COMP539 Deep Learning Assignment1

Group members:
Binghang Li(530191027)
Simiao Yu(520428764)
Qiuhan Li(530748315)

Abstract

This study examines the efficacy of a multilayer perceptron (MLP) for multi-class classification tasks without relying on automated gradient frameworks like PyTorch. By manually analyzing and adjusting the MLP's key components—activation functions, optimizers, regularizers, and batch normalization—this research highlights how strategic modifications can enhance model performance. Findings suggest that targeted hyperparameter adjustments significantly improve accuracy and generalization, reaffirming the MLP's versatility in deep learning applications. This study emphasizes the potential of basic neural network architectures to remain relevant amid advancing AI technologies.

1 Introduction

1.1 Study Aim

As a pioneer in artificial intelligence, deep neural networks (DNN) have played a key role in improving our ability to interpret complex patterns and enhance decision-making processes. The multilayer perceptron (MLP) is a basic DNN architecture that has instructive value in explaining the nature of neural networks. This research aims to dissect and reconstruct MLPs for multi-class classification tasks, avoiding dependence on automatic gradient frameworks such as PyTorch. Our study aims to reveal the complex mechanisms of the network by paying close attention to the main components: activation function, optimizer, regularizer, loss function and batch normalization.

Due to the lack of automatic differentiation, our systematic approach allows for a nuanced assessment of how individual components within an MLP impact learning capabilities. At the heart of our investigation are ablation studies that help extract optimal model parameters and discern their operational efficacy. Through empirical testing, this study not only aims to confirm the theoretical propositions of previous studies, but also to verify the applicability of these principles in actual model configurations.

1.2 Significance of the Study

The significance of this research is that it demonstrates the adaptability of MLPs, extending their usefulness beyond the fields of natural language processing and image recognition, where Transformer-like models currently dominate. This effort contributes to a growing body of research that highlights the versatility and enduring relevance of MLPs. Our commitment to an exhaustive exploration of MLPs is to reveal underlying aspects of neural network functionality. These revelations are expected to spur breakthroughs critical to the advancement of deep learning, establishing new benchmarks for both academic and applied artificial intelligence fields.

2 Methods

2.1 Preprocessing

2.1.1 One Hot Encoding

One-hot encoding is a technique used to convert categorical data into a numerical format that machine learning algorithms can interpret more effectively. Each unique category within a feature is transformed into a new binary attribute, representing the presence or absence of that category with a 1 or 0.

For a categorical feature with n unique categories, each category k is represented as a vector v_k of length n. The vector contains a '1' in the k-th position and '0's in all other positions:

$$v_k[i] = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases}$$

In multi-class classification tasks, each class is typically represented by a unique natural number. However, directly feeding these numerical labels to a learning algorithm or neural network can lead to incorrect interpretations by the model, as it might treat these labels as continuous data points. Such an approach could imply an unintended ordinal relationship among categories (e.g., assuming that one category is "greater" than another based on its label).

This misinterpretation can cause the model to assume dependencies among the categories that do not exist, potentially leading to flawed outcomes. To prevent this issue and ensure that each class is treated as a distinct and independent entity, we implement one-hot encoding of the labels. One-hot encoding transforms each label into a separate binary vector, eliminating any false ordinal or numerical relationships and allowing the model to focus on the true structure of the categorical data. This approach is critical in ensuring accurate and reliable model training and inference in multi-class classification scenarios.

2.1.2 Standardization

Data standardization is an essential preprocessing step in machine learning that involves rescaling the attributes of the data to a common scale. This process is vital as it prevents the model from assigning undue importance to features simply based on their magnitude. Without standardization, attributes with larger ranges could dominate the model's learning process, potentially leading to biased or inefficient outcomes.

For our project, we have selected Z-score normalization due to its effectiveness in handling outliers and its ability to maintain the distribution of the data. Z-score normalization, also known as standard score normalization, adjusts the features of the data to have zero mean and a variance of one. This technique transforms each data point using the formula:

$$z = \frac{x_i - \overline{x}}{\sigma}$$

where x_i represents an individual data point, $\overline{x}$ is the mean of the data points for a particular feature, σ is the standard deviation of that feature.

By applying Z-score normalization, we ensure that each feature contributes equally to the analysis, thereby avoiding bias due to the inherent differences in unit scales of the features.

2.2 Module Principles

2.2.1 Activation Functions

Activation functions introduce non-linear properties to the model, enabling it to learn complex mappings between inputs and outputs. We utilized:

ReLU (Rectified Linear Unit) for hidden layers, defined as:

$$f(x) = \max(0, x)$$

Softmax for the output layer, which transforms logits into probabilities:

$$z_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

where z is the input vector, K is the number of classes, and i is the index of a particular class.

2.2.2 Weight Decay

Implemented as L2 regularization to prevent overfitting by adding a penalty to the loss function based on the magnitude of the weights:

$$L_{\text{reg}} = L + \frac{\lambda}{2m} \sum_{i} w_i^2$$

where L is the original loss, λ is the regularization term, m is the batch size, and w_i are the weights.

2.2.3 SGD

Stochastic Gradient Descent (SGD) with Momentum was employed to update the network's parameters. The update rule for the weights is given by:

$$W(t+1) = W(t) - v(t+1)$$

where $v(t+1)$ is the velocity term, η is the learning rate, and t indicates the iteration number. The velocity term is a linear combination of the previous velocity and the current gradient, with the momentum coefficient μ controlling the blend:

$$v(t+1) = \mu v(t) + (1-\mu)\nabla W_L$$

2.2.4 Cross entropy loss

We adopted the Cross-Entropy Loss function for multi-class classification tasks, combined with the Softmax function. For a single sample, the loss is:

$$L(y, \hat{y}) = -\sum_{c=1}^{C} y_c \log \hat{y}_c$$

where y is the one-hot encoded true label vector, $\hat{y}$ is the predicted probability vector from the Softmax function, and C is the number of classes.

2.2.5 Batch Normalization

Each mini-batch goes through a normalization step defined as follows:

$$X^{(k)} = \frac{X^{(k)} - B}{\sqrt{B2 + \epsilon}}$$

where $X^{(k)}$ is the input for the mini-batch, B is the mini-batch mean, $B2$ is the mini-batch variance, and ϵ is a small constant for numerical stability. The normalized values are then scaled and shifted using learned parameters γ and β, where:

$$y^{(k)} = \gamma X^{(k)} + \beta$$

2.2.6 Mini-Batch Training

The network was trained using mini-batch gradient descent. In each epoch, the training data was divided into batches of a predefined size. The network processed each batch sequentially, performing forward and backward propagation to calculate gradients and update the parameters.

The backward propagation involved computing the gradient of the loss function concerning each parameter. For a layer l with activation $a^{(l)}$ and weights $W^{(l)}$, the gradients were computed as:

$$\Delta W^{(l)} = \frac{1}{m^{(l)}} a^{(l-1)T}$$

The forward and backward computations and parameter updates were iterated until convergence, as evidenced by the stabilization of the loss function or the attainment of a predetermined number of epochs.

2.3 Optimal Model Design

2.3.1 Model Architecture

The neural network architecture for the study was meticulously crafted, comprising:

- **Input Layer:** Corresponds to the number of features in the dataset.
- **Hidden Layers:** Two hidden layers equipped with 128 and 64 neurons, respectively, utilizing the ReLU activation function. This choice is advantageous for mitigating the vanishing gradient problem and expediting the training process.
- **Output Layer:** Utilizes the Softmax activation function, ideal for multi-class classification tasks by producing a probability distribution across the classes.

2.3.2 Hyperparameters

Optimized hyperparameters include:

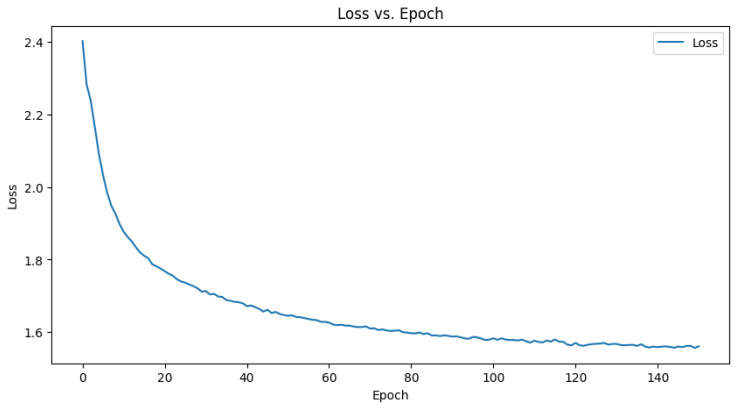
- **Learning Rate:** Set at 0.04, chosen to balance between rapid convergence and maintaining stability during training.
- **Batch Size:** 500, to optimize computational efficiency and accuracy in gradient estimation.
- **Dropout Rate:** 0.3, applied in hidden layers to prevent overfitting.
- **Momentum:** 0.7, to aid in overcoming local minima and accelerate convergence.
- **Batch Normalization:** Excluded to prevent potential negative interactions with the effectiveness of dropout.
- **L2 Regularization (Lambda):** Set at 0.003 to provide mild regularization without overly penalizing the magnitude of weights.

2.3.3 Training Details

- **Epochs:** Up to 200, with an early stopping mechanism based on a convergence threshold.
- **Convergence Threshold:** A 0.1% change in loss was utilized to prevent overfitting, ensuring the model ceased training once no significant improvements were observed.

2.3.4 Results

After completing the training process, The loss vs epoch are shown below.



The model achieved:

- **Final Training Accuracy:** 57.59%
- **Final Test Accuracy:** 53.20%

3 Experiments

For the assigned classification task, the lecturer has supplied a dataset along with preliminary code and modules, which encompass the neural network's architecture, gradient optimization techniques, and regularization methods. The next phase of the project involves refining the fundamental structure of the neural network to optimize its performance. Furthermore, the integration of additional modules, including dropout, batch normalization, and mini-batching, is necessary to enhance the neural network's functionality.

3.1 Evaluation Metric Performance

To evaluate the performance of our neural network model, we utilized two principal metrics: accuracy and loss.

Accuracy is defined as the proportion of correctly predicted observations to the total observations. It is an indicator of the model's overall effectiveness in classification:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{1}$$

Loss, also known as cost, measures the prediction error of a model. It represents how far off the model's predictions are from the actual values. Lower loss values are indicative of better model performance. The Cross-Entropy Loss used in our model for multi-class classification has been introduced in section 2.2.4. Comprehensive Analysis Throughout the development of our neural network model, a thorough hyperparameter tuning process was conducted. This optimization aimed to enhance the model's predictive accuracy and minimize the loss function. The table below summarizes the various configurations tested and their corresponding performance metrics:

Table 1: Hyperparameter Tuning Results

Parameter	**Training Accuracy (%)**	**Test Accuracy (%)**	**Loss**
Hidden Layer Neurons			
[128, 64, 32]	54.25	50.29	1.46
[128, 64]	57.59	53.2	1.39
[64, 32]	51.79	48.71	1.56
Batch Size			
400	57.88	52.34	1.37
500	57.59	53.2	1.39
600	56.36	53.52	1.35
Learning Rate			
0.03	57.05	52.69	1.41
0.04	57.59	53.2	1.39
0.05	58.01	53.51	1.37
Momentum			
0.5	58.04	52.62	1.36
0.7	57.59	53.2	1.39
0.9	58.34	53.16	1.37
Lambda			
0.001	56.45	53.2	1.31
0.003	57.59	53.2	1.39
0.005	58.1	53.5	1.35
Dropout Rate			
0.2	58.74	52.09	1.21
0.3	57.59	53.2	1.39
0.4	56.21	54.31	1.44

After extensive testing and analytical evaluation, the following hyperparameters were adopted for the final model:

- **Hidden Layer Neurons:** $[128, 64]$
  The architecture with two hidden layers, consisting of 128 and 64 neurons, struck the best balance between model complexity and computational efficiency while maintaining high accuracy levels.
- **Learning Rate:** 0.04
  This learning rate was chosen for its effectiveness in achieving convergence to a lower loss, without compromising the stability of the optimization process.
- **Batch Size:** 500
  A batch size of 500 was determined to optimally balance the stochastic nature of the gradient descent and the benefits of vectorized operations.
- **Dropout Rate:** 0.3
  A 30% dropout rate was applied to prevent overfitting and to bolster the model's ability to generalize.
- **Momentum:** 0.7
  The momentum parameter was set to 0.7 to facilitate a faster convergence during training and to assist in navigating past local minima.
- **Batch Normalization:** *False*
  Batch normalization was deactivated in our final model configuration due to its negligible benefits when used in conjunction with dropout in our particular network setup.
- **Lambda (L2 Regularization):** 0.003
  An L2 regularization parameter, lambda, was set to 0.003, effectively reducing overfitting by penalizing large weights, without causing a significant decline in the model's fitting capacity.

4 Model Justification

The development of our neural network model was guided by a series of empirical evaluations and theoretical considerations aimed at achieving an optimal balance between performance and computational efficiency. This section justifies the final model's design, emphasizing the rationale behind the chosen architecture and hyperparameters.

Model Architecture: The architecture was chosen to reflect a compromise between depth and complexity. While deeper networks can model more complex relationships, they also require more data and computational power, and they can overfit if not carefully regularized. The two-layer structure with 128 and 64 neurons was found to be sufficiently complex to capture the necessary patterns in the data without leading to overfitting, as evidenced by the model's stable test accuracy.

Learning Rate: A learning rate of 0.04 facilitated steady convergence to an optimal set of weights. Higher rates were tested but led to instability in the learning process, while lower rates slowed down convergence significantly, impacting the training time without substantial gains in accuracy.

Batch Size: After experimenting with various batch sizes, 500 emerged as the optimal choice. Smaller batches, while offering more frequent updates, did not yield significant improvements in model performance to justify the increased computational cost. Conversely, larger batches did not facilitate the fine-grained update resolution required for this model's training.

Dropout Rate: The selected dropout rate of 0.3 serves as an effective regularization method to combat overfitting. It was determined to be the sweet spot where it discouraged complex co-adaptations of neurons on training data without degrading the network's capacity to learn from the data provided.

Momentum: A momentum value of 0.7 was used to accelerate convergence and prevent the optimization process from getting stuck in local minima. The value was carefully tuned to maintain a balance between rapid advancement in the loss landscape and the risk of overshooting the global minimum.

Batch Normalization: Our decision to forgo batch normalization was based on its observed impact on the training dynamics in conjunction with dropout. Given that both techniques aim to regularize the model, their simultaneous application was redundant and sometimes counterproductive in our experiments.

Lambda (L2 Regularization): The lambda value for L2 regularization was set to 0.003 to penalize large weights and thus prevent overfitting. This value was sufficiently large to impose a regularization effect, yet small enough to avoid overly constraining the model's ability to fit the training data.

Empirical Validation: The final configuration was validated through a series of experiments measuring the training and validation performance across epochs. The results affirmed the robustness of the model, as indicated by stable loss and accuracy curves, and consistent test accuracy metrics.

5 Discussion and Conclusion

This research embarked on a journey to dissect and optimize a multilayer perceptron for multi-class classification tasks. Our findings indicate that the chosen MLP architecture, coupled with the strategic selection of hyperparameters, provides a strong foundation for such tasks.

Discussion: The experimental results have underscored the critical role of each hyperparameter in the model's performance. The use of ReLU activation functions in the hidden layers, a medium batch size, and a moderate learning rate, for instance, contributed to the model's ability to generalize well from the training data to unseen data. Furthermore, the incorporation of dropout and L2 regularization effectively mitigated the risk of overfitting, which is often a challenge in neural network models.

One of the key limitations of this study is the gap between training and test accuracy, which may suggest that there is still some degree of overfitting or that the model could benefit from additional data or further tuning. Moreover, given the rapid advancement in deep learning methodologies, there might be newer architectures or optimization techniques that could further enhance the model's performance.

Conclusion: In conclusion, the study has demonstrated that MLPs remain a viable and powerful tool for classification tasks, even in an era where more complex models are in vogue. The ability to fine-tune an MLP with careful consideration of its hyperparameters can yield a model that not only performs well but also provides insights into the underlying mechanisms of learning within deep networks.

Future Work: Looking forward, we recommend exploring the following avenues:

- Expanding the dataset to provide the model with a more diverse and comprehensive learning experience.
- Experimenting with alternative regularization techniques such as dropout variations or novel penalty functions.
- Investigating the benefits of more sophisticated optimization algorithms that may further reduce the loss and improve the generalization capability of the model.
- Considering the adoption of newer neural network architectures such as Transformers, which have shown promising results in various domains.

Ultimately, this study serves as a testament to the enduring relevance of MLPs and provides a roadmap for their optimization in contemporary machine learning tasks. Our approach highlights the importance of methodical experimentation and in-depth understanding of each component's impact on the model's learning and performance.

Appendix

How to Run the Code:

1. Access the code and datasets in the Google Cloud folder at: `Google Drive Link`.
2. Open the notebook `codes.ipynb` in Google Colab.
3. Upload the dataset files `train_data.npy`, `test_data.npy`, `train_label.npy`, and `test_label.npy` to the `'/content/'` directory in Colab. Adjust the file paths in the code as needed if using a different directory.

4. Ensure all files are completely uploaded before starting the execution of the code.
5. Execute all cells in the notebook by pressing `Ctrl+F9` or by selecting *Runtime* → *Run all* from the Colab menu.