The University of Sydney

School of Computer Science

COMP5329 Assignment 1

Author:
Penghao Jiang 530835006
Mingyue Ma 530400789
Leke Tan 530709501

Supervisor:
Chang Xu

An Assignment report submitted for the COMP5329:

COMP5329

April 12, 2024

Contents

1 Abstract

This report investigates methods for enhancing performance in multi-classification problems. Subsequent sections will cover the problem introduction, analysis, dataset details, model overview, experimental outcomes, and a summary with reflections.
The link to the code and data is link.

2 Introduction

2.1 Aim of Study and Important of Study

In the realm of deep learning, multi-class classification stands as a pivotal task, characterized by its ability to categorize instances into more than two distinct classes. This task is fundamental across various applications, ranging from facial recognition systems and medical diagnosis to sentiment analysis and beyond. The essence of multi-class classification lies in its capability to process and analyze complex data, facilitating nuanced decision-making and prediction in various scientific and industrial domains.

Historically, the evolution of multi-class classification methodologies has been closely tied to advancements in deep learning, where neural networks have progressively been optimized to handle the intricacies of high-dimensional data spaces and the nuanced distinctions between multiple categories. Adopting deep learning in multi-class classification has enhanced the accuracy of predictive models and expanded the horizons of applicability, enabling solutions to previously intractable problems.

The significance of multi-class classification in the contemporary digital era cannot be overstated. In the age of data proliferation, where vast amounts of information are generated and captured, the ability to efficiently and accurately categorize data into relevant classes is paramount. However, the journey of refining multi-class classification models is fraught with challenges, including but not limited to the handling of imbalanced datasets, the curse of dimensionality, and the quest for models that can generalize well from training to unseen data while maintaining computational efficiency. These challenges underscore the ongoing need for research and innovation in this field, driving the quest for more sophisticated, robust, and efficient classification algorithms.

In summary, multi-class classification represents a core facet of deep learning, with deep ramifications across a multitude of sectors. The ongoing advancements in this area are not merely academic pursuits but are pivotal in shaping the technological landscape, driving efficiencies, and enhancing the systems' capabilities that permeate our daily lives and industrial processes. The motivation behind this project is rooted in this context, The research goal of this project is to build a multi-classification model by using and pairing different techniques.

In this paper, we introduce a multi-classification deep learning model designed to optimize classification outcomes by integrating various neural network techniques and optimization strategies. The fundamental components of the model encompass activation functions, weight adjustment mechanisms, optimizers, regularization techniques, and training strategies. Through an extensive examination of the synergy and efficacy of these components, this paper identifies an optimal model configuration for our multi-classification task. The experimental section details our systematic approach to testing and evaluating these combinations, aiming to ascertain the most suitable model structure for our data and specific task.

2.2 Introduction of Dataset

The dataset utilized in the experiment was bifurcated into two subsets: the training set and the test set. The training set comprised 50,000 samples, each endowed with 128 features and a unique label, utilized individually for the classification task. Each label has the same number of training examples which is 5000. The test set encompassed 10,000 samples, with each sample also featuring 128 attributes and an associated label.

3 Methodology

3.1 Data Pre-processing

In this study, data normalization methods are initially employed in the data pre-processing to scale the data within the 0 to 1 range. Given that each sample possesses 128 features, normalizing these features to a uniform scale enhances model convergence speed and mitigates the influence of outliers on model performance. Subsequently, standardization is applied, adjusting data features to have a mean of zero and a standard deviation of one. It normalizes the feature distribution, facilitating comparison across different features and further reducing outlier effects.
The label vector is transformed into a one-hot encoded format, which is essential for representing the model's multiple, mutually exclusive output categories because this study's task is multiclass classification. This encoding method not only eliminates numerical significance and enhances model interpretability but also aligns with the cross-entropy loss function utilized by the model. It enables efficient computation of discrepancies between predicted and actual values, thus optimizing model performance and interpretability.

3.2 Principle of Different Technology Modules

3.2.1 Activation Function

This section will concentrate on the activation functions evaluated in the experiments, highlighting their specific usage scenarios and their advantages and disadvantages. Subsequently, I will employ the mathematical formulas for each activation function to analyze and illustrate their characteristics.
Without an activation function, each layer's output becomes a linear function of the previous

layer's inputs, resulting in a linear combination of these inputs regardless of the neural network's layer count, which resembles the most primitive perceptron. In such cases, each layer acts merely as a matrix multiplication, with additional layers contributing to nothing beyond extended matrix multiplication. Conversely, incorporating an activation function introduces nonlinearity to the neurons, empowering the neural network to approximate any nonlinear function. This capability enables the network to adapt to various nonlinear models. Next, I will describe in detail the activation functions used in the experiments.

Sigmoid

The sigmoid function, also known as the Logistic function, produces outputs for hidden layer neurons in the range (0,1), making it suitable for binary classification tasks. (Rumelhart, Hinton, & Williams, 1986) The Sigmoid function is an activation function with an S-shaped curve, formally defined as:

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

where the output is constrained between 0 and 1, making it particularly suitable for models that predict probabilities. The derivative of the Sigmoid function is:

$$\sigma'(x) = \sigma(x)(1-\sigma(x)) \tag{2}$$

Advantages:

Limited Output Range: The sigmoid function's output lies between (0,1), offering a stable, limited range, ideal for use in the output layer.

Continuity and Differentiability: Being a continuous function, it is easily differentiable, facilitating gradient-based optimization methods.

Disadvantages:

Saturation Leading to Vanishing Gradients: The sigmoid function tends to saturate at extreme values of the input, leading to almost flat slopes. This causes the gradients to be near zero during backpropagation, resulting in minimal weight updates and often leading to the vanishing gradient problem, which hampers the training of deep networks.

Non-Zero Mean Output: Since the sigmoid function's output is not zero-centered, it produces non-zero mean signals for subsequent layers. This can negatively impact the gradient flow through the network.

High Computational Complexity: Due to its exponential form, the sigmoid function is computationally intensive, which can be a drawback in resource-constrained environments or with large-scale data.

While Sigmoid functions are useful for their interpretability, they can suffer from vanishing gradients and are computationally intensive due to the exponential component.

Tanh

The hyperbolic tangent function (tanh) is an S-shaped function similar to the sigmoid but with an output range from -1 to 1(Rumelhart et al., 1986), which is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3}$$

The derivative of the tanh function is:

$$\tanh'(x) = 1 - \tanh^2(x) \tag{4}$$

Advantages:
The tanh function is zero-centered, which can help with the convergence of gradient-based optimization algorithms. And it is faster than a sigmoid.
Disadvantages:
It still shares the vanishing gradient problem with the sigmoid function for inputs with large magnitudes.

ReLU
The Rectified Linear Unit (ReLU) is a piecewise linear function that outputs the input if it is positive, and zero otherwise(Krizhevsky, Sutskever, & Hinton, 2012). It is mathematically represented by:

$$\text{ReLU}(x) = \max(0, x) \tag{5}$$

The ReLU function's derivative is:

$$\text{ReLU}'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{6}$$

Advantages:
Faster Convergence: The SGD algorithm with ReLU converges more rapidly compared to using sigmoid or tanh, enhancing training efficiency.
No Gradient Saturation or Vanishing: In the region where x > 0, ReLU does not suffer from gradient saturation or vanishing issues, maintaining robust gradient flow.
Low Computational Complexity: ReLU has lower computational complexity as it does not require exponential operations, relying instead on a simple threshold operation to determine the activation value.
Disadvantages:
Non-Zero Mean Output: ReLU's outputs are not zero-centred.
Dead ReLU Problem: ReLUs can become inactive in negative regions, a condition known as "dead ReLU." They are vulnerable during training, and once a ReLU neuron outputs zero (for x < 0), its gradient also becomes zero. This causes both the neurons and subsequent neurons in the network to stop responding to input variations, preventing further updates to their parameters.
Despite its advantages in promoting faster learning and sparsity, the ReLU function can suffer from the dead neuron problem, where neurons can become inactive if they consistently receive negative inputs.

Leaky ReLU
The Leaky Rectified Linear Unit (Leaky ReLU) aims to mitigate the dead neuron problem by allowing a small, non-zero gradient when the neuron's input is negative(Maas, Hannun, & Ng, 2013). It is described as:

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{7}$$

The derivative of the Leaky ReLU function is:

$$\text{Leaky ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases} \tag{8}$$

Advantages:
Mitigation of Dead ReLU Problem: The Leaky ReLU function addresses the Dead ReLU Problem by assigning a small slope to negative inputs, effectively countering the zero-gradient issue and revitalizing the learning process in areas where ReLU fails.
Extended Output Range: Leaky ReLU extends the output range from negative infinity to positive infinity, enhancing the flexibility of the ReLU function. The coefficient α is typically set to a small value, such as 0.01, to maintain the function's effectiveness without overemphasizing negative inputs.
Disadvantages:
Inconsistency in Performance: Although theoretically superior to the ReLU function, Leaky ReLU has shown unstable results in practice, leading to its limited adoption in real-world applications.
Inconsistent Relationship Between Positive and Negative Inputs: The differential treatment of positive and negative inputs by Leaky ReLU can lead to inconsistent relationships in the data representation, complicating the model's ability to generalize across different input ranges. This small slope for negative inputs ensures that neurons remain active, which can be beneficial during the training process of deep networks.

Softmax
The Softmax function is an essential component in multi-class classification problems, particularly when the task involves assigning class membership among more than two classes. For any given real vector z of length K, the Softmax function maps it onto a new vector of the same length, with elements constrained to the range (0, 1), such that the components of the output vector sum to one](Bridle, 1990). This transformation is defined as:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \dots, K \tag{9}$$

where z_i represents the i-th element of the input vector z, and K is the total number of classes. A key characteristic of the Softmax function is that it incorporates all the individual elements of the input vector into the normalization process, resulting in a mutually exclusive probability distribution across the classes.
Advantages:
Preservation of Data Integrity: Unlike the traditional max function that outputs only the maximum value, Softmax ensures that smaller values retain smaller probabilities and are not discarded outright, thereby preserving the integrity of the data to the greatest extent.
Interrelated Probabilities: The denominator of the Softmax function incorporates all the original output values, meaning the probabilities obtained by Softmax are interrelated, making it better suited for multi-class classification tasks.
Disadvantages:
Non-Differentiability at Zero and Zero Gradient for Negative Inputs: The Softmax function is non-differentiable at zero, and the gradient for negative inputs is zero. This means that weights will not update during backpropagation for activations in this region, leading to permanently inactive "dead neurons."
GELU(Advanced Module)
In constructing neural networks, nonlinearity is a critical feature of the model, essential for enhancing its generalization capabilities. To achieve this, stochastic regularization, such as

dropout (to be detailed in a forthcoming report), is incorporated. It is crucial to understand that stochastic regularization and nonlinear activation are distinct concepts, yet both these elements in tandem govern the model's input(Hendrycks & Gimpel, 2020).
The Gaussian Error Linear Unit (GELU) embodies the concept of stochastic regularization within its activation mechanism, providing a probabilistic interpretation of neuron inputs. This approach aligns more naturally with our intuitive understanding and has shown superior experimental results compared to both ReLU and ELU functions.
GELU effectively integrates the principles of dropout, zoneout, and ReLU by multiplying the input with a mask composed of 0s and 1s. The creation of this mask is random, contingent on the input, and adheres to a Bernoulli distribution $(\Phi(x), \Phi(x) = P(X \leq x)$ X is assumed to follow a standard Gaussian distribution. Then $\phi(x)$represents the cumulative distribution function (CDF) of the standard normal distribution. This is particularly relevant as neuron inputs often exhibit a normal distribution, a tendency that becomes more pronounced in deep networks where Batch Normalization is extensively applied." This selection is predicated on the tendency of neuron inputs to conform to this distribution, allowing for an increased probability of input dropout as x diminishes. Consequently, the activation transformation in GELUs is stochastically dependent on the input, fostering a dynamic and responsive model behavior.

$$GELU(x) = x\Phi(x) \tag{10}$$

$$\Phi(x) = \int_{-\infty}^{x} \frac{e^{-t^2/2}}{\sqrt{2\pi}} dt = \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right] \tag{11}$$

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2} dt \tag{12}$$

Equations 10, 11, and 12 present the formulas for the Gaussian Error Linear Unit (GELU) activation function. In these formulas, $e^{-\frac{t^2}{2}}$ denotes the probability density function of the standard normal distribution.$\sqrt{2\pi}$ represents the normalisation constant in the probability density function of the standard normal distribution that ensures that the total probability is 1. x in the erf function denotes the parameters of the error function, representing the integral over the range from 0 to x in the standard normal distribution and e^{-t^2} denotes the product function in the integral expression, and denotes the core part of the Gaussian function.
Advantages:
Adaptive Gating Mechanism: GELU incorporates a gating mechanism that allows the model to regulate information flow adaptively. This feature enables automatic adjustment of neuron activation levels, facilitating the capture of intricate patterns in the input data.
Non-saturation: GELU is a non-saturating activation function, addressing the vanishing gradient issue prevalent in saturated functions like Sigmoid and Tanh, thereby enhancing training efficacy in deep networks.
Improved Model Performance: Empirical evidence indicates that GELU can bolster model performance across various tasks, particularly in natural language processing, leading to superior training and testing outcomes.
Disadvantages:
Computational Complexity: GELU exhibits higher computational complexity relative to ReLU and its variants, due to its exponential operations, potentially slowing down performance in certain implementations.

3.2.2 Optimisation algorithms and strategies

Momentum in Stochastic Gradient Descent

The standard stochastic gradient descent (SGD) algorithm updates the network parameters, including weights and biases, to minimize the loss function(Polyak, 1964). It accomplishes this by taking incremental steps in each iteration towards the negative gradient of the loss function. The updated Formula for SGD is as follows:

$$\theta_{l+1} = \theta_l - \alpha\delta\mathbb{E}(\theta_l) \tag{13}$$

In the context of gradient descent algorithms, l represents the number of iterations, $\alpha > 0$ denotes the learning rate, θsignifies the parameter vector, and$\mathbb{E}(\theta)$is the loss function. It is also the gradient of the loss function, computed using the entire training set in the standard gradient descent method and utilizing the full dataset for each update. Conversely, stochastic gradient descent (SGD) evaluates the gradient with only a subset of the training data in each iteration, leading to parameter updates. This process involves using different subsets, referred to as mini-batches, in each iteration.

The term "epoch" in training algorithms denotes a complete pass through the entire training set using mini-batches. SGD is characterized as stochastic because the parameter updates, calculated with these mini-batches, are approximate and noisy compared to the updates derived from the full dataset. This stochastic nature results from the smaller, varying subsets of data used in each iteration.

The derivatives in all dimensions approach zero near the local optimum point, causing the gradient descent method, which updates model parameters based on the product of the derivative and the step size, to struggle in escaping local optima. Consequently, traditional stochastic gradient descent (SGD) is prone to converging to local optima and getting bad performance at saddle points and bad/sharp minima.

The algorithm of stochastic gradient descent with momentum (SGD with momentum) enhances the standard gradient descent optimization process. This method, often referred to as gradient descent with momentum, is designed to accelerate optimization, for instance, by reducing the function evaluations needed to achieve the optimum or by amplifying the optimization algorithm's efficacy.

With momentum, if the gradient maintains the same sign as in the previous iteration, the descent accelerates (increasing in magnitude), addressing the issue of excessively slow descent. Conversely, if the gradient sign differs from the previous iteration, the momentum causes mutual dampening, reducing oscillations. The momentum aspect allows the model to overcome local optima, making it less likely to get trapped in these points. The Formula of SGD with momentum is as follows:

$$\theta_{l+1} = \theta_l - \alpha\delta\mathbb{E}(\theta_l) + \gamma(\theta_l - \theta_{l-1}) \tag{14}$$

The momentum factor, represented by γ, reflects the influence of the previous gradient step on the current iteration. Stochastic Gradient Descent with momentum (SGD with momentum) is particularly effective in flat or nearly flat gradient search spaces, such as areas with zero gradient. Momentum facilitates the continuation of the search past flat regions, aiding in overcoming areas of gradient flatness.

In SGD with momentum, parameter updates are influenced by the current gradient and the direction of the previously accumulated gradient descent. When the current gradient direction aligns with the historical gradient direction, it reinforces the current gradient, leading to a larger step size in the descent. Conversely, if the current gradient direction opposes the accumulated gradient direction, the descent's magnitude is reduced. This mechanism helps accelerate convergence and mitigate oscillations.

Batch Normalization
In neural networks, parameter updates result in the input distributions for each layer changing, which causes the phenomenon known as internal covariate shift. This variability complicates network convergence and necessitates careful control of the learning rate, precise parameter initialization, and a restrained number of network layers. However, these constraints may not always align with the requirements of specific tasks. Investigation revealed that the root of this issue lies in the excessive and unpredictable distributional variance across layers(Ioffe & Szegedy, 2015).
Batch normalization is employed in this project to standardize the distribution of inputs across batches, effectively mitigating the issue of significant distributional discrepancies between layers to address this challenge. Batch normalization not only normalizes the data in a less resource-intensive manner but also reintroduces flexibility through learnable parameters α and β, which compensate for any potential reduction in data representational capacity induced by normalization. Batch normalizations help maintain the stability of data distribution across layers, thereby accelerating the learning process and preventing the network from getting stuck in zones of gradient saturation, which in turn facilitates faster network convergence. The Formula of batch normalization is formulated as follows:

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \tag{15}$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2 \tag{16}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_b^2 + \varepsilon}} \tag{17}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \tag{18}$$

Formula 15 delineates the mathematical procedure for calculating the mean of a batch, while Formula 16 specifies the method for determining the batch's variance. Formula 17 illustrates the normalization process for a batch, and Formula 18 outlines the procedure for adjusting the scale and shift of a batch. This sequence of equations represents the systematic steps involved in the batch normalization process, each serving a distinct function in transforming the batch data to facilitate more stable and efficient neural network training.

Mini-batch training
In deep learning, mini-batch training is a prevalent method characterized by partitioning the training dataset into smaller sets known as mini-batches, which are then utilized for model training. A single mini-batch updates the model's parameters in each iteration. This technique is advantageous as it lessens memory usage, avoiding the need to simultaneously load the entire dataset into memory(Robbins & Monro, 1951). Additionally, it facilitates faster training due

to more frequent updates of the model parameters. Compared to traditional Batch Gradient Descent and Stochastic Gradient Descent, mini-batch training typically achieves faster convergence and exhibits greater resilience to variations in data distribution. With batch gradient descent, the entire training set is processed in each iteration, typically resulting in a consistent decrease in the cost function; any increase in the cost function across iterations may indicate issues in the training process requiring prompt resolution.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{19}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2 \tag{20}$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{21}$$

$$y_i = \gamma \hat{x}_i + \beta \tag{22}$$

The formulas for mini-batch normalization are delineated from Equations 19 to 22. In these formulas, μ_B represents the mean of the mini-batch B. x_i denotes the ith data point in the mini-batch, and m is the mini-batch size. The variance of the mini-batch is denoted by σ_B. $\hat{x}_i$ refers to the normalized data point. The term ε in the denominator is a small constant, preventing zero division, thereby avoiding a vanishing gradient. The final output is y_i adjusted by two learnable parameters, γ and β, which facilitate the model's learning of the optimal data distribution. Unlike batch gradient descent, mini-batch training does not guarantee a reduction in loss after every iteration, particularly when training involves varying mini-batches. Consequently, fluctuations in the loss are observed, albeit with a general downward trend. The effectiveness of mini-batch training heavily depends on the batch size or the number of samples per mini-batch, which should be selected considering the computational resources and data characteristics. While smaller batches can expedite training iteration, larger batches provide more accurate gradient estimates.

3.2.3 Regularisation method

Dropout

In deep learning network construction, when a model possesses excessive parameters relative to the number of training samples, it becomes susceptible to overfitting (Krizhevsky et al., 2012). Overfitting is characterized by low loss on training data with high prediction accuracy, contrasted with higher loss and lower accuracy on test data. This issue is prevalent in machine learning and deep learning, leading to poor model performance.

Ensemble methods, which involve training and combining multiple models, are commonly employed to address overfitting. However, this approach is resource-intensive, requiring significant time for both training and testing multiple models. An effective alternative to mitigate over-fitting is using dropout, which is a form of regularization method. Dropout helps prevent overfitting in complex feed-forward neural networks, especially when trained on limited

datasets, by reducing the interdependence of feature detectors. It operates by randomly deactivating a portion of the neurons (setting their output to zero) during each training batch, thus enhancing the network's generalization capability. The principle of dropout involves randomly omitting a neuron's activation during forward propagation with a certain probability p, reducing the model's reliance on specific local features and promoting a more generalizable model. The Formula for dropout is formulated as follows:
We assume the standard networks are Formula 27 and Formula 28. The Formula using

$$z^{l+1} = w^{l+1}y^{l} + b^{l+l} \tag{23}$$

$$y^{l+1} = f(z^{l+1}) \tag{24}$$

$$r^{l} = Bernoulli(p) \tag{25}$$

$$\hat{y}^{l} = r^{l}y^{l} \tag{26}$$

$$z^{l+1} = w^{l+1}\hat{y}^{l} + b^{l+l} \tag{27}$$

$$y^{l+1} = f(z^{l+1}) \tag{28}$$

Formulas 25 and 26 introduce the mathematical representation of the dropout mechanism. In this mechanism, each neuron has a probability p of being set to zero, following a Bernoulli distribution. For example, in a Dense/Linear Layer preceding a Dropout Layer with 1000 neurons, setting the dropout probability p to 0.5 means each neuron has a 50% chance of being deactivated, effectively randomizing the network's learning process to mitigate over-fitting. Additionally, adhering to the principles of neural networks, it is crucial to maintain the expected value of each neuron's output consistent between the training and testing phases. This ensures that the network's behaviour remains stable across different stages of model use, preventing discrepancies that could affect performance and reliability.
Dropout addresses over-fitting in deep learning networks by functioning like an averaging process. Training the same dataset across n distinct neural networks without dropout typically yields n varied outcomes. To determine the final result, one might employ averaging these outcomes or a majority vote strategy, both effective against over-fitting. This diversity in network results can lead to different over-fitting patterns, averaging and potentially neutralizing conflicting fits.
Dropout randomly eliminates a portion of hidden neurons, simulating the effect of averaging across multiple neural networks, where each network's distinct over-fitting may cancel out others, thus mitigating the overall over-fitting effect. Additionally, dropout ensures that two neurons are not consistently co-present in all networks, disrupting the dependence on fixed neuron relationships for weight updates. This disruption prevents the network from relying solely on specific feature combinations, compelling the learning of more generalizable features that remain effective across various neuron subsets, further aiding in over-fitting reduction.

Weight decay
This report delves into using L2 regularization as a method of weight decay, where the primary objective is to moderate the model's weights by introducing a regularization term to the loss function. This term typically involves the squared magnitude of the weights, augmented by a positive hyperparameter that modulates the regularization's impact on the loss function. An increase in the hyperparameter value intensifies the regularization effect, thereby simplifying the model. The weight decay equation is represented as Formula 29, with $L_{data}(\theta)$ denoting the

original loss function, $L_{reg}(\theta)$the regularization term, and $\lambda$the regularization hyperparameter. This report proceeds to detail the mathematical derivation of weight decay, initially presenting the basic loss function for a simple linear regression model as Formula 30, where $\hat{y}$ signifies the predicted value, y is the actual value, and m is the count of training samples. The regularization term for a straightforward classification model is articulated as Formula 31, with parameter w representing the weight count and $\lambda$the regularization hyperparameter(Hoerl & Kennard, 2000).
Combining the basic loss function and the regularization term yields the comprehensive weight decay loss function, encapsulated in Formula 28. The loss function's gradient is computed and applied to update the parameters by utilizing a gradient descent algorithm, following the gradient calculation rule outlined in Formula 31, with the parameter update Formula depicted in Formula 32, where α is the learning rate. This mathematical exploration underscores that weight decay primarily curtails model complexity and bolsters generalizability by incorporating regularization terms. Contrary to dropout and L1 regularization, because L1 induces weight sparsity and dropout introduces model stochasticity by random neuron deactivation, and L2 regularization systematically moderates weight magnitudes, thereby streamlining the model's complexity.

$$L(\theta) = L_{data}(\theta) + \lambda L_{reg}(\theta) \tag{29}$$

$$L_{data}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2 \tag{30}$$

$$L_{reg}(\theta) = \frac{\lambda}{2m}\sum_{i=1}^{w}\theta_i^2 \tag{31}$$

$$\frac{\partial L(\theta)}{\partial \theta_i} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)x_i + \lambda\theta_i \tag{32}$$

$$\theta_i = \theta_i - \alpha\frac{\partial L(\theta)}{\theta_i} \tag{33}$$

3.2.4 Loss Function

Cross Entropy Loss
The first thing that needs to be explained is the definition of entropy. The model aims to find a way to encode information efficiently/losslessly: efficiency is measured by the average length of the encoded data; the smaller the average length, the more efficient it is; and the condition of "losslessness" is also met, i.e. no loss of original information after encoding. Therefore, the definition of entropy is the minimum average coding length for lossless coding of event information. For the information with N possibility states, the possibility of no state is P = 1/N. So we can get the average minimum coding length needed to encode the information as $Entropy = -\sum_i P(I) log_2 P(I)$ where P(i) is the possibility of the ith information state.
Based on the above Formula for entropy, we can get the probability distribution P(x) for a continuous variable x(Shannon, 1948).

The entropy Formula is $Entropy = -\int p(x)log_2P(x)dx$. For both discrete and continuous variables, the expectation of the logarithm of the probability of a negative is computed in mathematics which represents the theoretical average minimum coding length of the event, so the entropy Formula can also be expressed as follows, where $x \sim P$ in the Formula means that we use the probability distribution P to compute the expectation. The Formula for calculating entropy can be expressed as $Entropy = E_{x\sim p}[-logP(x)]$.
However, in the vast majority of cases, the true probability distribution is not available. Only the estimated probability distribution Q is available, and the estimated probability distribution introduces a two-part uncertainty for the sake of our Formula. The first point is that the probability distribution for calculating the expectation is Q, which is different from the true probability distribution P. The second point is that the probability of calculating the minimum code length is $-logQ$, which differs from the true minimum code length $-logP$. Because the estimated probability distribution Q affects both of these components (expectation and encoding length), the result obtained by the model is likely to be very wrong.
Consequently, the comparison of that result with the true entropy is meaningless. The entropy formula gives the aim of having the encoding length as short as possible. So now we need to compare the coding length and (entropy). Assuming that the true probability distribution P is obtained from subsequent observations, the average code length can be calculated using P. In contrast, the actual code length is calculated based on Q. This calculation results in the cross-entropy of P and Q. In this way, the actual coding length and the theoretical minimum coding length are compared. The formula of cross entropy loss is formulated as follows:

$$CrossEntropy(P,Q) = \sum_{i=1}^{n} P(x_i)logQ(x_i) \tag{34}$$

3.3 Kaiming Initialization Method(Advanced Module)

This paper employs the ReLU activation function, and correspondingly, the Kaiming initialization method is chosen due to its consideration of ReLU's nonlinear characteristics on the negative half-axis. Kaiming initialization aims to maintain consistent variance across each layer's inputs in deep neural networks, thus mitigating gradient vanishing or explosion. The method is mathematically formulated in Equation 35, where $\mathcal{N}(0, \sqrt{\frac{2}{\eta_l}})$ signifies a normal distribution with a mean of 0 and a variance of $\sqrt{\frac{2}{\eta_l}}$, and η_l represents the number of units in a layer's input, which corresponds to the rows in the weight matrix or the neuron count in the preceding layer(He, Zhang, Ren, & Sun, 2015).

$$W \sim \mathcal{N}(0, \sqrt{\frac{2}{\eta_l}}) \tag{35}$$

By preserving the variance consistency of activation functions across layers, Kaiming initialization helps prevent gradient issues and facilitates faster model convergence.

3.4 Assumption of Best model

Based on the reasoning and analysis of the mathematical process above, we can infer that the composition of the best-performing multiclassification model should be used:
Normalization and one-hot encoding are utilized for data preprocessing. Kaiming initialization is applied for weight initialization, while L2 regularization is the weight decay strategy. To optimize the model, batch normalization, dropout, and Stochastic Gradient Descent (SGD) with momentum are employed. The ReLU and GELU activation functions are selected for different layers, with the softmax activation function designated for the output layer. Finally, the cross-entropy loss function is adopted as the model's loss function.
Data normalization, one-hot encoding, and L2 regularization enhance the model's generalizability and mitigate overfitting. Kaiming initialization and SGD with momentum expedite model convergence and dampen oscillations. Batch normalization and dropout refine the training process, diminish internal covariate shift, and bolster training stability. Collectively, these elements contribute to a high-performing model design.

4 Experiment and Analysis

4.1 Experimental hardware requirements

This experiment was conducted entirely on Colab, utilizing Python 3 on Google Compute Engine. The system was configured with a maximum RAM of 12.7GB, GPU RAM of 15.0GB, and disk space of 78.2 GB. The experiment was completed without the need for external resources or an upgrade to Colab Pro.

4.2 Assessment Criteria

This paper evaluates the model's performance across three key metrics: **Accuracy**, **F1 score**, and **time efficiency**. The subsequent section will elaborate on the methodologies and applications of these metrics. Accuracy is assessed by applying the trained model to test data, with its computation detailed in Equation 36.

$$accuracy = \frac{the \quad number \quad of \quad true \quad predict}{the \quad number \quad of \quad all \quad test \quad samples} \tag{36}$$

The F1 score is an amalgam of precision and recall, essential for assessing model performance. Precision is the ratio of accurately predicted positive samples to all samples classified as positive by the model. High precision indicates fewer false positives. Recall measures the fraction of actual positive samples correctly identified by the model, with high recall signifying fewer false negatives. The range of the F1 score is from 0 to 1, which is calculated as shown in Equation 37, where 1 represents perfect precision and recall, and 0 is the poorest. It is particularly crucial in scenarios of class imbalance, as it encapsulates precision and recall, offering a holistic view of the model's performance(United States. National Archives And Records Service.

Office Of Records Management, 1972).

$$F1score = 2 \times \frac{precision \times recall}{precision + recall} \tag{37}$$

$$Precision = \frac{TP}{Tp + FP} \tag{38}$$

$$Recall = \frac{TP}{TP + FN} \tag{39}$$

Formula 37 details the calculation of precision and recall, as outlined in Formula 38 and Formula 39. TP stands for True Positive in these formulas, referring to instances the model correctly identifies as positive. FP stands for False Positive, indicating instances the model incorrectly predicts as positive. TN denotes True Negatives, which are instances correctly identified as negative. FN, meaning False Negatives, refers to instances incorrectly labeled as negative. The duration of the model's operation gauges time efficiency, reflecting its capacity to expedite the training and testing phases.

4.3 Hyperparameter Optimization

4.3.1 Base Model

Hyperparameter Analysis

Table 1: Hyperparameter Of Base Model

Layer size	[128,128,10]
Activation	[None, 'relu', 'softmax']
Learning Rate	0.001
epoch	30
lr _decay	0.99
momentum	0.9
batch_size	256
drop out rate	0
batch norm	False
weight decay	0

Our base model comprises three layers: an input, a hidden, and an output layer. A three-layer perceptron is the simplest form of a multilayer perceptron, which is the reason for its selection as our base model. We will not employ dropout or batch normalization techniques during the initial capability testing, setting these hyperparameters to 0. Other parameters will adhere to the default hyperparameter values commonly found in the library, serving as the parameters for this base model.

Result Analysis

According to Figure 1, our model's accuracy stabilizes around 30 epochs without significant changes. Thus, we will set 30 as our epoch parameter in subsequent training sessions.

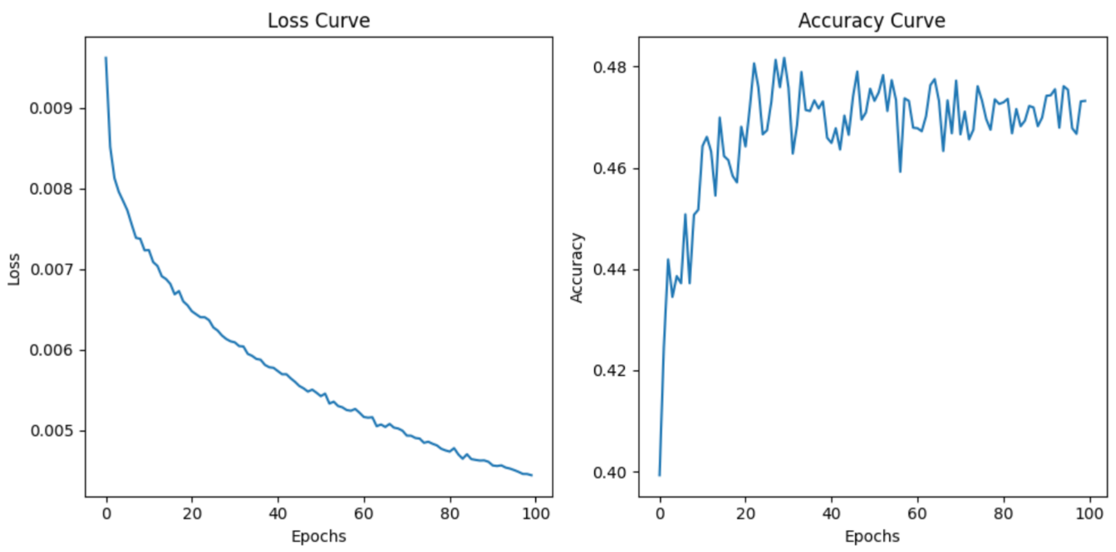

Figure 1: Loss and Accuracy of Base Model

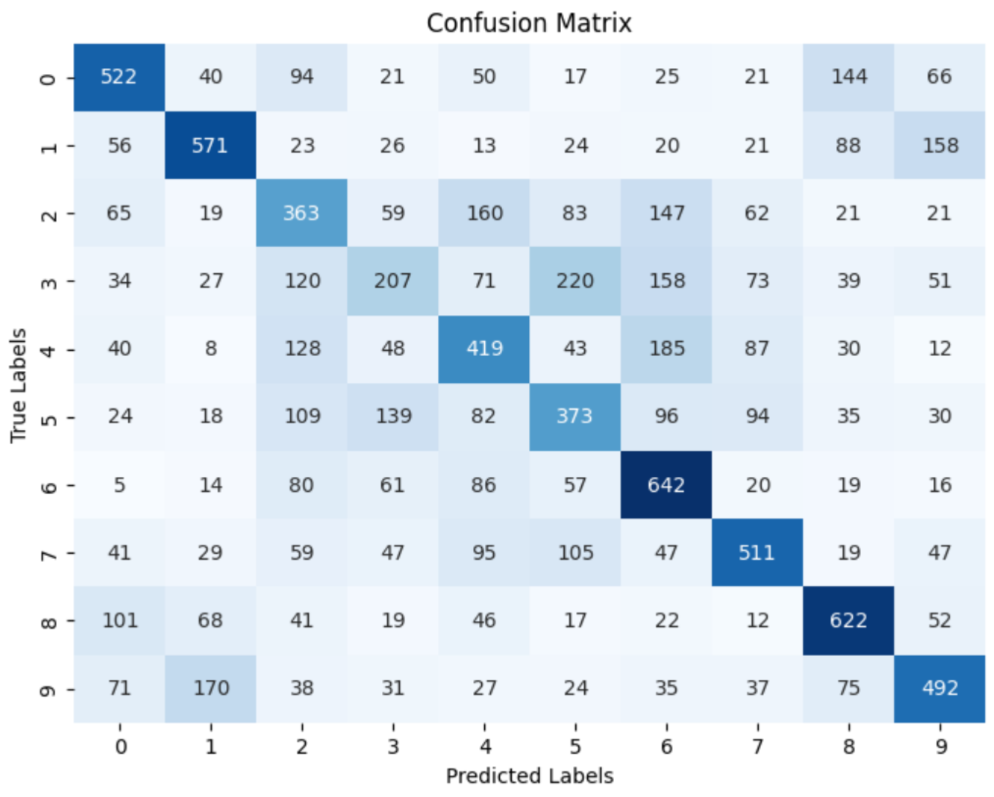

Figure 2: Confusion Matrix for Base Model

Table 2: Result For Base Model

Accuracy	F1	Test Time(s)	Training Time(s)
0.4722	0.47	0.94	94.81

From Table 2, we can find the accuracy of our model based on the default parameter is approximately 47%.

4.3.2 Batch Size optimizing

In deep learning networks, batch size is a critical parameter that directly influences the training efficiency and performance of the model. Batch size refers to the number of samples used to train the network during each gradient update. In this section of our report, we analyze the effects of different batch sizes on the performance of our base model.

Table 3: Set Up for Batch Size

Batch size	256	128	64	32

Table 4: Other Hyperparameter in Batch Size Optimizing experiment

Hidden Layer Neurons	[128, 128, 10]
Hidden Layer Nums	2
Activation function	[None, 'relu', 'softmax']
Learning rate	0.001
lr _decay	0.99
Weight decay	0
Momentum	0.9
Dropout rate	0
Batchnorm	False

The configuration of our model is summarized in Table 4 and the results of the performance evaluation based on varying batch sizes are summarized in Table 5. The metrics considered for this evaluation included accuracy, F1 score, test time, and training time.

Table 5: Performance on Different Batch Size

Batch Size	Accuracy	F1	Test Time	Training Time
256	0.4688	0.46	0.57	27.84
128	0.4729	0.47	0.56	48.21
64	0.4628	0.46	0.52	35.44
32	0.479	0.48	0.54	33.22

According to Table 5, a batch size of 32 yielded the highest accuracy of 47.90% and an F1 score of 0.48, with a training time of 33.22 seconds. In contrast, a batch size of 256 resulted in a moderate accuracy of 46.88% and an F1 score of 0.46 but had the shortest training time of 27.84 seconds. The batch size of 128 showed a slight improvement in accuracy to 47.29% and maintained an F1 score of 0.47, although the training time increased to 48.21 seconds. It was previously stated in the last section that the outcomes of various assessment indicators would be considered in making the selection. In this paper, we argue that the model's performance within a reasonable runtime becomes the primary criterion for selection. Given these results, we have decided to select a batch size of 32 as the optimal size for our model. This batch size consistently offers the best performance improvement, making it the most suitable option for our training strategy during this phase of model development.

4.3.3 Learning Rate optimizing

The learning rate is one of the most critical hyperparameters in the deep learning training process. It determines the size of the steps taken to update the model's parameters during optimization, thereby controlling the rate at which the model adjusts its weights in each iteration. Tuning the learning rate can be challenging; a high learning rate may accelerate learning but risks overshooting the optimal point or diverging. Conversely, a low learning rate ensures stability but can slow the learning process, potentially leading to prolonged training periods and the risk of settling into local minima. An appropriately set learning rate enables the network to converge quickly and stably. Therefore, we design experiments to adjust the learning rate in this paper to explore the impact of different learning rates on our model. The objective is to find an optimal learning rate that balances training efficiency and model accuracy.

Table 6: Options for Learning Rate

Learning Rate	0.0001	0.0005	0.001

Table 7: Other Hyperparameter for Learning Rate Optimizing

Hidden Layer Neurons	[128, 128, 10]
Hidden Layer Nums	2
Activation function	[None, 'relu', 'softmax']
batch size	32
lr _decay	0.99
Weight decay	0
Momentum	0.9
Dropout rate	0
Batchnorm	False

According to the performance data, a learning rate of 0.0001 resulted in the highest accuracy of 51.06% and an F1 score of 0.51, with the most efficient training time of 24.33 seconds. Conversely, a learning rate of 0.0005 yielded a slightly lower accuracy of 49.32% and an F1 score of 0.49, alongside an increased training time of 42.90 seconds. So we will use 0.0001 as the learning rate for the deep learning model.

Table 8: Performance on Different Learning Rate

Learning Rate	Accuracy	F1	Test Time	Training Time
0.0001	0.5106	0.51	0.56	24.33
0.0005	0.4932	0.49	0.86	42.90
0.001	0.4735	0.46	1.04	52.53
0.002	0.4702	0.45	1.3	54.39

4.3.4 Layer Number optimizing

In deep learning networks, the number of layers significantly influences both the performance and the complexity of the model. As the model depth increases with each additional layer, the network learns to identify more complex features. While the lower layers may capture only simple patterns, the deeper layers can recognize increasingly complex features. However, adding more layers introduces greater complexity and can lead to overfitting, particularly when the available data is limited. Additionally, deeper networks demand more computational resources and longer training times. In this context, our model performance evaluation considers training time and efficiency as key metrics. In this section, we design experiments to investigate the impact of layer quantity on network performance, aiming to identify an optimal number of layers that delivers improved performance within a reasonable training timeframe.

Table 9: Options for Layer Number

Hidden Layer Neurons	[128, 128, 10]	[128,128,128,10]	[128,128,128,128,10]
Hidden Layer Nums	2	3	4

Table 10: Other Hyperparameter for Layer Number Optimizing

Learning Rate	0.0001
Activation function	[None, 'relu', 'softmax']
batch size	32
lr _decay	0.99
Weight decay	0
Momentum	0.9
Dropout rate	0
Batchnorm	False

According to the results, the model with three layers demonstrated optimal performance, achieving an accuracy of 51.24% and an F1 score of 0.51, with a manageable increase in training time to 43.38 seconds. Conversely, the two-layer model exhibited lower accuracy at 49.37% and a similar F1 score of 0.48, albeit with a shorter training time. The four-layer model experienced a decrease in performance, with accuracy dropping to 48.40% and a significantly extended training time to 95.18 seconds.

Table 11: Performance on Different Layer Number

Layer Number	Accuracy	F1	Test Time	Training Time
2	0.4937	0.48	0.29	33.70
3	0.5124	0.51	0.72	43.38
4	0.484	0.48	0.84	95.18

4.3.5 Neurons Number optimizing

The number of neurons in a deep learning network, representing the units in each layer, plays a crucial role in the model's learning capabilities. This significance stems from the fact that the number of neurons determines the network's ability to capture and learn features from the data. Each neuron functions as a processing unit, and having more neurons enhances the network's capacity to represent complex and abstract patterns in the input data. However, an excessive number of neurons can lead to overfitting, causing the model to learn noise rather than actual features. Therefore, this paper includes experiments designed to investigate the impact of varying neuron counts on model performance. The objective of these experiments is to identify an optimal number of neurons that yields improved performance within a reasonable training time.

Table 12: Options for Neurons Number

Hidden Layer Neurons	[128,256,256,10]	[128,128,128,10]	[128,64,64,10]

Table 13: Other Hyperparameter for Neurons Number Optimizing

Hidden Layer Nums	3
Learning Rate	0.0001
Activation function	[None, 'relu', 'softmax']
batch size	32
lr _decay	0.99
Weight decay	0
Momentum	0.9
Dropout rate	0
Batchnorm	False

According to the table, the neuron configuration [128, 256, 256, 10] achieved the highest accuracy of 51.39% and an F1 score of 0.51, but it required the longest training time of 94.00 seconds. The setup [128, 128, 128, 10] achieved slightly lower accuracy at 51.24% but significantly reduced the training time to 43.38 seconds. The smallest configuration, [128, 64, 64, 10], resulted in an accuracy of 50.69% and the shortest training time of 18.32 seconds, indicating faster but less effective learning.

Table 14: Performance on Different Neurons Number

Neurons Number	Accuracy	F1	Test Time	Training Time
128,256,256,10	0.5139	0.51	1.56	94.00
128,128,128,10	0.5124	0.51	0.72	43.38
128,64,64,10	0.5069	0.50	0.33	18.32

4.3.6 Momentum optimizing

In deep learning models, the momentum coefficient is a crucial parameter in the optimization strategy. The momentum approach is designed to accelerate the convergence of stochastic gradient descent methods in the desired direction and to reduce oscillations. It achieves this by incorporating historical gradient information, effectively introducing inertia to the parameter updates. This inertia aids the optimization process in moving swiftly through flat regions and dampening oscillations in steep areas. Therefore, the setting of the momentum coefficients is critical to the model's performance. In this paper, experiments are conducted to explore the impact of momentum coefficients on deep learning models. The objective is to identify an optimal momentum setting that enhances performance while maintaining reasonable training times.

Table 15: Options for Momentum

Momentum	0.9	0.95	0.98	0.99

Table 16: Other Hyperparameter for Momentum Optimizing

Hidden Layer Neurons	[128, 256, 256, 10]
Hidden Layer Nums	3
Learning Rate	0.0001
Activation function	[None, 'relu', 'softmax']
batch size	32
lr _decay	0.99
Weight decay	0
Dropout rate	0
Batchnorm	False

According to the table, increasing the momentum value initially improved performance, with a momentum of 0.95 achieving the highest accuracy of 51.44% and an F1 score of 0.52, without significantly extending the training time. However, further increases to 0.98 and 0.99 led to reductions in both accuracy and F1 scores, dropping to 49.39% and 48.57% respectively, while also considerably prolonging the training time.

Table 17: Performance on Different Momentum

Momentum	Accuracy	F1	Test Time	Training Time
0.9	0.5067	0.51	0.94	40.99
0.95	0.5144	0.52	0.94	40.98
0.98	0.4939	0.49	0.88	117.54
0.99	0.4857	0.49	1.05	88.02

4.3.7 Weight Decay optimizing

In deep learning, L2 regularization is a commonly used weight decay technique. Its primary purpose is to prevent model overfitting and enhance the model's generalization ability. Thus, determining the appropriate regularization coefficients poses a challenge for deep learning models. Coefficients that are too large may inhibit model learning, preventing the model from fitting the data adequately. Conversely, too small coefficients may render regularization ineffective at preventing overfitting effectively. In this paper, experiments are designed to investigate the impact of weight decay coefficients on modeling and to identify the weight decay coefficient that optimally fits the model.

Table 18: Options for Weight Decay

Weight Decay	0.2	0.1	0.01	0.001	0

Table 19: Other Hyperparameter for Weight Decay Optimizing

Hidden Layer Neurons	[128, 256, 256, 10]
Hidden Layer Nums	3
Learning Rate	0.0001
Activation function	[None, 'relu', 'softmax']
batch size	32
lr _decay	0.99
Momentum	0.95
Dropout rate	0
Batchnorm	False

According to Table 20, a weight decay of 0.1 resulted in the highest accuracy of 54.61% and an F1 score of 0.54, with a training time of 127.53 seconds. Other weight decay settings showed lower performance: 0.2 led to an accuracy of 52.09%, 0.01 to 51.21%, 0.001 to 49.51%, and 0 (no decay) to 49.40%, accompanied by the longest training time of 158.45 seconds.

Table 20: Performance on Different Weight Decay

Weight Decay	Accuracy	F1	Test Time	Training Time
0.2	0.5209	0.52	0.95	129.81
0.1	0.5461	0.54	0.85	127.53
0.01	0.5121	0.51	0.92	139.52
0.001	0.4951	0.49	0.87	138.38
0	0.494	0.5	0.93	158.45

4.4 Comparison Experiment

4.4.1 Comparison for Activation

In this section, we will design comparative experiments for various activation functions. Activation functions are vital in deep learning networks, as they introduce nonlinearities into the model. It enables neural networks to learn from and perform complex tasks involving nonlinear data. Additionally, activation functions influence the flow and magnitude of the gradient, which is crucial during the backpropagation process. These experiments aim to investigate the impact of different activation functions on model performance and identify the most effective activation function for use in the model.

Table 21: Options for Activation

Activation	gelu	leaky relu	relu	tanh	logistic

Table 22: Other Hyperparameter for Activation Optimizing

Hidden Layer Neurons	[128, 256, 256, 10]
Hidden Layer Nums	3
Learning Rate	0.0001
Weight decay	0.1
batch size	32
lr _decay	0.99
Momentum	0.95
Dropout rate	0
Batchnorm	False

According to the table, the Leaky ReLU activation function achieved the highest accuracy of 54.65% and an F1 score of 0.54, albeit with longer test and training times. ReLU followed closely with an accuracy of 54.07% and a similar F1 score but offered more efficient training and test durations. GELU and Tanh delivered moderate performances with accuracies over 51%, while Logistic significantly lagged behind with only 39.21% accuracy, and it also had the longest test and training times.

Table 23: Performance on Different Activation

Activation	Accuracy	F1	Test Time	Training Time
gelu	0.5198	0.51	1.85	308.78
leaky relu	0.5465	0.54	2.71	172.37
relu	0.5407	0.54	1.11	120.324
tanh	0.5147	0.51	0.89	194.51
logistic	0.3921	0.37	5.16	385.69

4.5 Ablation Experiment

Ablation experiments are designed to assess the importance of individual components in a model. This method involves systematically removing or modifying certain parts of the model and observing the effects on its performance. Such experiments help researchers identify which components are crucial and which may be redundant. By incrementally removing parts of the model, researchers can directly observe the specific impact of each element on the overall performance. It enhances understanding of the model and helps validate theoretical hypotheses.

4.5.1 Batch Normalization

In this experiment, we explore the impact of using batch normalization and not on our model. The objective is to find if batch normalization operation will improve our model performance at a reasonable training time.

Table 24: Options for Batch Normalization

Batch Normalization	True	False

Table 25: Other Hyperparameter for Batch Normalization Optimizing

Layer size	[128,256,256,10]
Activation	[None, 'leaky relu', "leaky relu",'softmax']
Learning Rate	0.0001
epoch	30
lr _decay	0.99
momentum	0.95
batch_size	32
drop out rate	0
weight decay	0.1

The performance data clearly illustrates the significant impact of batch normalization on the model's accuracy and efficiency. With batch normalization enabled, accuracy significantly

Table 26: Performance on Different Batch Normalization

Batch Normalization	Accuracy	F1	Test Time	Training Time
True	0.3967	0.38	7.17	242.40
False	0.5465	0.54	2.71	172.37

dropped to 39.67% and the F1 score to 0.38, compared to when it was disabled, where the model achieved a higher accuracy of 54.65% and an F1 score of 0.54.
Given that our dataset used for training is already normally distributed, employing batch normalization at each layer tends to narrow the distribution and potentially results in the loss of some distinguishing features. This is likely why enabling batch normalization results in decreased accuracy. Batch normalization may be more beneficial for datasets that are not as well balanced, helping to improve model performance under those conditions.

4.5.2 Dropout

In this experiment, we aim to explore the impact of various dropout rates on the model's performance. Specifically, we seek to understand the role of dropout in the overall model architecture. To this end, we have designed experiments using different dropout rates. The dropout rates tested in this experiment are 0, 0.01, 0.05, and 0.1.

Table 27: Options for Dropout Rate

Drop Out Rate	0.1	0.05	0.01	0

Table 28: Other Hyperparameter for Dropout Rate Optimizing

Layer size	[128,256,256,10]
Activation	[None, 'leaky relu', "leaky relu",'softmax']
Learning Rate	0.0001
epoch	30
lr _decay	0.99
momentum	0.95
batch_size	32
batch norm	False
weight decay	0.1

According to the performance data, the model without dropout (rate of 0) achieved the highest accuracy of 54.93% and an F1 score of 0.55, accompanied by a relatively efficient training time of 140.33 seconds. As the dropout rate increased, there was a noticeable decrease in both accuracy and F1 scores, with the highest dropout rate of 0.1 resulting in the lowest accuracy of 48.78%. In this context, our optimal batch size is small, indicating that the dataset may be insufficient for effective dropout utilization. Utilizing a high dropout rate under these conditions can lead to the loss of many useful features.

Table 29: Performance on Different Dropout Rate

Dropout Rate	Accuracy	F1	Test Time	Training Time
0.1	0.4878	0.48	1.63	127.37
0.05	0.5151	0.51	1.61	199.55
0.01	0.5419	0.54	1.64	190.68
0	0.5493	0.55	1.97	140.33

4.6 Design of Best Model

In conclusion, we find our best model with the following parameters.

Table 30: Hyperparameter Of Best Model

Layer size	[128,256,256,10]
Activation	[None, ’leaky relu’, "leaky relu",’softmax’]
Learning Rate	0.0001
epoch	30
lr _decay	0.99
momentum	0.95
batch_size	32
drop out rate	0
batch norm	False
weight decay	0.1

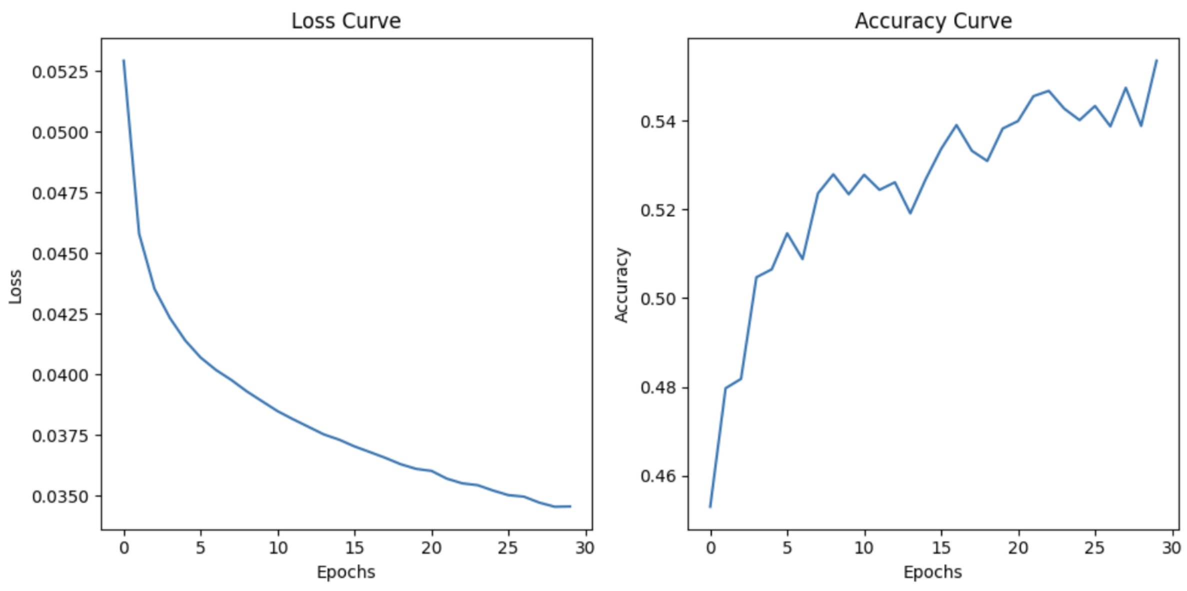

Figure 3: Loss and Accuracy of Best Model

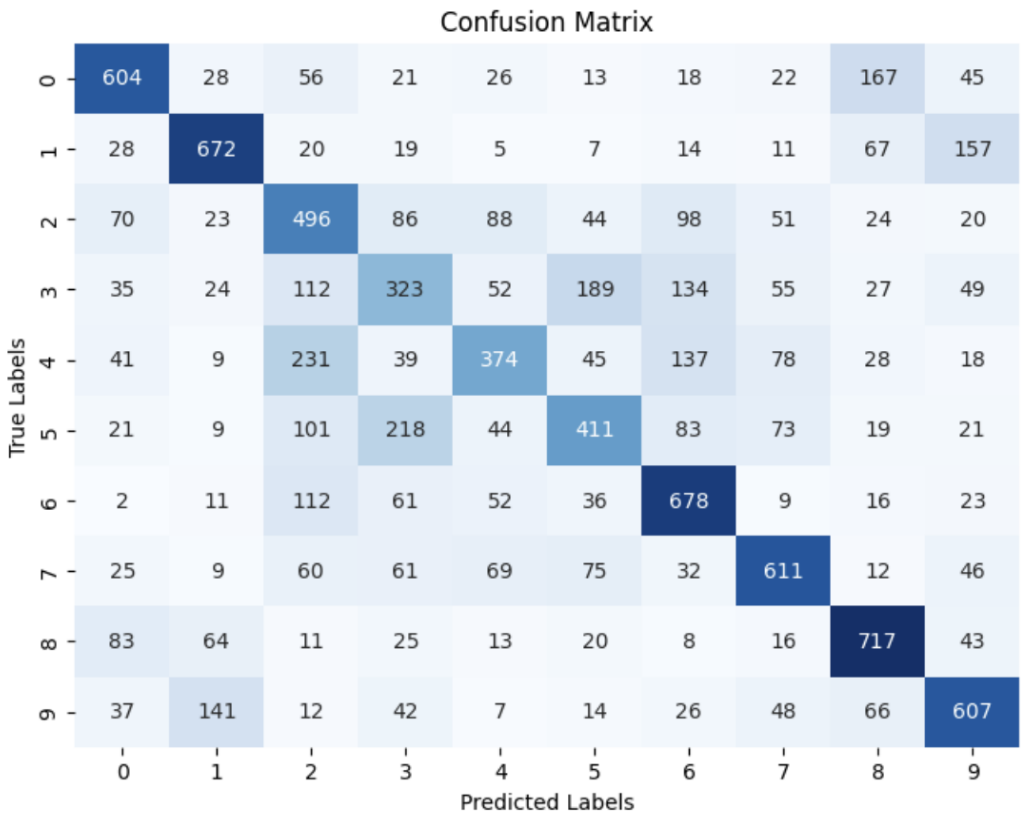

Figure 4: Confusion Matrix for Best Model

Table 31: Performance on Best Model

Accuracy	F1	Test Time	Training Time
0.5493	0.55	1.97	140.33

It is clear from Figure 3 that as the number of epochs increases, the model's loss is gradually converging, and the accuracy is progressively improving. And some differences can be observed from our initial assumptions about the optimal model configuration. Previously, we hypothesized that the model would benefit from batch normalization and dropout optimization. However, after conducting ablation and comparison experiments, we found that the model performed best without these techniques. This outcome may be attributed to the highly normalized nature of our dataset, where each label has an equal number of training samples. Using batch normalization in this context might introduce unnecessary complexity that could degrade model performance. Additionally, batch normalization might cause internal covariate bias, leading to further complications. The observed decline in model performance following the implementation of dropout could be due to the limited data available and the insufficient number of neurons in each layer, potentially diminishing the model's learning capability and resulting in suboptimal performance.

Figure 5 illustrates the changes in test accuracy for our model following a series of parameter adjustments, activation function comparisons, and ablation experiments. The initial baseline model achieved a test accuracy of 0.472. During the parameter adjustment phase, improve-

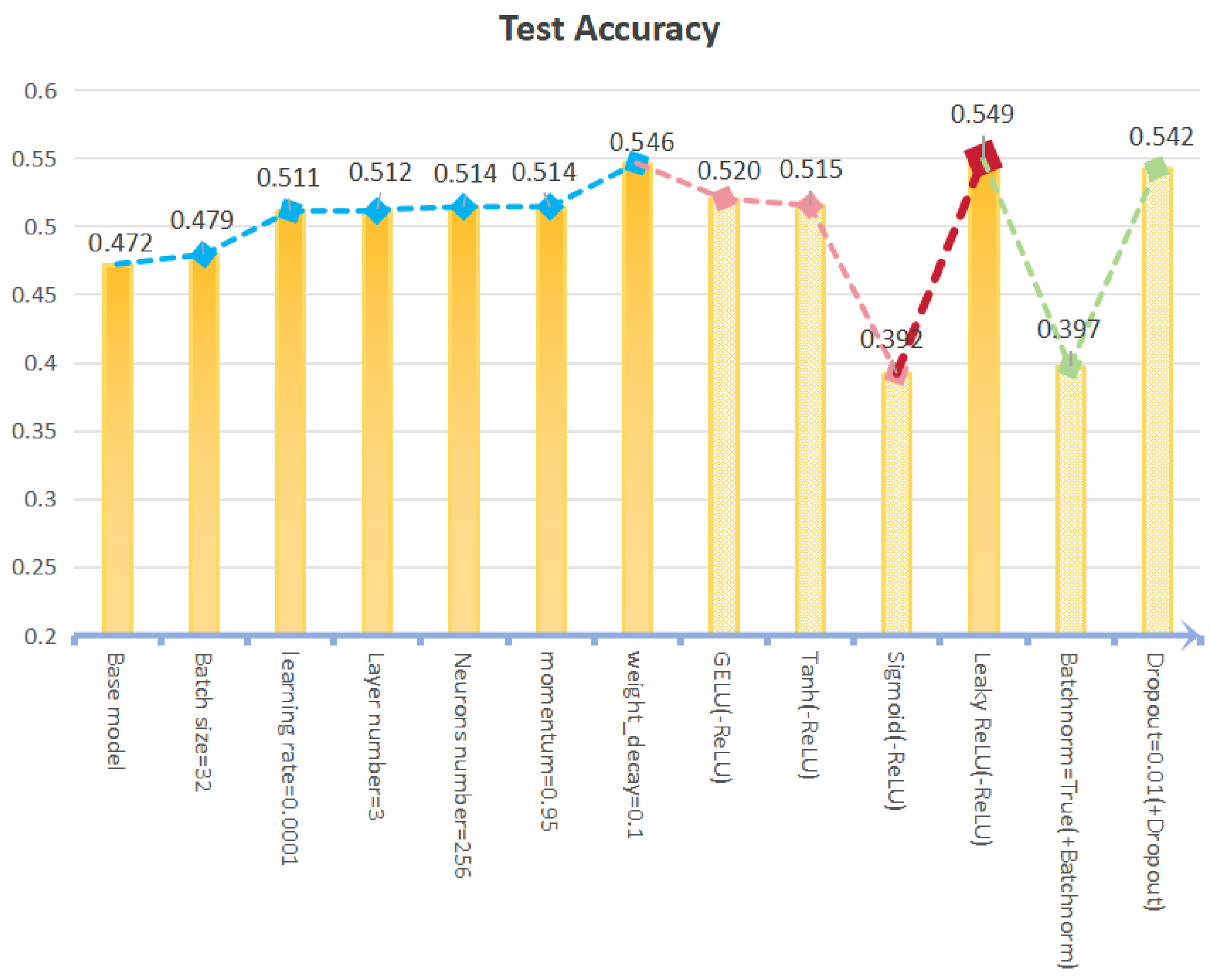

Figure 5: Convnext for our model (Liu et al., 2022)

ments in test accuracy were achieved by modifying the batch size, adjusting the learning rate, and changing the number of layers and neurons. Hyperparameter optimization revealed that batch size, learning rate, and weight decay significantly impact the model, while other hyperparameters only slightly affect performance. Comparative experiments with activation functions also highlighted the positive effects of Leaky ReLU. The accuracy reached 54.9%, which is the highest in our model. Consequently, we retained this configuration as our best model.
Finally, we found that adding batch normalization or dropout will negatively affect accuracy in the ablation studies. Therefore, we opted not to use these two techniques in our model, and this decision was crucial to our process of selecting the best model.

5 Summary and Reflection

The research presented in this paper focuses on developing more effective multiclassification models. Various data preprocessing techniques were employed in constructing the model, such as data normalization and one-hot encoding. L2 regularization was implemented as a weight decay strategy, while dropout and batch normalization were used to address overfitting. Additionally, momentum in SGD was tested as an optimizer to enhance and expedite the network's training process. The cross-entropy function was chosen for the multiclassification task as the loss function. The optimal configuration of the model was established after conducting comparative and ablation experiments.
We have also identified several areas for further consideration and reflection. Firstly, we experimented with a more comprehensive range of deep learning network structures and alternative loss functions, including mean squared error (MSE) and Kullback-Leibler divergence loss. However, these did not significantly improve model performance. Future work might explore integrating these loss functions and adjusting their weights through hyperparameters. Secondly, we considered the potential impact of using approximate solutions with the GELU activation function. Despite findings suggesting that the approximation has minimal effect on performance compared to the exact mathematical process, we believe further experimentation is warranted. Lastly, our initial efforts at mathematical derivation, moving away from standard tutorial code, introduced numerous challenges and inefficiencies. This experience underlines the need for more vital foundational skills in mathematical derivation to facilitate more independent model development. These reflections are crucial for advancing our ability to construct neural networks without relying heavily on library functions.

References

Bridle, J. S. (1990). Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. *Neurocomputing*, 227–236. doi: 10.1007/978-3-642-76153-9_28

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. Retrieved from https://arxiv.org/abs/1502.01852

Hendrycks, D., & Gimpel, K. (2020). Gaussian error linear units (gelus). *arXiv:1606.08415 [cs]*. Retrieved from https://arxiv.org/abs/1606.08415

Hoerl, A. E., & Kennard, R. W. (2000). Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, *42*(1), 80. doi: 10.2307/1271436

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv:1502.03167 [cs]*. Retrieved from https://arxiv.org/abs/1502.03167

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90. doi: 10.1145/3065386

Liu, Z., Mao, H., Wu, C.-Y., Feichtenhofer, C., Darrell, T., & Xie, S. (2022). A convnet for the 2020s. *arXiv:2201.03545 [cs]*. Retrieved from https://arxiv.org/abs/2201.03545

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). *Rectifier nonlinearities improve neural network acoustic models.* Retrieved from https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, *4*(5), 1–17. doi: 10.1016/0041-5553(64)90137-5

Robbins, H., & Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, *22*(3), 400–407. Retrieved from https://www.jstor.org/stable/2236626

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. doi: 10.1038/323533a0

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, *27*(3), 379–423.

United States. National Archives And Records Service. Office Of Records Management. (1972). *Information retrieval.* Washington: General Services Administration, National Archives And Records Service, Office Of Records Management.