data. Dropout essentially samples numerous sub-networks from the original network, training them independently. These networks are amalgamated during testing by scaling the weights, typically improving the model's generalization. As dropout randomly discards neurons in the input and hidden layers in each training iteration, it prevents the model from depending on any single feature, thereby fostering a more robust feature representation.

2.7 Cross-entropy Loss Function

Cross-entropy[1] is a crucial metric in classification tasks, assessing the match between predicted probabilities and actual labels and the dissimilarity between two probability distributions. For multi-class classification, the loss function is delineated in Equation 30. Cross-entropy quantifies the average coding length required for an event when coded with the probability distribution Q predicted by the model against the true distribution P. Here, P represents the distribution of proper labels, and Q is the model's predicted probability distribution. A lower cross-entropy indicates a closer alignment of the model's predictions with the true distribution.

$$H(P,Q) = -\sum_{c=1}^{M} y_c log(\hat{y_c}) \tag{32}$$

In the formula 32, M is the number of types. y_c denotes the true category. $\hat{c}$ denotes the probability distribution of predictions, indicating the probability that the prediction belongs to category c.
Cross-entropy loss imposes a higher penalty for correct but uncertain predictions, compelling the model to be accurate and confident. This characteristic enhances its effectiveness in optimizing classification performance. Cross-entropy provides a steeper gradient than other loss functions as the model nears the correct response, facilitating faster learning, notably when the model's output markedly deviates from the actual labels. Moreover, it sustains a more significant gradient later in the training process, preventing a sharp decline in the learning rate.

2.8 Mini-batch Training Strategy

Mini-batch training strategy[2] balances batch gradient descent and stochastic gradient descent to update weights based on a subset of the training data. The update mechanism is described in Formula 33. In this formula, θ represents the parameters, η the learning rate, L the loss function and $X^{(i:i+n)}$ and $y^{(i:i+n)}$ correspond to the n input data samples and labels of the mini-batch, respectively. In practice, this method involves processing multiple data points simultaneously during training, culminating in a single weight update based on these data points.

$$\theta = \theta - \eta \bigtriangledown_\theta L(\theta; X^{(i:i+n)}, y^{(i:i+n)}) \tag{33}$$

2.9 Batch Normalization

Batch normalization (BN)[3] is a technique that mitigates covariate shift by normalizing each layer's inputs to a standard normal distribution, facilitating faster gradient descent convergence. For a given mini-batch B of size m, the normalization process is mathematically detailed in Formulas 34, 35, 36, and 37, where the input values x_i in the mini-batch are adjusted.

$$\mu_B = \frac{1}{n}\sum_{i=1}^{m} x_i \tag{34}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{35}$$

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{36}$$

$$y_i = \gamma \hat{x_i} + \beta \tag{37}$$

A small constant ε is added to the variance to prevent division by zero. At the same time, γ and β are learnable hyperparameters that maintain the representational capacity of the layer, optimizing it for the model. The BN layer's output serves as the input for the subsequent layer is y_i. During backpropagation, the gradient is computed as per Formulas 38, Formula 39, Formula 40, Formula 41, Formula 42 and Formula 43. Each batch normalization layer has a mean and variance for each feature dimension, updated throughout training. Utilizing these updated statistics can enhance inference results. A momentum strategy averages the statistics across batches for stability during training, ensuring consistent statistics.

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial \gamma} \tag{38}$$

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} \tag{39}$$

$$\frac{\partial L}{\partial \hat{x_i}} = \frac{\partial L}{\partial y_i}\gamma \tag{40}$$

$$\frac{\partial L}{\partial \sigma_B^2} = \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x_i}}\frac{\partial \hat{x_i}}{\partial \sigma_B^2} \tag{41}$$

$$\frac{\partial L}{\partial \mu_B} = \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x_i}}\frac{\partial \hat{x_i}}{\partial \mu_B} + \frac{\partial L}{\partial \sigma_B^2}\frac{\partial \sigma_B^2}{\partial \mu_B} \tag{42}$$

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \hat{x_i}^2}\frac{1}{\sqrt{\sigma_B^2 + \varepsilon}} + \frac{\partial L}{\partial \sigma_B^2}\frac{2(x_i - \mu_B)}{m} + \frac{\partial L}{\partial \mu_B}\frac{1}{m} \tag{43}$$

In practice, BN reduces covariate shifts, enhances training stability, allows higher learning rates and fewer training iterations, and normalizes inputs to prevent extreme value scales in each layer. This normalization aids in controlling gradient magnitudes during backpropagation and prevents activation functions like the sigmoid from saturating.

2.10 Layer Normalization(Advanced)

The implementation of batch normalization is dependent on the mini-batch size and is not ideally suited for recurrent neural networks (RNNs). In contrast, layer normalization[6] computes the mean and variance using the summed inputs to the neurons within a single layer for each training case. The calculations for the mean and variance in layer normalization are delineated in Formulas 43 and 44.

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \tag{44}$$

$$\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2} \tag{45}$$

Where H is the number of hidden units in the layer, and a_i^l represents the summed input to the i^{th} hidden unit in the l^{th} layer. Unlike batch normalization, layer normalization does not calculate the mean and variance based on the batch size, making it more suitable for small batch sizes. Additionally, because layer normalization (LN) normalizes the inputs based on the input data's statistics, its behaviour remains consistent across both the training and inference phases.

2.11 Kaiming Initial

Assuming the activation function is ReLU, the Kaiming initialization method[2] sets the weight distribution to align with the distribution of the neurons' outputs and inputs. The weight initialization formula is typically specified to ensure this alignment, optimizing the network's performance with ReLU activations.

$$w \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n}}\right) \tag{46}$$

2.12 The Design of the Best Model

Based on the above theoretical analysis, I think the optimal model structure is as follows: in addition to the most basic input and output layers, it should have 1-2 hidden layers with twice as many neurons as the input layer. This allows the model to fully learn the higher-order features from the original features without sacrificing too much training and inference efficiency. At the same time each linear layer should be followed by a batch norm to improve the training stability and an activation function such as GELU or ReLU should be used to add nonlinear computation to the model. Finally, a dropout layer with a smaller dropout rate should be used to reduce overfitting. In order to ensure that the high-level features of each layer are fully utilised and to reduce the gradient vanishing or gradient explosion, we consider adding residual connections between the hidden layers to further improve the model performance.

3 Experiments

3.1 Metrics and Running Requirements

3.1.1 Running Requirements

Our hardware requirements are as follows: For running regular comparative experiments and ablation studies, you can choose CPU mode in the Colab environment and select high RAM. However, for executing the final extended comparative experiments section (Self Attention), please opt for the A100 GPU in the Colab environment and select high RAM to ensure a minimum of 64GB of memory. The operation of running the code is described in the appendix.

3.1.2 Accuracy

Accuracy measures the proportion of correctly classified samples (both positive and negative) out of the total number of samples. It is calculated based on the concepts of True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). The formula for accuracy is given by formula 47:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{47}$$

3.1.3 Recall

Recall is the proportion of samples correctly identified as positive out of all actual positive samples. It is also known as the true positive rate. Here is the formula for recall:

$$Recall = \frac{TP}{TP + FN} \tag{48}$$

3.1.4 Precision

Precision is the proportion of samples correctly identified as positive out of all samples identified as positive. The formula of precision is shown as formula 49.

$$Precision = \frac{TP}{TP + FP} \tag{49}$$

3.1.5 F1 Score

The F1 Score is the harmonic mean of precision and recall, providing a balanced measure of both metrics. It is particularly useful when neither precision nor recall can be prioritized over the other due to the problem domain's requirements. The formula for the F1 Score is:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{50}$$

3.1.6 Efficiency

Besides accuracy performance, we have also chosen the duration of model training, inference time, and the number of weights as criteria for measuring performance.

3.1.7 Baseline Model

To visually demonstrate the experimental results, we selected a classic neural network architecture as the baseline model. This model is composed of an input layer, a hidden linear layer equipped with 128 neurons—matching the number of features—and a GELU activation function. The output layer contains the same number of inputs as there are classification categories (10) and includes a SoftMax activation function. The learning rate and other parameters adhere to the default settings specified in the neural network class of the Sklearn library. The structure of the model and its parameters are detailed in Tables 1 and 2.

Number of Layer	Class of Layer	Number of neurons
1	Linear Layer	(128,128)
2	GELU	-
3	Linear Layer	(128,10)
4	SoftMax	-

Table 1: Structure of Baseline Model

Parameter	Value
Weight Decay	None
Learning Rate	0.001
Momentum	0.9
Epoch	200
Batch size	200
Dropout	None

Table 2: Parameters for Baseline Model

3.2 Hyper-parameter Analysis

3.2.1 Learning Rate

We experimented with various learning rates to assess their impact on model performance. As illustrated in Table 3 and Figure 2, we ultimately determined that a learning rate of 0.05 led to the fastest decrease in loss and the best performance on both the training and test sets. This rate also required the fewest epochs and the shortest training time.

Learning Rate	Stop Epoch	Training Accuracy	Test Accuracy	Training F1	Test F1	Training times	Inference times	Parameter counts
0.001	200	0.457	0.452	0.45	0.45	247	0.26	17802
0.005	200	0.542	0.506	0.54	0.50	251	0.26	17802
0.01	200	0.582	0.528	0.58	0.52	253	0.27	17802
0.05	45	0.595	0.530	0.60	0.53	69	0.27	17802

Table 3: Experiment Result for Learning Rate

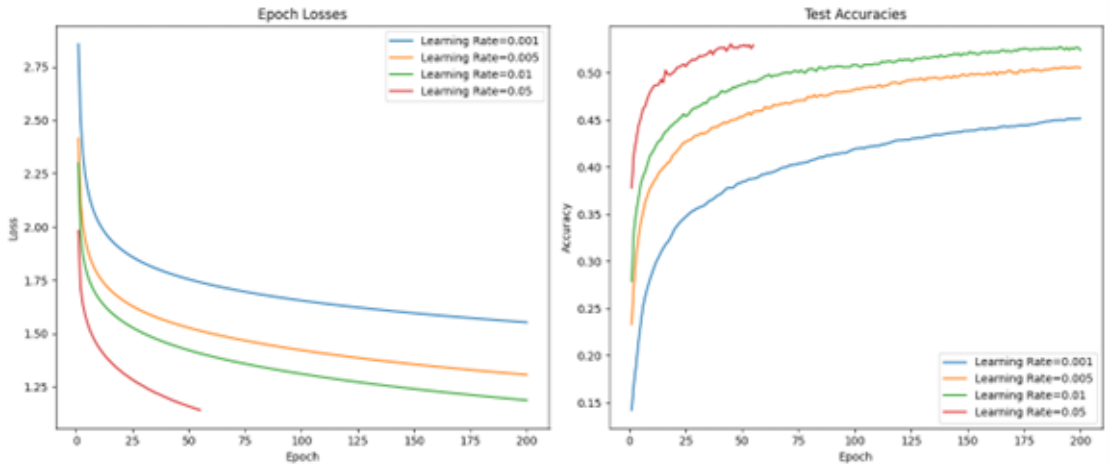


Figure 2: Experiment Result for Learning Rate

3.2.2 Batch Size

We experimented with various batch sizes to assess their impact on model performance, as detailed in Table 4 and Figure 3. We determined that the optimal batch size is 32. Although batch size does not significantly influence accuracy, smaller batch sizes facilitate faster convergence, thereby reducing training time.

Batch Size	Stop Epoch	Training Accuracy	Test Accuracy	Training F1	Test F1	Training times	Inference times	Parameter counts
200	65	0.616	0.541	0.62	0.54	93	0.28	17802
100	54	0.638	0.538	0.64	0.53	97	0.27	17802
256	86	0.621	0.533	0.63	0.53	117	0.27	17802
128	54	0.628	0.539	0.64	0.53	91	0.27	17802
64	22	0.608	0.535	0.62	0.53	54	0.27	17802
32	17	0.616	0.536	0.63	0.53	45	0.27	17802

Table 4: Experiment Result for Batch Size

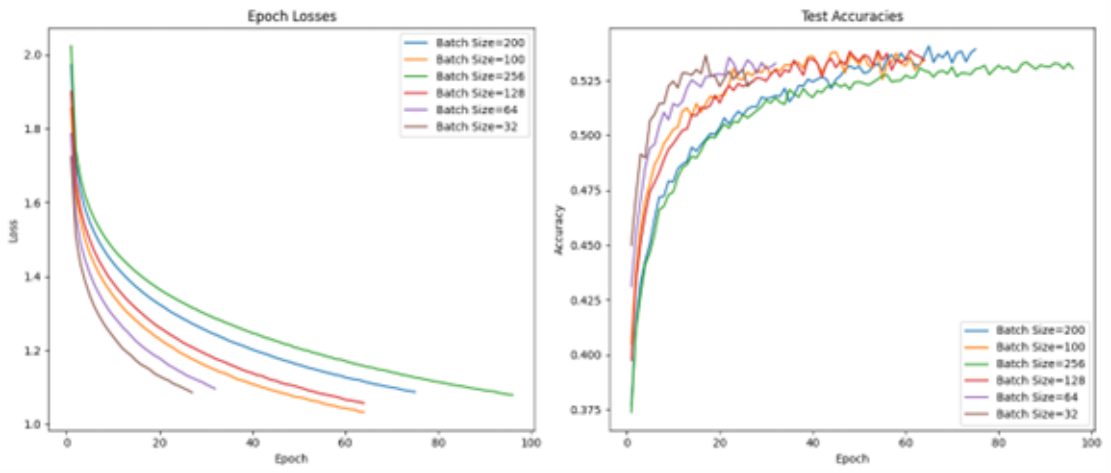


Figure 3: Experiment Result for Batch Size

3.2.3 Dropout Rate

We also tested different dropout rates to evaluate their effects on model performance, as illustrated in Table 5 and Figure 4. Based on the analysis of the test set and loss curves, we found that the most effective dropout rates are either 0.2 or not using dropout at all. While this parameter minimally impacts accuracy, a higher dropout rate is effective in preventing overfitting.

Dropout Rate	Stop Epoch	Training Accuracy	Test Accuracy	Training F1	Test F1	Training times	Inference times	Parameter counts
0.5	43	0.584	0.532	0.58	0.52	106	0.27	17802
0.3	35	0.606	0.540	0.62	0.53	89	0.27	17802
0.2	42	0.637	0.543	0.64	0.54	102	0.27	17802
0.1	30	0.634	0.541	0.64	0.54	79	0.27	17802
0.05	24	0.629	0.5395	0.63	0.53	67	0.27	17802
None	22	0.63	0.5385	0.64	0.53	53	0.29	17802

Table 5: Experiment Result for Dropout Rate

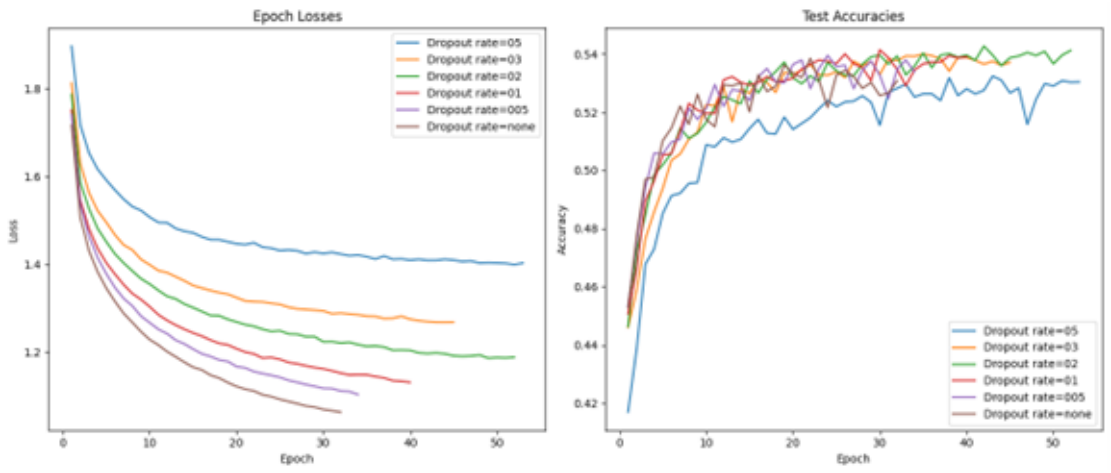


Figure 4: Experiment Result for Dropout Rate

3.2.4 Weight Decay

We experimented with various levels of weight decay to evaluate their impact on model performance, as detailed in Table 6 and Figure 5. Weight decay is effective in preventing overfitting; however, excessively high values can result in underfitting. Based on the loss curve and test set accuracy, the optimal weight decay value appears to be 0.0001.

Weight Decay	Stop Epoch	Training Accuracy	Test Accuracy	Training F1	Test F1	Training times	Inference times	Parameter counts
None	18	0.620	0.533	0.64	0.52	47	0.28	17802
0.0001	20	0.604	0.543	0.61	0.53	47	0.26	17802
0.001	25	0.485	0.481	0.48	0.47	53	0.25	17802
0.01	15	0.329	0.332	0.31	0.31	35	0.20	17802

Table 6: Experiment Result for Weight Decay

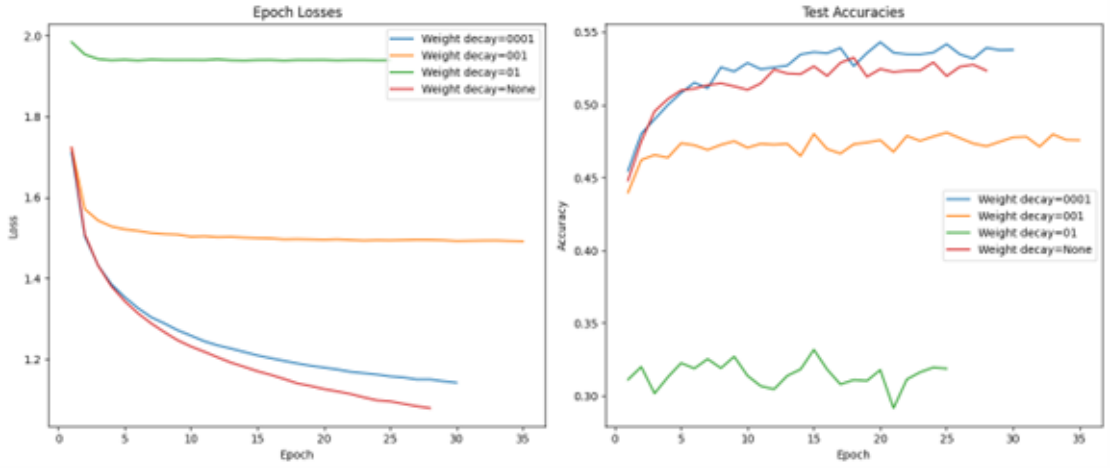


Figure 5: Experiment Result for Weight Decay

3.2.5 Momentum

We tested different momentum values to determine their influence on model performance, as depicted in Table 7 and Figure 6. While the setting of momentum does not significantly affect the final accuracy of the model, an optimal setting can greatly reduce the number of training iterations required. After considering all factors, the most suitable momentum value is determined to be 0.999.

Momentum	Stop Epoch	Training Accuracy	Test Accuracy	Training F1	Test F1	Training times	Inference times	Parameter counts
0.9	28	0.616	0.544	0.62	0.54	62	0.26	17802
0.95	30	0.614	0.547	0.62	0.54	65	0.26	17802
0.98	38	0.628	0.549	0.63	0.54	78	0.26	17802
0.99	37	0.63	0.549	0.62	0.54	76	0.25	17802
0.999	18	0.599	0.541	0.62	0.54	45	0.25	17802

Table 7: Experiment Result for Momentum

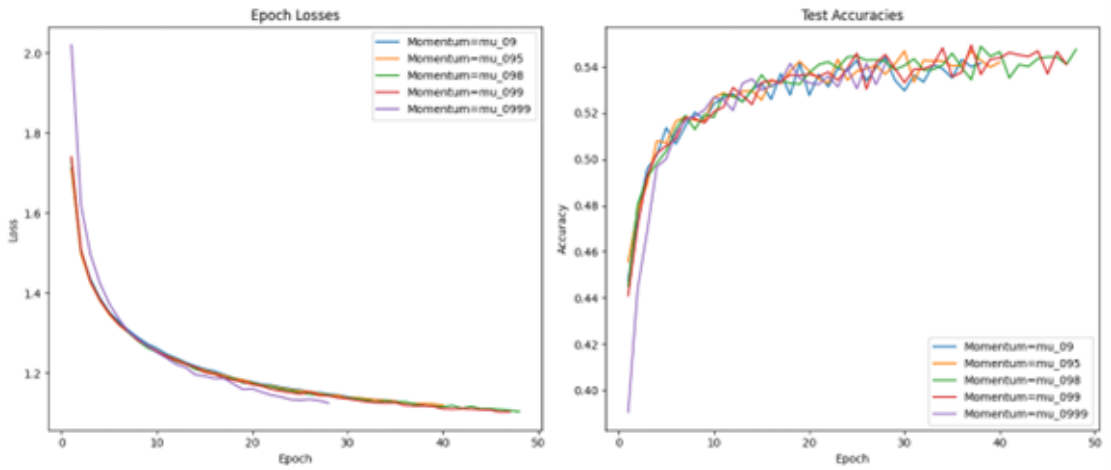


Figure 6: Experiment Result for Momentum

3.2.6 Neurons

We conducted experiments with various numbers of neurons in the hidden layers to evaluate their impact on model performance, as detailed in Table 8 and Figure 7. Increasing the number of neurons in the hidden layers generally enhances the accuracy of both the training and testing datasets. However, this can also lead to overfitting and extended training and inference times. After thorough consideration, the optimal number of neurons for this parameter was determined to be 256.

Hidden Layer Numbers	Number of neurons	Training Accuracy	Test Accuracy	Training F1	Test F1	Training times	Inference times	Parameter counts
1	256	0.682	0.562	0.68	0.56	185	0.42	35594
1	128	0.641	0.549	0.64	0.54	106	0.26	17802
1	64	0.581	0.536	0.57	0.52	64	0.16	8906
1	200	0.630	0.548	0.65	0.54	63	0.35	27810
1	150	0.609	0.545	0.63	0.54	47	0.28	20860
1	100	0.609	0.540	0.61	0.53	62	0.21	13910
1	50	0.563	0.523	0.55	0.51	44	0.11	6960

Table 8: Experiment Result for Number of Neurons

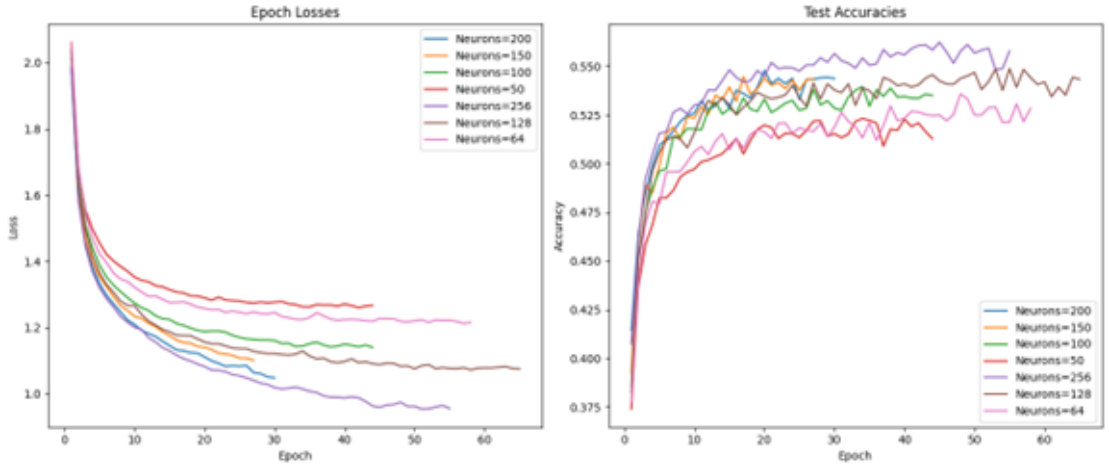


Figure 7: Experiment Result for Number of Neurons

3.2.7 Layers

We conducted experiments with various numbers of hidden layers to evaluate their impact on model performance, as shown in Table 9 and Figure 8. Increasing the number of hidden layers typically enhances performance on the training set; however, it also tends to lead to overfitting and longer training and inference times. After careful consideration, we determined that the most suitable number of hidden layers is two.

Hidden Layer Numbers	Number of neurons	Training Accuracy	Test Accuracy	Training F1	Test F1	Training times	Inference times	Parameter counts
1	256	0.663	0.558	0.67	0.55	140	0.42	35594
2	256	0.744	0.561	0.79	0.55	205	0.85	101386
3	256	0.788	0.557	0.83	0.54	335	1.21	167178
4	256	0.790	0.549	0.83	0.53	427	1.52	232970

Table 9: Experiment Result for Number of Layers

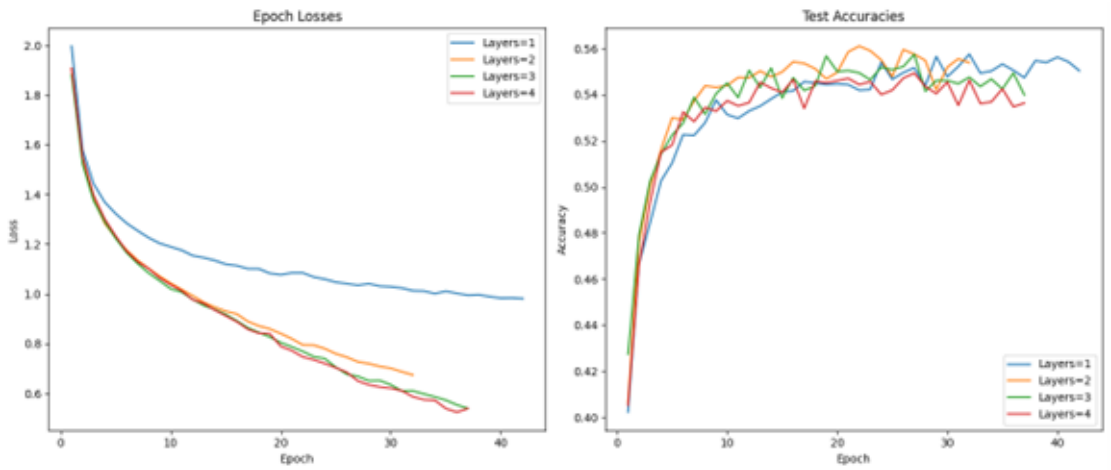


Figure 8: Experiment Result for Number of Layers

3.2.8 Epoch

From our research, we found that under the current parameters, all models were able to achieve good convergence within 80 training epochs. In most cases, test set accuracy did not continue to improve stably beyond 50 epochs, and signs of overfitting began to appear. Therefore, we established an epoch limit of 80 and, to prevent overfitting, we implemented an early stopping mechanism set at 10 rounds.

3.3 Ablation Study

An ablation study is a research method that evaluates the specific impact of certain components on model performance by progressively removing or disabling parts of the model. The results of the ablation study are displayed in Table 11.

3.3.1 Activation Function

In this experiment, we evaluated the effect of adding an activation function to the baseline model to determine its impact on model performance. Based on the experimental results, the activation function improves the neural network prediction accuracy significantly. Also the GELU activation function increases the inference time.

3.3.2 Batch Normalization Layer

In this experiment, we incorporated a Batch Normalization layer into the previously optimized parameters and model structure to assess if it enhances performance. This layer improves prediction results by a small amount, but slows down inference significantly.

3.3.3 Dropout

Following the optimal parameters and model structure determined previously, this experiment involved adding a Dropout layer to evaluate its effect on performance. From the experimental results dropout reduces overfitting and brings a slight improvement in accuracy.

3.3.4 Residual Block

Based on the optimal parameters and model structure previously identified, this experiment tested the integration of residual connections to examine their potential to improve performance.The residual layer further improves accuracy and reduces overfitting. Although the effect is not obvious numerically, it is actually very difficult to improve every 0.01 after the model accuracy exceeds 0.57.The results of the ablation experiments tell us that the addition of each layer is valuable in improving the overall performance of the model.

Add Layer	Training Accuracy	Test Accuracy	Training times	Inference times	Parameter counts
Base model	0.400	0.400	98	0.215	101386
+Activation function	0.671	0.559(+0.159)	148	0.841	101386
+Batchnorm	0.730	0.564(+0.005)	307	1.19	102410
+Dropout	0.68	0.578(+0.014)	491	1.10	102410
+Residual Block	0.678	0.585(+0.007)	904	1.49	168202

Table 10: Ablation Study Result

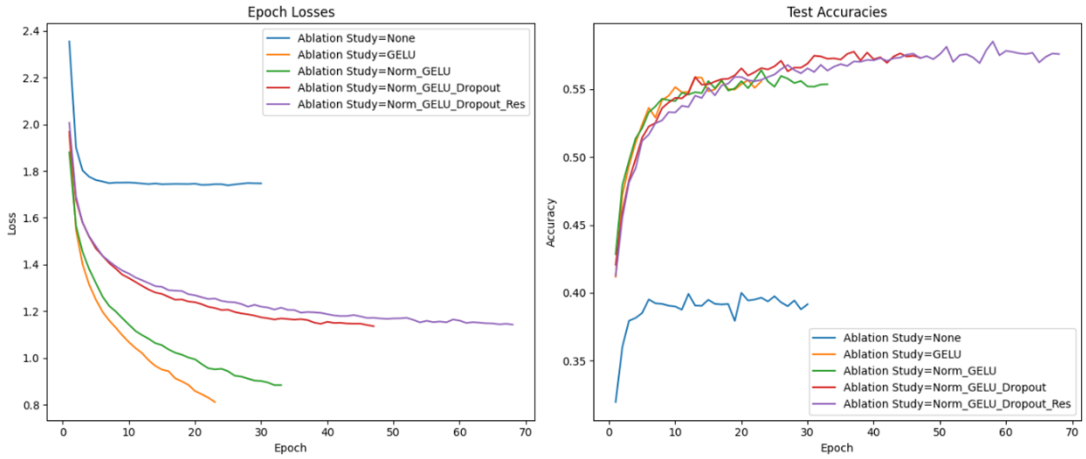


Figure 9: Ablation Study Result

3.4 Comparsion Experiments

3.4.1 Batch Normalization vs. Layer Normalization

In this experiment, we implemented Layer normalization instead of Batch normalization, based on the optimal parameters and model structure determined in the previous

step, to evaluate its impact on model performance. We can find that the accuracy of the model decreases after replacing the Batchnorm layer with the Layernorm layer. The results are presented in Table 11 and Figure 10.

Normalization	Training Accuracy	Test Accuracy	Training F1	Test F1	Training times	Inference times	Parameter counts
Batchnorm	0.676	0.581	0.68	0.58	1009	1.49	168202
Layernorm	0.553	0.506	0.55	0.50	927	2.16	168202

Table 11: Comparision of BN and LN

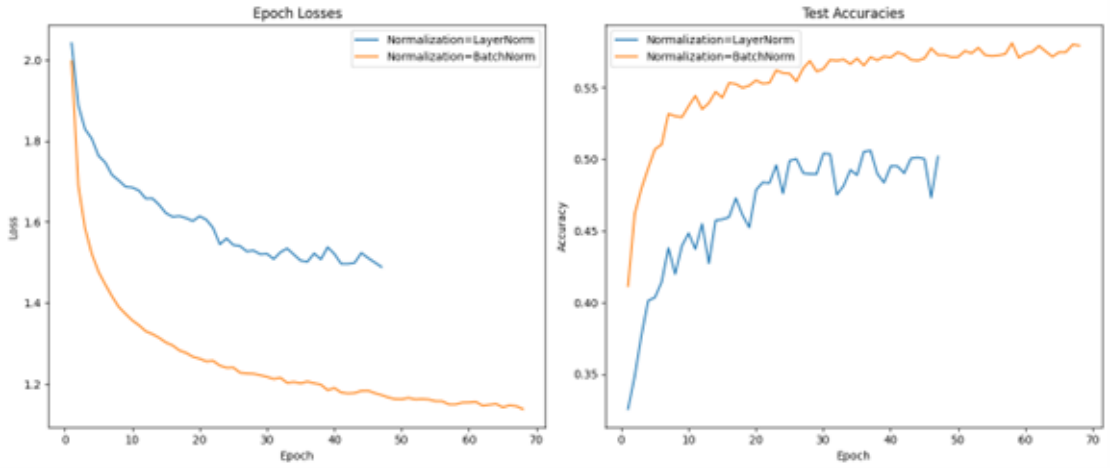


Figure 10: Comparision of BN and LN

3.4.2 Activation Function

In this experiment, we altered the activation function based on the optimal parameters and model structure determined in the previous step to assess whether it could enhance model performance.Comparative experiments reveal that GELU is the most suitable activation function, followed by ReLU and Leaky ReLU. The results are displayed in Table 12 and Figure 11.

Activation Function	Training Accuracy	Test Accuracy	Training F1	Test F1	Training times	Inference times	Parameter counts
ReLU	0.638	0.571	0.63	0.57	534	0.99	168202
Leaky ReLU	0.653	0.571	0.65	0.57	824	1.14	168202
GELU	0.672	0.581	0.67	0.57	768	1.49	168202
Tanh	0.648	0.566	0.67	0.56	521	1.13	168202
Sigmoid	0.579	0.547	0.59	0.54	922	1.19	168202

Table 12: Comparision of Different Activation Function

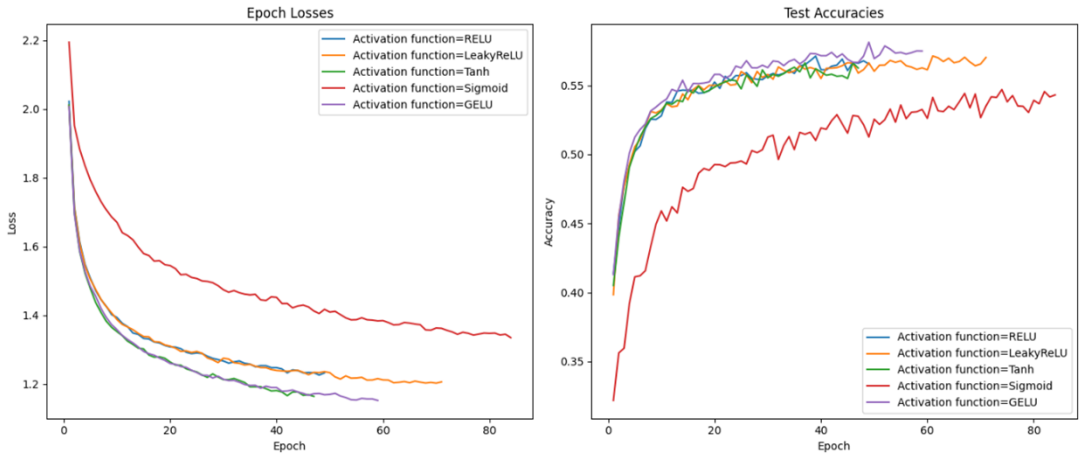


Figure 11: Comparision of Different Activation Function

3.4.3 Feature Extract or Embedding

This is an extended comparison experiment. We explored various methods of feature extraction—including linear layers, CNNs, RNNs, and Self Attention—based on the optimal parameters and model structure determined in the previous step, to determine if these could further enhance model performance. The results are displayed in Table 13 and Figure 12.

Embedding	Training Accuracy	Test Accuracy	Training F1	Test F1	Training times	Inference times	Parameter counts
Linear Layer	0.682	0.585	0.68	0.57	1060	1.49	168202
CNN	0.655	0.562	0.66	0.56	14823	67	135184
RNN	0.394	0.397	0.39	0.39	752	1.62	217610
Self Attention	0.195	0.195	0.12	0.12	635	30	7754

Table 13: Comparision of Different Feature Extraction Method

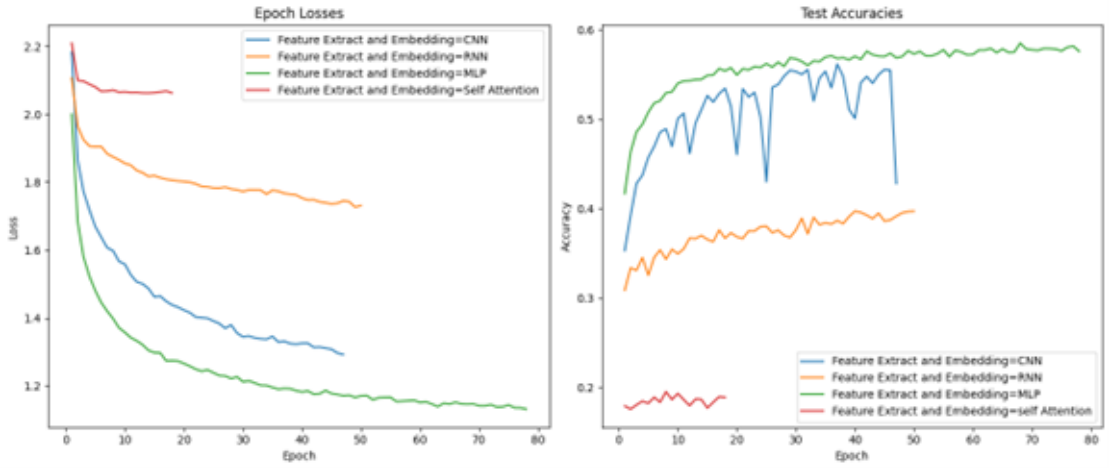


Figure 12: Comparision of Different Feature Extraction Method

3.5 Justification on Best Model

Drawing inspiration from the style of the ConvNeXt paper, we have streamlined the illustration of parameter adjustments and their impacts on model performance into Figure 13. In this figure, a green line indicates parameter adjustments, an orange line represents comparative experiments where the current module replaces the module indicated in parentheses, and a blue line shows the addition of modules as specified in parentheses. A bright blue line indicates that this step achieved the highest global accuracy. A bright yellow bar signifies a performance improvement at this step, leading to the adoption of the change, while a light yellow bar indicates no improvement, causing the model's parameters and structures to revert to those of the previous step (indicated by the last bright yellow bar).

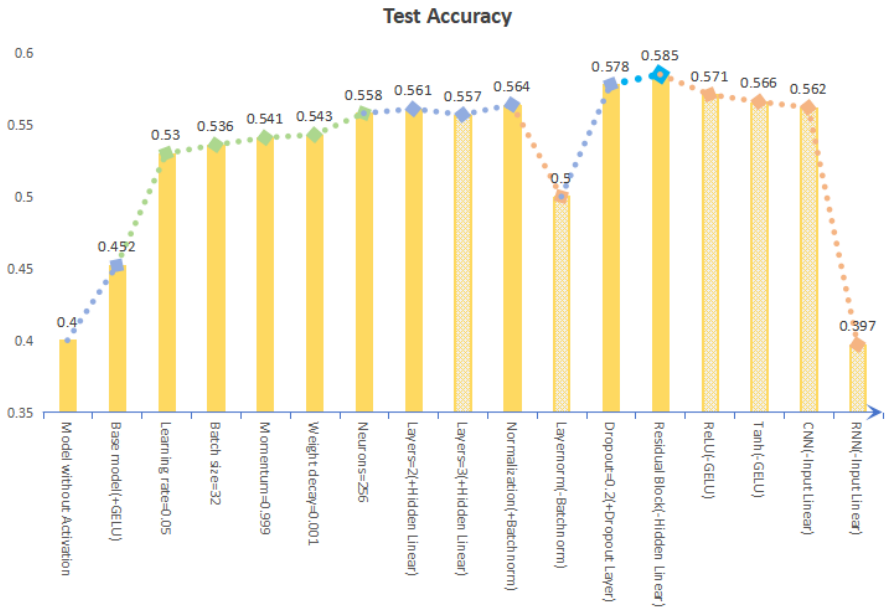


Figure 13: Process of All Adjustments

The final best model parameters and structures are presented in Tables 14 and 15.

Number of Layer	Class of Layer	Number of neurons
1	Linear Layer	(128,256)
2	Batchnorm	-
3	GELU	-
4	Dropout Layer	-
5	Residual Block	(256,256)
6	Batchnorm	-
7	GELU	-
8	Dropout Layer	-
9	Linear Layer	(256,10)
10	SoftMax	-

Table 14: Optimal Structure

Parameter	Value
Weight Decay	0.0001
Learning Rate	0.05
Momentum	0.999
Epoch	Early Stopping 10 steps
Batch size	32
Dropout	0.2

Table 15: Optimal Parameter

Eventually this model reached the optimum at the 87th epoch (with an accuracy of 0.584 for the test set), with a training time of 1269 seconds, an inference time of 1.49 seconds for the training set (50,000 samples), and a model parameter count of 168,202. Figure 14, Figure 15 and Figure 16 show the training and testing process and results.

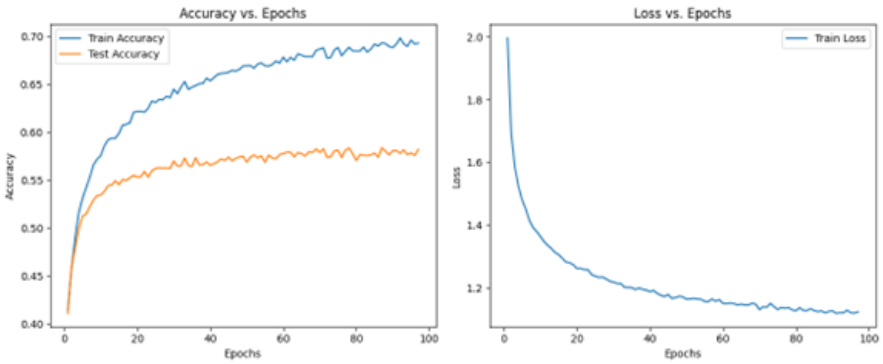


Figure 14: Final Training and Testing Process

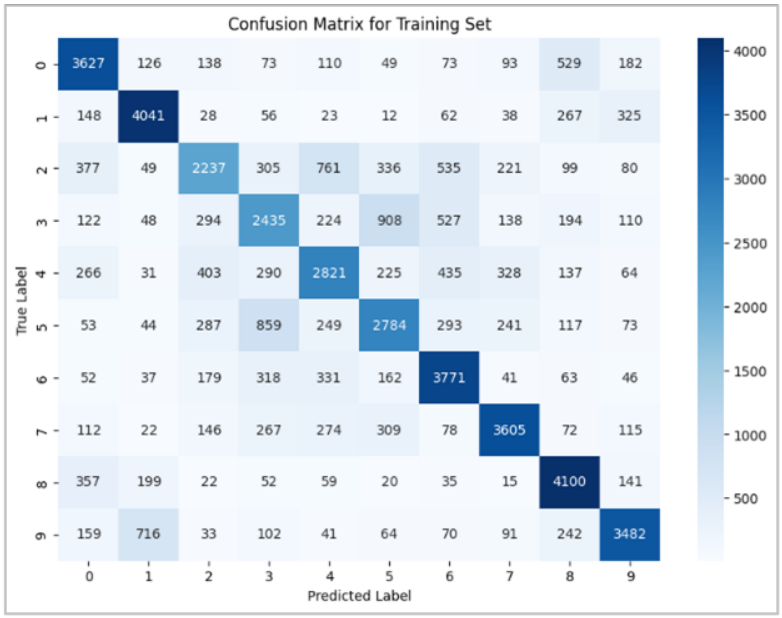


Figure 15: Confusion Matrix for Training Set

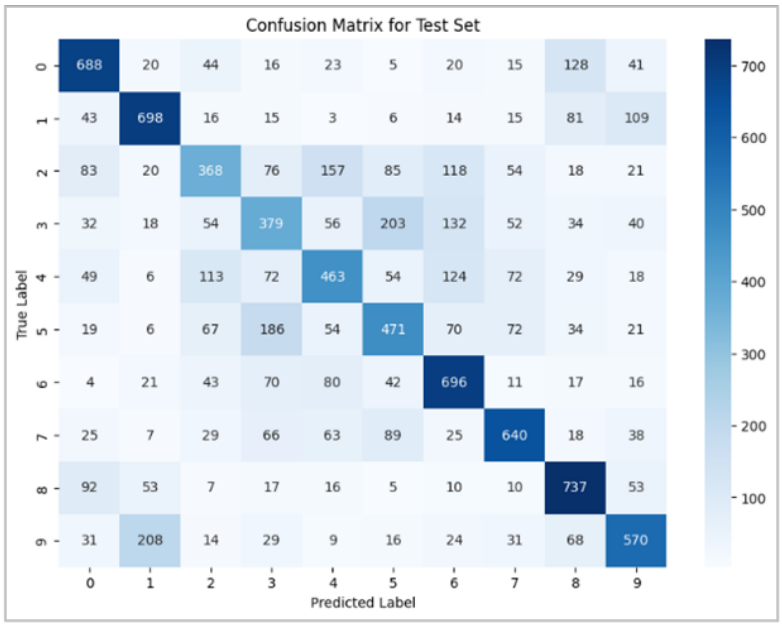


Figure 16: Confusion Matrix for Testing set

4 Conclusion and Discussion

4.1 Conclusion

Through this report, we have elucidated the theoretical basis for our team's code implementation. By conducting progressive parameter analysis, comparative experiments, and ablation studies, we gradually improved the model's performance, ultimately achieving an accuracy of 58.52%. We discovered that setting the momentum close to 1, increasing the learning rate, and decreasing the batch size significantly accelerates convergence and reduces training time without compromising model accuracy. Additionally, setting weight decay close to zero and using hidden layers with twice the number of neurons compared to the input parameters proved beneficial for improving accuracy. Further enhancements in accuracy were achieved by adding 1-2 hidden layers and employing residual connections within these layers. The sequential addition of Batch Normalization, GELU, and Dropout layers after the hidden layers contributed to the gradual improvement in model performance. Comparative experiments demonstrated that GELU is the most effective activation function among the five types we tested, and Batch Normalization proved more suitable than Layer Normalization for this particular task.

Beyond the requirements of the assignment, we explored the implementation of modules such as CNN1D, RNN, and Self-Attention, experimenting with various methods of data feature extraction. Although these modules did not ultimately enhance the model's performance—due to the need to balance time and space costs—they all demonstrated normal forward propagation, gradient descent, backward propagation, parameter updates, and overall model loss reduction within our model generation and training framework. This confirms that our code implementation offers a high degree of decoupling, compatibility, and readability.

4.2 Reflection

While we have fully met the task requirements in terms of model implementation and tuning, there remains significant room for improvement in the engineering of code implementation and hyperparameter search strategies.

Despite using the ConvNeXt structure and hyperparameter optimization process as references and adhering to a variable control principle for designing comparative experiments, the base model for subsequent experiments was updated using a greedy strategy with optimal parameters and structures from the current batch. However, the results of the greedy strategy often fail to represent the global optimum. Recent developments in neural architecture search (NAS) presented at top AI conferences like CVPR and NeurIPS—including approaches like ENAS, PNAS, DARTS, RLNAS, and those based on evolutionary algorithms—suggest promising alternatives. In our future work, we plan to integrate advanced automatic architecture search schemes into our training pipeline to outperform the greedy algorithms potentially.

In terms of engineering, to enhance code readability and usability, we adopted an implementation style closer to the PyTorch and Keras architectures. We implemented Dropout, Batch Normalization, and activation functions as separate layers, instead of treating them as parameters within hidden layers as typically done in Sklearn. However, improvements in code usability are still possible. By utilizing the forward function, we maintain the propagation of matrices (tensors) in the neural network as a dynamic computational graph, enabling automatic generation of the corresponding backward propagation process for each forward operation, thus freeing users from manually rewriting the backward process for new structures.

Moreover, the neural network architecture implemented with dynamically maintained computational graphs allows us to allocate different hardware units for various operators (or computational steps) through hardware-aware algorithms, considering the order of invocation of these operators in forward and backward propagation. This optimizes memory and GPU usage for training and inference deployment on the current training cluster. In our subsequent efforts, we aim to rewrite the code using the Rust language and implement dynamic computational graph maintenance to achieve greater computational and memory management efficiency. Mamba's implementation has already transitioned to using C++ to refactor the backward propagation process, achieving tangible performance improvements.

Furthermore, the implementation of Momentum for weight updates merits reflection. We discovered that Geoffrey Hinton's momentum-based gradient descent method, proposed in 2009, might not be optimal for this task. In practice, more effective techniques for calculating momentum exist. We explored alternatives, such as the method proposed by Polyak in 1964 and updates considering momentum weights, finding that these can enhance training efficiency beyond Hinton's classical approach.

Our ultimate goal is to develop a highly efficient neural network training pipeline architecture that could potentially rival PyTorch in the era of large models. This involves incorporating cutting-edge architecture search methods, hardware-aware training and inference deployment schemes, and modern gradient descent strategies.

References

[1] Xu, C. (2024). Lecture 2: Multilayer Neural Networks. *Lecture Slides, COMP5329: Deep Learning*, The University of Sydney.

[2] Xu, C (2024). Lecture 2: Optimization for Deep Models, *Lecture Slides, COMP5329: Deep Learning*, The University of Sydney.

[3] Xu, C (2024). Lecture 2:Regularization for Deep Models, *Lecture Slides, COMP5329: Deep Learning*, The University of Sydney.

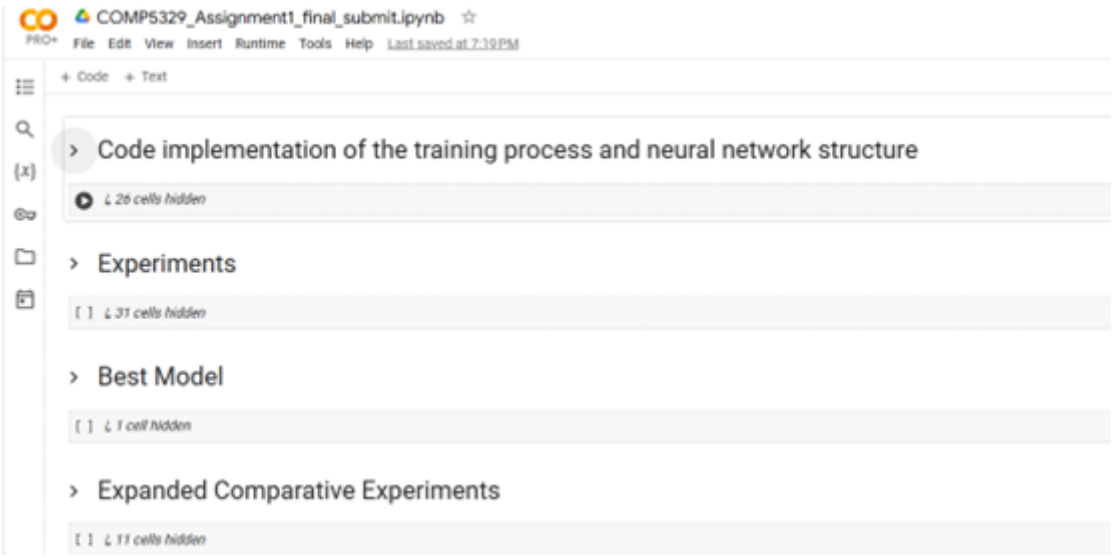
[4] Hendrycks, D., & Gimpel, K. (2016). Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415.*

[5] Cho, K. (2013). Understanding dropout: Training multi-layer perceptrons with auxiliary independent stochastic neurons. *Neural Information Processing*, 474–481. doi:10.1007/978-3-642-42054-2-59

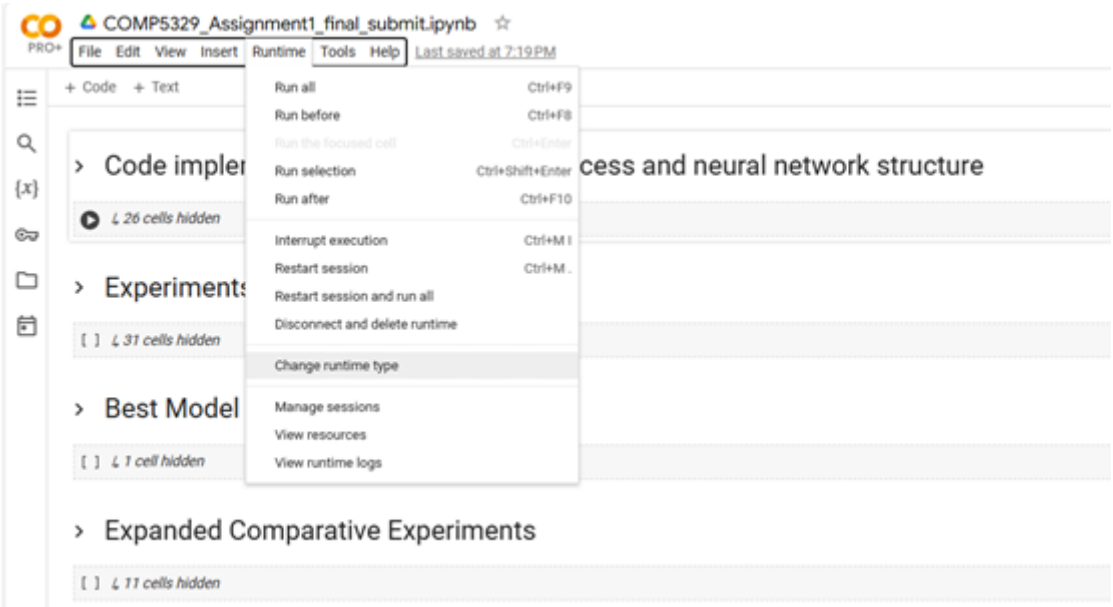
[6] Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450.*

Appendix

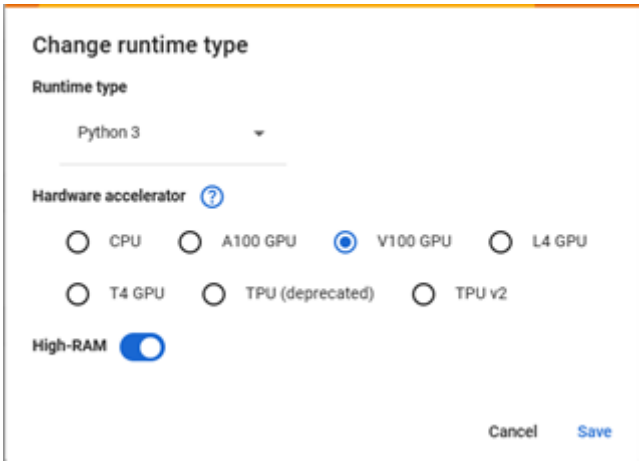
You can access our code at the following link:
https://colab.research.google.com/drive/1nA022qYFd6TcIIA3U-YzR7hnu8Km1qLC?usp=sharing
Once you are on the page you can see that the code is divided into four sections, if you only need to check the Best Model you need to run the Code implementation of the training process and neural network structure as well as the Best Model section. For this section you need to make sure that the runtime type is High RAM mode. To run the code click on the white triangle to the left of the hidden cell.



Modify the runtime as follows: Click to select Runtime and select Change runtime type in the drop-down menu.



Afterwards, just save the following settings in the pop-up window:



If you want to reproduce our experiment, you need to run the Code implementation of the training process and neural network structure, Experiments, and Best Model sections in turn. WARNING: Because our experiments are well-done, Experiments may take longer to run, so please check the results of the experiments before running the Experiments part of the code to avoid overwriting the results of the previous experiments! If you need to check our Expansion Experiments (this part is mainly used to test whether our framework is decoupled or not), you need to change the runtime to A100, high RAM type. After that you need to run the Code implementation of the training process and neural network structure as well as the Expanded Comparative Experiments section.