Image Classification Based on Multilayer Perceptron

Authors: Huaizhi Yin

Date: 2024-01-07T00:00:00+00:00

Abstract

Multilayer Perceptron (MLP) is a feedforward neural network that overcomes the limitations of linear models and opens the door to deep learning by incorporating one or more hidden layers into the network. This paper utilizes MLP to accomplish image classification, conducting exploration on the Fashion MNIST dataset and attempting to transfer to the MNIST dataset. On Fashion MNIST, after performing feature preprocessing, we selected different optimization methods and conducted comparisons. Additionally, by incorporating regularization methods such as dropout and weight decay, we achieved optimization and improvement of the multilayer perceptron. Experimental results demonstrate that appropriate feature processing can enhance the numerical stability of the model. The momentum method significantly improves model performance, while methods such as weight decay contribute to improving the model's generalization capability.

Full Text

Preamble

Beijing University of Chemical Technology
Graduate Course Paper

Course Name: Deep Learning
Course Code: Comp51801
Instructor: Li Ruirui
Completion Date: January 5, 2024
Major: Computer Science and Technology
Student ID: 2023200833
Name: Yin Huaizhi
Grade:

Image Classification Based on Multi-Layer Perceptron

Multi-Layer Perceptron (MLP) is a feedforward neural network that overcomes the limitations of linear models by adding one or more hidden layers, opening the door to deep learning. This paper utilizes MLP to accomplish image classification, exploring the Fashion MNIST dataset and attempting to transfer to the MNIST dataset. On Fashion MNIST, we performed feature preprocessing, compared different optimization methods, and implemented regularization techniques such as dropout and weight decay to optimize and improve the MLP.

Experiments demonstrate that appropriate feature processing enhances numerical stability of the model. The momentum method significantly improves model performance, while techniques like weight decay help enhance generalization.

Keywords: Multi-Layer Perceptron; Deep Learning; Feedforward Neural Network

---

1.1 Introduction

This paper addresses the task of image classification, selecting the classic Fashion MNIST and MNIST datasets for training. The primary algorithm employed is Multi-Layer Perceptron, which is improved and compared against simple linear classification algorithms (softmax regression).

1.1.1 Dataset Introduction

(1) MNIST Dataset
MNIST consists of 70,000 handwritten digit images created by American high school students and Census Bureau employees. Each image is 28$×$28 pixels and labeled with its represented digit. This dataset is widely used and often referred to as the "Hello World" of machine learning.

[Figure 1: see original paper]-1: Overview of MNIST Dataset

(2) Fashion MNIST Dataset
Fashion MNIST can be considered a direct replacement for MNIST, sharing the exact same format (70,000 grayscale images, each 28$×$28 pixels, with 10 classes). However, these images represent fashion items, making it more challenging. Some machine learning algorithms perform well on MNIST but only achieve mediocre performance on Fashion MNIST. For example, KNN and Softmax regression achieve accuracies of 98% and 93% on MNIST, respectively, but only 84% and 83% on Fashion MNIST [1]. Therefore, Fashion MNIST is more suitable for evaluating the effectiveness of machine learning algorithms for image classification.

[Figure 1: see original paper]-2: Overview of Fashion MNIST

1.1.2 Algorithm Introduction

This paper employs Multi-Layer Perceptron (MLP) as the primary technique. MLP is chosen mainly due to the massive scale of data and the image classification task, which are difficult to solve satisfactorily using traditional machine learning methods. Among deep learning methods, MLP is easier to implement than Convolutional Neural Networks while still achieving considerable effectiveness in image classification.

Multi-Layer Perceptron, also known as a multi-layer neural network, consists of an input layer, one or more hidden layers, and a final output layer. Except for the output layer, each layer includes bias neurons and is fully connected to the next layer (i.e., fully connected layers).

[Figure 1: see original paper]-3: Structure of a Simple Multi-Layer Perceptron

To train more complex non-linear models, activation functions must be added to hidden layers. Common activation functions include Sigmoid, ReLU, etc.

Compared to traditional machine learning algorithms, MLP training requires more iterations, occupies larger model space, and needs more training data, but often achieves higher accuracy. MLP is suitable for applications where interpretability is not critical, such as image classification and speech recognition, and is widely used in data mining, machine learning, and pattern recognition [1].

The effectiveness of MLP is also influenced by optimization methods, loss function settings, and generalization techniques. This paper implements several improvements, which will be detailed later.

Chapter 2: Construction of Multi-Layer Perceptron

Multi-Layer Perceptron is a multi-layer feedforward neural network. During training, signals propagate forward while errors propagate backward. Therefore, MLP construction primarily considers two aspects: forward propagation and backward propagation.

Additionally, several details must be considered during MLP construction, such as weight initialization and loss function selection, which will be introduced here.

2.1 Forward Propagation Implementation

During training, MLP combines forward propagation and backward propagation, while prediction only requires forward propagation from input to output. To implement forward propagation, we defined several classes to realize essential neural network structures including weight layers, bias layers, activation

function layers, and softmax layers. The following is the forward propagation architecture of our MLP:

A brief introduction to each component in the diagram:

[Figure 2: see original paper]-1: Forward Propagation Architecture Diagram

(1) **inputLayer**: Input vector of size 1×n, becoming a matrix when multiple samples are present.

(2) **MultiplyGate**: Multiplication unit that defines a forward function to implement forward propagation as follows:

$$output = input \times W_{n \times m}$$

(3) **AddGate**: Addition unit that defines a forward function to implement forward propagation as follows:

$$output = input + b_{1 \times m}$$

(4) **ActivateLayer**: Activation function layer. This paper uses the ReLU activation function, whose forward function implements forward propagation as:

$$\text{ReLU: } output = \begin{cases} input & \text{if } input > 0 \\ 0 & \text{otherwise} \end{cases}$$

To improve generalization performance during training, we designed a Dropout layer. Dropout is a common technique for training neural networks that injects noise while computing each internal layer during forward propagation. It is called dropout because it appears to drop out some neurons during training. In each iteration of the entire training process, standard dropout involves setting some nodes in the current layer to zero before computing the next layer [2].

Generally, dropout is not applied to input and output layers; it is only used in hidden layers. When applying dropout to hidden layers, each intermediate hidden layer output value h is replaced by a random variable h' with dropout probability p as follows:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{with probability } 1-p \end{cases}$$

According to this design, the expectation remains unchanged, i.e., $E[h'] = h$. When not using dropout, simply set the dropout parameter for that layer to 0.

(5) **softmax Layer**: The final layer typically has no activation function. After passing through bias neurons, the output does not directly represent the probability of a sample belonging to a certain class; these values require softmax function processing for unified computation.

For a K-class classification problem, Softmax regression trains K linear models $s_1, s_2, \ldots, s_k$, then uses the softmax function to output the probability of an instance belonging to class k:

$$P(y = k|x) = \frac{e^{s_k(x)}}{\sum_{j=1}^{K} e^{s_j(x)}}$$

2.2 Backward Propagation Implementation

This design employs reverse-mode automatic differentiation for backward propagation. The fundamental principle of automatic differentiation is that all numerical computations can be decomposed into basic operations including addition, subtraction, multiplication, division, and elementary functions such as exp, log, sin, cos, etc., then uses the chain rule to automatically compute gradients of composite functions.

Suppose we need the gradients of loss function $f(x; w, b)$ with respect to all parameters w and b. First, we decompose the composite function $f(x; w, b)$ into a series of basic operations and construct a computational graph. A computational graph is a graphical representation of mathematical operations where each non-leaf node represents a basic operation and each leaf node is an input variable or constant [3].

[Figure 2: see original paper]-2: Example of a Computational Graph

Reverse mode recursively computes gradients in the opposite direction of the computational graph's computation flow, applying the chain rule for partial derivative calculations [4]:

$$\frac{\partial f}{\partial x_i} = \sum_{j:i \in \text{parents}(j)} \frac{\partial f}{\partial u_j} \frac{\partial u_j}{\partial x_i}$$

For a general function $\mathbb{R}^N \to \mathbb{R}^M$, forward mode requires one pass per input variable (totaling N passes), while reverse mode requires one pass per output (totaling M passes). When $N > M$, reverse mode is more efficient. In feedforward neural network parameter learning, the risk function is $f : \mathbb{R}^N \to \mathbb{R}$ with scalar output, making reverse mode the most efficient computation method requiring only one pass. Intermediate results are saved during forward propagation and used sequentially from back to front during backward propagation to compute gradients.

Applying computational graph concepts, each class corresponding to forward propagation units in this design includes both forward functions for forward propagation and backward functions for error backpropagation:

Assuming the gradient from loss function f to the current layer's output is dZ:

(1) **MultiplyGate**: Multiplication unit that defines a backward function for backpropagation as:

$$dW = X^T \times dZ$$

(2) **AddGate**: Addition unit that defines a backward function for backpropagation as:

$$db = M \times dZ$$

where M is represented in code as `np.ones((1, dZ.shape[0]))`, having the same shape as b with all elements equal to 1.

(3) **ActivateLayer**: Activation function layer. This paper implements ReLU, Sigmoid, Tanh, and other activation functions, whose backward functions implement backpropagation as follows (let activation function be $\sigma(x)$):

- ReLU: $\sigma'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$
- Sigmoid: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- Tanh: $\sigma'(x) = 1 - \sigma^2(x)$
- LeakyReLU: $\sigma'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{otherwise} \end{cases}$

(4) **softmax Layer**: This paper uses cross-entropy loss function. The derivative process for softmax layer with cross-entropy loss yields a remarkably simple expression:

$$\frac{\partial L}{\partial s_i} = p(i) - y_i$$

where $p(i)$ is the predicted probability and y_i is the true label.

2.3 Overall Code Structure

Our overall code architecture is shown below:

[Figure 2: see original paper]-3: Overall Code Framework

The code encapsulates an optimizer class in `optimizer.py` and activation functions in `Lay.py`. The core lies in the Model class. When defining a Model object,

one can specify MLP layer structure, dropout probabilities for each layer, optimizer (a class object from `optimizer.py`), and activation function (a class object from `Lay.py`). The Model class completes member object initialization, where weight W and bias b initialization uses the Xavier method to improve numerical stability: assuming a layer has n_{in} input neurons and n_{out} output neurons, elements in the Xavier-initialized weight matrix follow the distribution:

$$W_{ij} \sim \mathcal{U}\left[-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right]$$

Member variables `self.vb` and `self.vw` store previous gradient information for use in certain optimization methods, initialized to 0.

The Model class contains member functions: `train`, `predict`, `calculate_{loss}`, and `clear`, for model training, prediction, loss calculation, and model re-initialization, respectively. The first three functions automatically construct AddGate, MultiplyGate, and Softmax objects for sequential forward propagation and backward propagation calculations.

Additionally, the project includes model evaluation functions encapsulated in `evaluate.py`. Due to time constraints, two functions were completed for computing confusion matrices and plotting ROC curves.

Chapter 3: Training and Improvement of Multi-Layer Perceptron

The training and improvement of our implemented MLP primarily occurs on the Fashion MNIST dataset, following the general deep learning workflow shown below:

[Figure 3: see original paper]-1: MLP Training Flowchart

3.1 Dataset Reading and Visualization

Fashion MNIST's training and test sets are in CSV format, containing labels and pixel grayscale values. When reading data, the label represents the image category, corresponding to the following table:

.1: Fashion MNIST Label Categories

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress

Label	Description
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

The visualized images appear as follows:

[Figure 3: see original paper]-3: Fashion MNIST Visualization

Combined with [Figure 1: see original paper]-2, we can see that items in this dataset have various patterns, shapes, and colors, posing challenges for recognition.

3.2 Feature Processing

Our model first improves feature processing. We built a neural network with two hidden layers (256 and 128 neurons respectively), no dropout, using mini-batch stochastic gradient descent with batch size 128 and learning rate 1.28×10^{-3}.

We compared models with and without feature processing, evaluating training and test set accuracies. Without feature processing, the 784 pixel grayscale values are directly used as input. With feature processing, each pixel's grayscale value is log-transformed:

$$\text{Input} = \ln(\text{image} + 1)$$

Both models were trained for 5 epochs (each epoch traverses all 60,000 samples once). Curves of training loss, training accuracy, and test accuracy versus epochs are plotted in [Figure 3: see original paper]-4 and [Figure 3: see original paper]-5.

[Figure 3: see original paper]-4: Results without feature processing
[Figure 3: see original paper]-5: Results with feature processing

Comparison reveals that feature processing significantly accelerates loss reduction and improves accuracy within limited epochs. The reason is that logarithmic transformation reduces data span (originally 0 to 255, where inputs could differ by hundreds of times), preventing potential gradient explosion and numerical instability. Additionally, neural networks without feature processing are highly sensitive to learning rates, a problem avoided by feature processing.

3.3 Batch Size Setting

Deep learning typically employs mini-batch stochastic gradient descent. To explore batch size effects, we set batch size to 6000, which closely approximates full gradient descent. The training results are shown in [Figure 3: see original paper]-6.

[Figure 3: see original paper]-6: Results with batch size 6000

Compared to [Figure 3: see original paper]-5, gradient descent consumed significantly more time while traversing all data five times, and loss reduction was much slower. Although gradient descent avoids curve oscillations, it lacks the generalization benefits brought by mini-batch gradient descent.

3.4 Optimization Algorithm Selection

Section 3.2 used standard mini-batch stochastic gradient descent. Here, we additionally implemented momentum and RMSProp algorithms to improve training effectiveness.

3.4.1 SGD (Mini-Batch Stochastic Gradient Descent) Let learning rate be α. In iteration t, SGD randomly selects a subset $\mathcal{S}_t$ containing K samples, computes the gradient of the loss function for each sample in this subset, averages them, and updates parameters [3]:

$$w_{t+1} = w_t - \frac{\alpha}{K} \sum_{i \in \mathcal{S}t} \nabla \ell_i(w_t)$$

To compare optimization algorithms in a relatively converged state, we trained for 10 epochs:

.2: SGD Training Results

Metric	Value
Training Accuracy	85.2%
Training Loss	0.42
Validation Accuracy	84.1%

[Figure 3: see original paper]-7: SGD Training Results

3.4.2 Momentum Method Momentum uses accumulated momentum from previous steps to replace the true gradient. The update formulas are:

$$v_t = \beta v_{t-1} + \eta \nabla \ell(w_t)$$

$$w_{t+1} = w_t - v_t$$

Thus, each parameter's actual update depends on the weighted average of recent gradients. When gradient directions are inconsistent recently, the true update magnitude decreases; when directions are consistent, the magnitude increases, providing acceleration. Generally, in early iterations, gradient directions are consistent, so momentum accelerates convergence to the optimum. In later iterations, gradient directions become inconsistent, causing oscillations near convergence, where momentum provides deceleration and increased stability.

With $\beta = 0.9$ and $\eta = 0.8 \times 10^{-3}$, training for 10 epochs yields:

.3: Momentum Training Results

Metric	Value
Training Loss	0.35
Training Accuracy	87.8%
Validation Accuracy	86.7%

[Figure 3: see original paper]-7: Momentum Training Results

3.4.3 RMSProp Algorithm RMSProp is an adaptive learning rate algorithm. Its weight update method is:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)(\nabla \ell(w_t))^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \nabla \ell(w_t)$$

With $\gamma = 0.9$ and $\eta = 8 \times 10^{-2}$, training for 10 epochs yields:

.4: RMSProp Training Results

Metric	Value
Training Loss	0.41
Training Accuracy	85.5%
Validation Accuracy	84.3%

[Figure 3: see original paper]-8: RMSProp Training Results

A comparison of loss reduction curves for the three optimization algorithms:

[Figure 3: see original paper]-9: Loss Reduction Curves for Three Optimization Algorithms

Overall results show that momentum not only accelerates convergence but also significantly improves final model performance. RMSProp performs similarly to SGD on this dataset. Therefore, subsequent research is based on the momentum method.

3.5 Weight Decay Setting

To improve generalization, we experimented with weight decay, which effectively adds an L2 norm to the loss function to prevent overfitting:

$$\ell'(w) = \ell(w) + \frac{\lambda}{2}\|w\|^2$$

The equivalent weight update formula (using mini-batch SGD) is:

$$w_{t+1} = (1 - \alpha\lambda)w_t - \alpha\nabla\ell(w_t)$$

The optimization algorithm decays weights at each training step. Compared to feature selection, weight decay provides a continuous mechanism to adjust function complexity. Smaller λ values correspond to less constrained w, while larger λ values constrain w more heavily.

Using momentum with $\lambda = 0.01$ and other hyperparameters identical to the momentum method, training for 10 epochs yields:

.5: Weight Decay Training Results

Metric	Value
Training Loss	0.36
Training Accuracy	87.5%
Validation Accuracy	86.9%

[Figure 3: see original paper]-9: Weight Decay Training Results

Comparing .3 and [Figure 3: see original paper]-10 shows slight alleviation of overfitting.

3.6 Dropout Method

Dropout is a simple yet effective regularization technique: at each training step, each neuron has probability p of being temporarily "dropped out," meaning it is completely ignored for that step but may be active in the next. After training, neurons are no longer dropped, as shown in [Figure 3: see original paper]-10 [6].

[Figure 3: see original paper]-10: Dropout Schematic Diagram

In our implemented MLP, perhaps due to fewer hidden layers, dropout effects were not significant. Setting dropout to 0.05 and 0.1 for the two hidden layers and training for 20 epochs showed minimal effect. Higher dropout rates affected convergence and decreased performance.

[Figure 3: see original paper]-11: Results without dropout
[Figure 3: see original paper]-12: Results with dropout

Chapter 4: Results Evaluation and Analysis

Based on Chapter 3, we selected as our improved model: a network with feature processing (log transformation), momentum optimization, two hidden layers (256 and 128 neurons), learning rate $\eta = 0.8 \times 10^{-3}$, and weight decay rate 0.01. This chapter provides further analysis and evaluation through additional metrics, comparison with traditional machine learning algorithms, and migration to MNIST to examine robustness.

4.1 Additional Evaluation Metrics

From Chapter 3, our model achieves 87.8% accuracy on the training set and 86.7% on the test set.

ROC curves are suitable for binary classification. Here, we plot ROC curves for class 0 vs. non-class 0 based on softmax layer outputs:

[Figure 4: see original paper]-1: ROC Curve for Improved Model (Class 0 vs. Non-Class 0)

The area under the ROC curve is substantial, indicating MLP achieves good classification performance.

4.2 Comparison with Softmax Regression

Softmax regression is a simple linear model without any hidden layers. Using the same optimization method and hyperparameters, the results are:

.1: Softmax Regression Training Results

Metric	Value
Training Accuracy	83.2%
Training Loss	0.48
Validation Accuracy	82.5%

[Figure 4: see original paper]-2: Softmax Regression Training Results

Comparing .5 and [Figure 3: see original paper]-9 shows MLP significantly outperforms softmax regression.

Similarly plotting ROC curves for class 0 vs. non-class 0:

[Figure 4: see original paper]-3: ROC Curve for Softmax Model (Class 0 vs. Non-Class 0)

Compared to [Figure 4: see original paper]-1, the area under the ROC curve is noticeably smaller, demonstrating inferior performance to the improved MLP.

4.3 Migration to MNIST Dataset

Using the same network structure and hyperparameters mentioned at the beginning of this chapter, but with dropout rates of 0.2 and 0.5 for the two hidden layers, the results on MNIST are:

.1: MNIST Training Results

Metric	Value
Training Loss	0.12
Training Accuracy	96.8%
Validation Accuracy	96.2%

[Figure 4: see original paper]-2: MNIST Training Results

As shown in [Figure 4: see original paper]-2, directly migrating our trained MLP model to MNIST achieves high accuracy. Further hyperparameter tuning and fine-tuning could yield even better results.

Chapter 5: Conclusion

This paper designed and implemented a Multi-Layer Perceptron. Through experiments comparing log-transformed features, momentum optimization, and weight decay regularization, model performance was improved, achieving 87.8% training accuracy and 86.7% test accuracy on Fashion MNIST. The model was directly migrated to MNIST, where it also achieved high accuracy.

References

[1] 刘天舒. BP 神经网络的改进研究及应用 [D]. 东北农业大学, 2011.
[2] Aston Zhang, Zachary C. Lipton, Mu Li. 动手深度学习 [M]. 人民邮电出版社, 2019.
[3] 邱锡鹏. 神经网络与深度学习 [M]. 机械工业出版社, 2020.
[4] 杨海涛. 高等数学. 下册. 第 3 版 [M]. 同济大学出版社, 2013.
[5] 龚禹. Implementing Multiple Layer Neural Network from Scratch[EB/OL]. dennybritz/nn-from-scratch: Implementing a Neural Network from Scratch (github.com)
[6] Aurelien Geron. 机器学习实战：基于 Scikit-Learn、Keras 和 Tensorflow（原书第二版）[M]. 机械工业出版社, 2020.

Note: Figure translations are in progress. See original paper for figures.