An exploration of the impact of model network complexity on the performance of perceptrons in tasks

Hengyi Ma
a1875198
The University Of Adelaide

Abstract

In this task, we need to use the dataset Pima Indians Diabetes Database for diabetes prediction. This dataset has eight physiological indicators such as age, blood pressure, and BMI, and a label indicating whether you have diabetes. The task of this dataset is to predict whether a patient has diabetes. Therefore, taking patients as samples, our prediction results will focus on predicting whether the sample "has" or "does not have" diabetes, which is a binary classification problem.

The perceptron [5] is the cornerstone of support vector machines and neural networks. Its goal is to obtain a separation hyperplane that can completely separate the positive instance points and negative instance points in the dataset, thereby achieving the classification of positive and negative indicators.

In this task, I will try to use the perceptron model to learn eight features of the dataset, so that the model can establish the correspondence between features and results, and let the model use new feature data to predict whether new samples have diabetes. Furthermore, I will change the complexity of the perceptron by modifying the model structure, observe the changes in the loss function and accuracy of the model under different complexity, and analyze the reasons.

The code of this report can be accessed at: `https://github.com/HengyiMa/deeplearning-demo/blob/main/%E6%84%9F%E7%9F%A5%E6%9C%BA%E9%A2%84%E6%B5%8B%E5%88%9D%E6%AD%A5%E5%B0%9D%E8%AF%95/using%20perceptron%20to%20prediect%20diabetes.ipynb`

1. Introduction

The perceptron was proposed by Rosenblatt in 1957. It is a linear classification model. In the earliest version, the perceptron accepts input from a neuron, multiplies the input and weight, sums and adds with the bias, uses a step function that acts as an activation function for classification and updates the parameters based on the difference between the output and the actual label. The basic structure of the earliest perceptron is shown in the figure[8]:

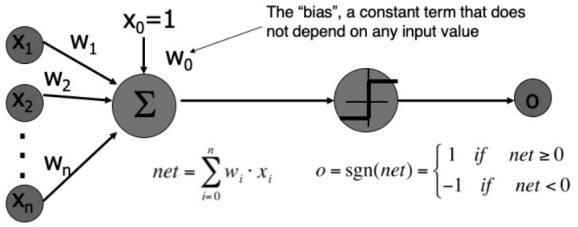

Figure 1. A structure diagram of Perceptron

The perceptron [1] is used to solve the binary classification problem. Since there is only one linear layer in the middle, the perceptron tries to find a linear hyperplane to separate positive and negative samples. Specifically, the perceptron determines the sample category through a step function, and the process of the step function determining the sample input can be reflected as: $f(x) = \text{sign}(w \cdot x + b)$

Through the processing of activation functions, the perceptron attempts to find a hyperplane: $\text{sign}(w \cdot x + b) = 0$

This hyperplane can completely separate positive and negative samples. The image of the step function and the process of the perceptron finding the hyperplane in the task are as follows:

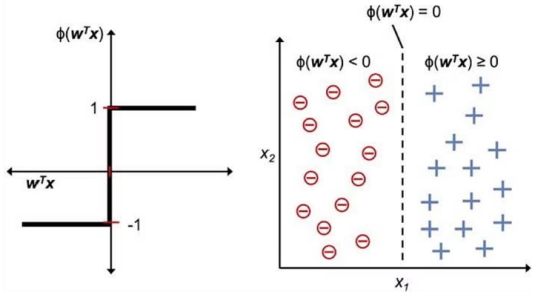

Figure 2. The image of the symbolic function and the hyperplane of the perceptron

2. Related work

Pytorch architecture: My model is actually based on the pytorch framework. Pytorch is one of the popular deep learning frameworks today and is especially suitable for training neural network models such as perceptrons. Pytorch has an automatic derivation function, eliminating the need for manual design. In the design of neural network models, this architecture is simple to design and has good readability. At the same time, this architecture has a lot of machine learning related content, such as optimizers like Adam and SGD.

SGD(stochastic gradient descent): My models all use SGD [4] as the optimizer. SGD is a variant of the gradient descent algorithm used to optimize machine learning models. In SGD, each iteration does not use the entire dataset, but only selects a single random training sample (or a small batch) to calculate the gradient and update the model parameters. This random selection introduces randomness into the optimization process, making it possible for SGD to jump out of the current local minimum and find a new local minimum, giving it a chance to find a better solution.

BCELoss function: The models all use the BCEloss [3] function as the loss function. The BCEloss function is a special form of the cross-entropy function in binary classification problems. The cross-entropy function as a loss function is particularly suitable for use in classification problems. Because when a neural network performs a classification problem, the normalized output result using sigmoid/softmax is actually a logits, and the cross-entropy function can measure the difference between the predicted logits and the actual logits, intuitively showing the difference between the predicted value and the output value.

Back propagation: The model uses the back propagation algorithm [6], which is a revolutionary algorithm in the field of neural networks. By calculating the gradient of the loss function on the model parameters, the parameters could be updated, so that the model can be gradually optimized to minimize the loss function. The key to the backpropagation algorithm is the use of the chain rule, which allows the gradient to be propagated from the loss function back to each layer of the neural network in order to calculate the gradient of the parameters of each layer, thereby enabling complex network training and parameter updates. Correspondingly, since the step function is non-differentiable, backpropagation cannot be used, and the final output layer is replaced with a sigmoid function for processing. The sigmoid function, like the softmax function, can normalize the output to a probability of 0-1 and output the logits of the classification problem. The image of the sigmoid function is as shown in the figure:

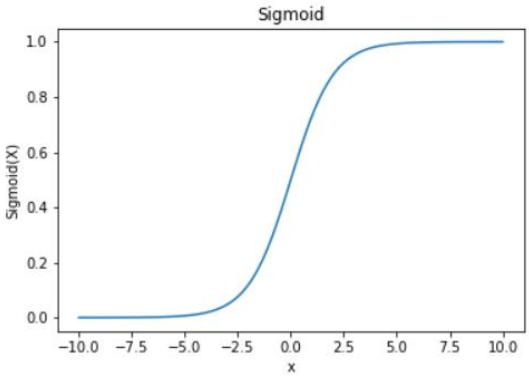

Figure 3. Sigmoid function

3. Proposed method

I divided the data into a training set, a validation set, and a test set, and used the same random seeds during the division process to ensure stable experimental results.

Modify the complexity of the perceptron model in the following ways:

1. Introduce more neurons and increase the width of the network (the neural network calculates the results through the weighted summation between neurons, increasing the number of neurons in an attempt to enable the network to capture more complex features and patterns)

2. Enhance the depth of the model and introduce hidden layers (adding hidden layers in a neural network helps neurons learn different levels of feature representations. Generally speaking, lower-level neurons can easily learn simple features, and higher-level neurons can easily learn complex features.)

3. When introducing the hidden layer, introduce an activation function [7]. Different from the activation function used in the final output layer, using the activation function in the hidden layer can introduce nonlinearity in the calculation process. If you only look at the simplest perceptron, you will find that only linear mapping is used, and the formula can be expressed as:

$$Y = f\left(\sum_i w_i x_i + b\right)$$

Introducing some common nonlinear functions as the activation function of the hidden layer, such as sigmoid and relu, can change the problem of model calculation with only linear mapping, and perform nonlinear transformation of the original linear gradient through mapping. This has the following advantages:

These activation functions have various shapes and can increase the complexity of the original linear mapping to varying degrees according to task requirements.

Different activation functions can be selected for data of

different scales and shapes to ensure that the calculation of backpropagation is the most effective.

Activation functions allow neural networks to deepen the model because stacking of linear mappings is meaningless and stacking of nonlinear mappings can attempt to fit any function computation.

In my model, the hidden layers use the relu function. The relu function can alleviate the vanishing gradient problem that may occur like the sigmoid function (the gradient is close to 0 at both ends of the sigmoid function, which may cause the backpropagation calculation gradient to be too small, thereby ineffective) , the image of the relu function is as follows:

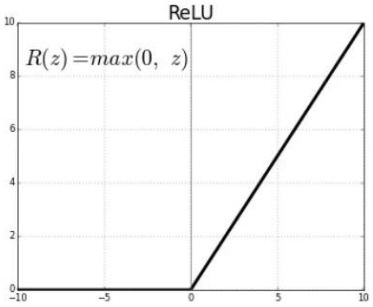

Figure 4. Relu function

In order to further avoid the problem of gradient disappearance or gradient explosion in the model, we add a batch normalization layer [2] before the activation function of each hidden layer. Batch normalization is a method used by the model to standardize the input. It accelerates the convergence of the network by normalizing the input of each feature to a standard normal distribution with a mean of zero and a variance of one. This can ensure that the gradient is in a smaller range, so that it is less likely to produce very extreme results when passing through the activation function, and alleviates the gradient problem of the model.

The purpose of the experiment is to explore how the perceptron performs on tasks under models of different complexity, using accuracy and loss curves as our evaluation criteria. In order to ensure the fairness of different models, the following processing is performed:

1. All models use the same loss function BCEloss, the same optimizer SGD and learning rate 0.005. The learning rate is small and is not prone to oscillations and does not easily lead to large differences in different models.

2. We don't need to keep the epochs of different models completely consistent, but stop when the model performs better.

To determine whether the model has reached a relatively good level, we start from the following three criteria:

1. Whether the loss curve continues to decline, and whether the decline speed is large

2. Whether the accuracy of the training set has declined or no longer improved.

3. On the premise that the accuracy of the training set no longer improves, will the performance of the test set no longer improve? Because if the accuracy no longer improves but the loss still decreases, it indicates that the model may still be able to learn the data but cannot be reflected in the training set, thus results in improvement in generalization ability.

Accordingly, we experimented with four models in total:

The simplest single-layer perceptron uses only one linear layer and a sigmoid normalized result.

The three-layer model includes three linear layers. Each hidden layer contains 4 neurons. Relu is introduced as the activation function. A batch normalization layer is introduced before each activation function for normalization.

The six-layer model includes six linear layers. Each hidden layer contains 4 neurons. Relu is introduced as the activation function. A batch normalization layer is introduced before each activation function for normalization.

The eleven-layer model includes eleven linear layers. Relu is introduced in the hidden layer as the activation function. A batch normalization layer is introduced before each activation function for normalization. At the same time, the number of neurons in the first four layers is increased to 128 128, 64, 32 to comprehensively improve the complexity of the model.

4. Experiment analysis

The results of accuracy of 4 different models could be shown as a chart:

Method \ Accuracy	Training Set	Validation Set	Test Set
single-layer perceptron	64.39%	68.13%	69.08%
three-layer perceptron	55.92%	75.82%	57.24%
six-layer perceptron	63.92%	65.38%	69.74%
eleven-layer perceptron	80.19%	60.99%	52.63%

Table 1. Results of perceptrons with different complexity

In the single-layer perceptron, I ran 2000 epochs. The loss curves of the training set and the test set dropped at a smoother slope, and the accuracy of the two tended to stabilize after about 250 epochs. The final accuracy comparison is: 64.39% for the training set, 68.13% for the validation set, and 69.08% for the test set. As shown in the figure:

It can be seen that the performance of the three single-layer perceptrons is relatively close. The accuracy of the test set and validation set is slightly higher than that of the training set. I think it has something to do with the small sample size, which leads to accidental data prediction. Different results will be obtained by replacing different random seeds.

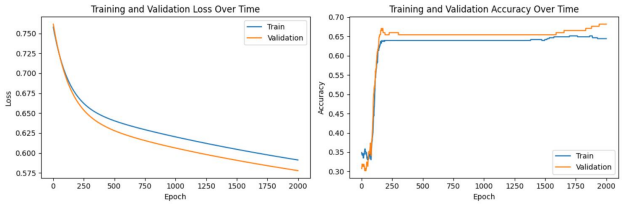

Figure 5. Experiment result of single-layer perceptron

In the three-layer perceptron, I ran 2000 epochs. The loss curves of the training set and the test set dropped at a smoother slope, and the accuracy of the two stabilized after about 1100 epochs. The final accuracy comparison is: 55.92% for the training set, 75.82% for the validation set, and 57.24% for the test set, as shown in the figure:

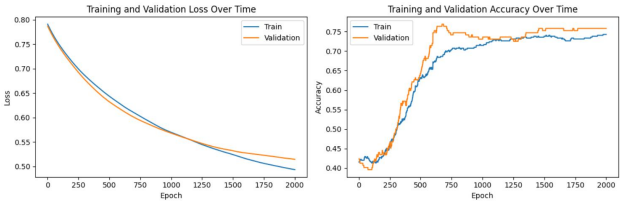

Figure 6. Experiment result of three-layer perceptron

The accuracy of the validation set is much higher than that of the training set. I think it has something to do with the small sample size, which leads to accidental data testing. Different results will be obtained by replacing different random seeds. The accuracy of the training set and the test set is close, but both are smaller than the performance of the single-layer perceptron, which is a surprising finding. I think there are several possible reasons:

1. The data set is too small, resulting in too strong randomness in training and testing.

2. The task is very simple. Features can be effectively learned using a single-layer perceptron. A more complex model may lead to poor performance.

In the six-layer perceptron, I ran 800 epochs. The loss curves of the training set and the test set dropped at a smoother slope, and the accuracy of the two stabilized after about 500 epochs. The final accuracy comparison is: 63.92% for the training set, 65.38% for the validation set, and 69.74% for the test set, as shown in the figure:

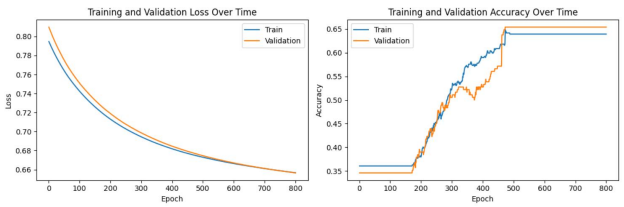

Figure 7. Experiment result of six-layer perceptron

This result is slightly better than the one-layer perceptron on test set and better than the three-layer perceptron, and the experimental performance is very stable on the training set, validation set and test set. By introducing nonlinear factors and strengthening the model depth to improve the learning ability of the model, at least in this task, the performance of the six-layer perceptron on the test set is slightly better than that of the one-layer perceptron. The experimental results support this view.

At the same time, the experimental results refute hypothesis 2, that is, for this task, increasing model complexity is effective, at least not completely counterproductive. But at the same time, a new hypothesis 3 is proposed: Shallow models may have strong generalization capabilities due to their simple decision boundaries, and complex models can effectively fit complex features and make good predictions. Models in between may lack both, resulting in poorer performance than simpler or more complex models.

In the eleven-layer perceptron with an increased number of neurons, I ran it for 400 epochs. The loss curve of the training set dropped at a smoother slope, and the loss curve of the validation set quickly tended to be flat or even rose slightly. The difference between the two The accuracy stabilizes after about 100 epochs. The final accuracy comparison is: 80.19% for the training set, 60.99% for the validation set, and 52.63% for the test set, as shown in the figure:

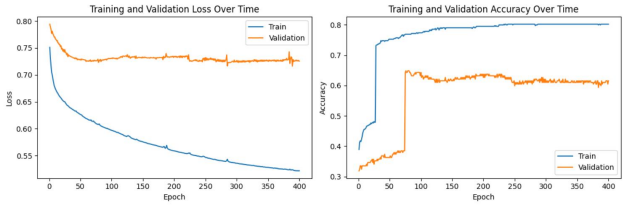

Figure 8. Experiment result of eleven-layer perceptron

It can be seen that this result is different from any previous results. The accuracy of the training set is much higher than the validation set and test set. At the same time, the test set performance is the worst among several models, indicating that after the complexity of the model is greatly improved, the generalization Ability has declined. This shows that the model has learned too many details, resulting in poor prediction ability. It does not mean that simply increasing the complexity of the model will make the model perform better. But at the same time, it can be found that the convergence speed of this model is much faster than the previous three models, which shows that in this task, a more complex model to a certain extent can learn features more quickly.

5. Conclusion

By designing perceptron models of different complexity and testing them on the same dataset, I compared the performance of models of different complexity and tried to analyze the reasons. Based on four models of different complexity, I came to the following conclusions that may be helpful in designing perceptron and other similar neural network models:

1. The size of the data set is very important to the model, which is reflected in two different aspects: for tasks with different number of features, we should use models of different complexity. For tasks with simple features, using a simple model may have good results, but a complex model may not be superior. The size of the sample directly affects the stability of the data and the division of the data set. For tasks with small sample sizes, model design must be very careful, because the model will be very sensitive to sample division.

2. It is effective to increase the complexity of the model to a certain extent, such as enhancing the depth of the model and increasing the number of model neurons. At the same time, if the complexity is increased based on a linear model, nonlinear factors must be introduced, otherwise it will still be a linear mapping and will not change the complexity of the model.

3. Blindly enhancing model complexity may not necessarily produce better results. Overly simple enhancements and overly complex enhancements may have no effect. Simply increasing the model complexity may result in the model not learning more complex features, while excessively increasing the complexity may cause the model to learn a large amount of details of the training dataset, resulting in poor generalization ability.

4. The convergence speed of model learning is related to the complexity of the model. Under certain circumstances, a more complex model may learn the characteristics of the dataset faster.

The future experiments can continue on different tasks and different datasets to test whether these conclusions are generalizable.

References

[1] M Banoula. What is perceptron: A beginners guide for perceptron. *dirección: https://www. simplilearn. com/tutorials/deep-learning-tutorial/perceptron*, 2022.

[2] Ketan Doshi. Batch norm explained visually—how it works, and why neural networks need it, 2022.

[3] Daniel Godoy. Understanding binary cross-entropy/log loss: a visual explanation. *towards data science*, 21, 2018.

[4] Nikhil Ketkar and Nikhil Ketkar. Stochastic gradient descent. *Deep learning with Python: A hands-on introduction*, pages 113–132, 2017.

[5] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[6] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[7] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316, 2017.

[8] Andreas Wichert and Luis Sacouto. *Machine Learning — A Journey to Deep Learning: with Exercises and Answers*. 02 2021.