Chapter 9 Overfitting

Everything should be made as simple as possible, but not simpler.

—Albert Einstein

The main purpose of machine learning is to learn the real model of the data from the training set, so that it can perform well on the unseen test set. We call this the generalization ability. Generally speaking, the training set and the test set are sampled from the same data distribution. The sampled samples are independent of each other, but come from the same distribution. We call this assumption the Independent Identical Distribution Assumption (i.i.d.).

The expressive power of the model has been mentioned earlier, also known as the capacity of the model. When the model's expressive power is weak, such as a single linear layer, it can only learn a linear model and cannot approximate the nonlinear model well. When the model's expressive power is too strong, it may be possible to reduce the noise modalities of the training set, but leads to poor performance on the test set (generalization ability is weak). Therefore, for different tasks, designing a model with appropriate capacity can achieve better generalization performance.

9.1 Model capacity

In layman's terms, the capacity or expressive capacity of a model refers to the model's ability to fit complex functions. An indicator reflecting the capacity of the model is the size of the hypothesis space of the model, that is, the size of the set of functions that the model can represent. The larger and more complete the hypothesis space, the more likely it is to search from the hypothesis space for a function that approximates the real model. Conversely, if the hypothesis space is very limited, it is difficult to find a function that approximates the real model.

Consider sampling from real distribution

$$p_{\text{data}} = \{(x, y)|y = \sin(x), x \in [-5,5]\}$$

A small number of points are sampled from the real distribution to form the training set, which contains the observation error ϵ, as shown by the small dots in Figure 9.1. If we only search the model space of all first-degree polynomials and set the bias to 0, that is $y = ax$, as shown by the straight line of the first-degree polynomial in Figure 9.1. Then it is difficult to find a straight line that closely approximates the distribution of real data. Slightly increase the hypothesis space so that the hypothesis space is all 3rd degree polynomial functions, that is $y = ax^3 + bx^2 + cx$, it is obvious that this hypothesis space is obviously larger than the hypothesis space of the 1st degree polynomial, we can find a curve (as shown in Figure 9.1) that reflects the relationship of the data better than the first-order polynomial model , but it is still not good enough. Increase the hypothesis space again so that the searchable function is a polynomial of degree 5, that is $y = ax^5 + bx^4 + cx^3 + dx^2 + ex$. In this hypothesis space, a better function can be searched, as shown by the polynomial of degree 5 in Figure 9.1. After increasing the hypothesis space again, as

shown in the polynomial curves of 7, 9, 11, 13, 15, 17 in Figure 9.1, the larger the hypothesis space of the function, the more likely it is to find a function model that better approximates the real distribution.

However, an excessively large hypothesis space will undoubtedly increase the search difficulty and computational cost. In fact, under the constraints of limited computing resources, a larger hypothesis space may not necessarily be able to search for a better model. Due to the existence of observation errors, a larger hypothesis space may contain a larger number of functions with too strong expression ability, which can also learn the observation errors of the training samples, thus hurting the generalization ability of the model. Choosing the right model capacity is a difficult problem.

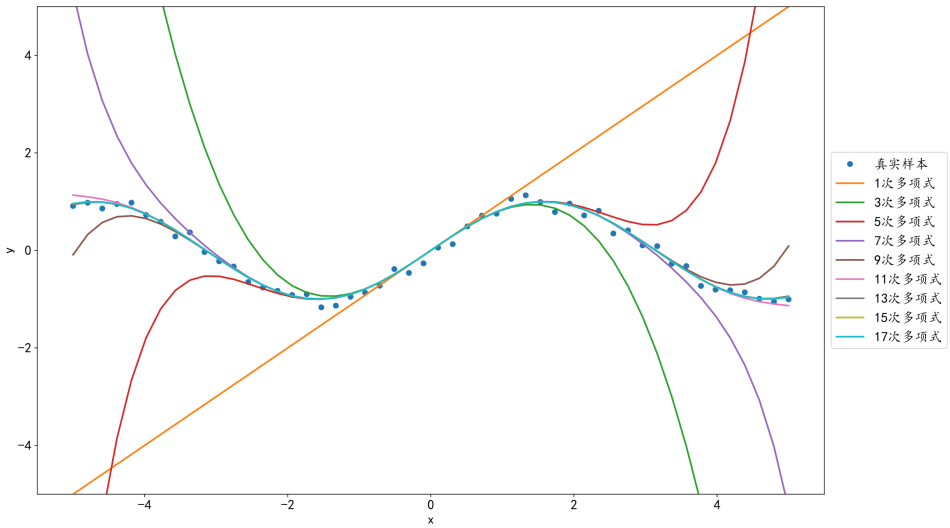


Figure 9.1 Polynomial capability

9.2 Overfitting and underfitting

Because the distribution of real data is often unknown and complicated, it is impossible to deduce the type of distribution function and related parameters. Therefore, when choosing the capacity of the learning model, people often choose a slightly larger model capacity based on empirical values. However, when the capacity of the model is too large, it may appear to perform better on the training set, but perform worse on the test set, as shown in the area to the right of the red vertical line in Figure 9.2. When the capacity of the model is too small, it may have poor performance in both the training set and the testing set as shown in the area to the left of the red vertical line in Figure 9.2.

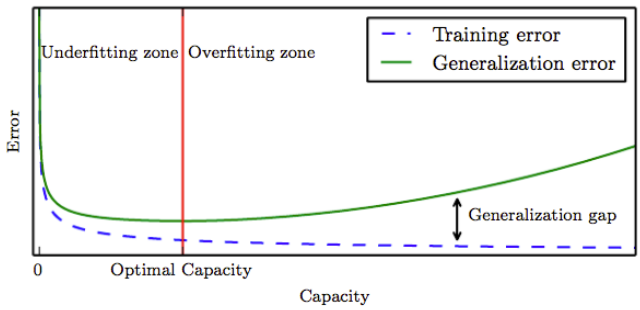


Figure 9.2 The relation between model capacity and error [1]

When the capacity of the model is too large, in addition to learning the modalities of the training set data, the network model also learns additional observation errors, resulting in the learned model performing better on the training set, but poor in unseen samples, that is, the generalization ability of the model is weak. We call this phenomenon overfitting. When the capacity of the model is too small, the model cannot learn the modalities of the training set data well, resulting in poor performance on both the training set and the unseen samples. We call this phenomenon underfitting.

Here is a simple example to explain the relationship between the model's capacity and the data distribution. Figure 9.3 plots the distribution of certain data. It can be roughly speculated that the data may belong to a certain degree 2 polynomial distribution. If we use a simple linear function to learn, we will find it difficult to learn a better function, resulting in the underfitting phenomenon that the training set and the test set do not perform well, as shown in Figure 9.3 (a). However, if you use a more complex function model to learn, it is possible that the learned function will excessively "fit" the training set samples, but resulting in poor performance on the test set, i.e. overfitting, as shown in Figure 9.3 (c). Only when the capacity of the learned model and the real model roughly match, the model can have a good generalization ability, as shown in Figure 9.3 (b).

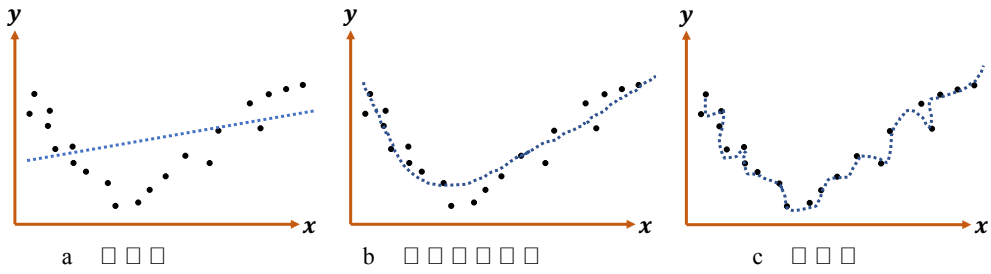


Figure 9.3 Overfitting and underfitting

Consider the distribution p_{data} of data points (x, y), where

$$y = \sin(1.2 \cdot \pi \cdot x)$$

During sampling, random Gaussian noise is added to obtain a data set of 120 points, as shown in Figure 9.4. The curve in the figure is the real model function, the black round points are the training samples, and the green matrix points are the test samples.

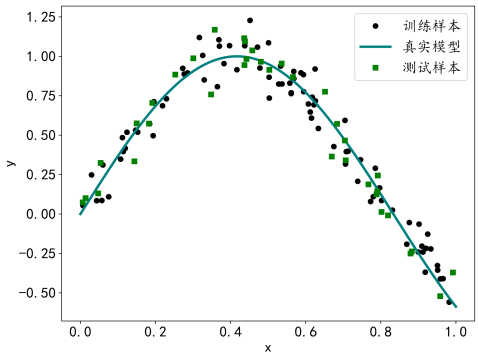


Figure 9.4 Dataset and the real function

In the case of a known real model, it is natural to design a function space with appropriate capacity to obtain a good learning model. As shown in Figure 9.5, we assume that the model is a second-degree polynomial model, and the learned function curve is approximating the real model. However, in actual scenarios, the real model is often unknown, so if the design hypothesis space is too small, it will be impossible to search for a suitable learning model. If the design hypothesis space is too large, it will result in poor model generalization ability.

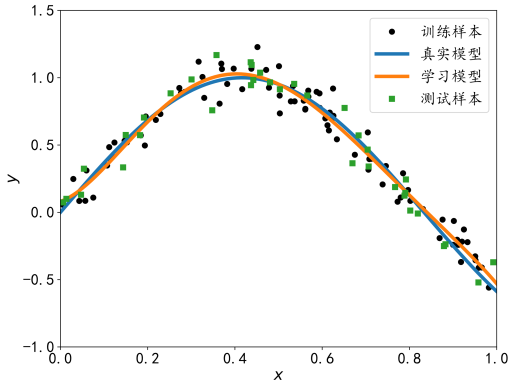


Figure 9.5 Appropriate model capability

So how to choose the capacity of the model? Statistical learning theory provides us with some ideas. The VC dimension (Vapnik-Chervonenkis dimension) is a widely used method to measure the capacity of functions. Although these methods provide a certain degree of theoretical guarantee for machine learning, these methods are rarely applied to deep learning. Part of the reason is that the neural network is too complicated to determine the VC dimension of the mathematical model behind the network structure.

Although statistical learning theory is difficult to give the minimum capacity required by a neural network, it can be used to guide the design and training of a neural network based on Occam ’s razor. The Occam's razor principle was a rule of solution proposed by William of Occam, a 14th-century logician and Franciscan monk of the Franciscans. He stated in his book that “Don’t waste more things and do things that you can do well with less.” In other words, if the two-layer neural network structure can express the real model well, then the three-layer neural network can also express well, but we should prefer to use the simpler two-layer neural network because its parameters amount is smaller, it is easier to train, and it is easier to get a good

generalization error through fewer training samples.

9.2.1 Underfitting

Let us consider the phenomenon of underfitting. As shown in Figure 9.6, black dots and green rectangles are independently sampled from the distribution of a parabolic function. Because we already know the real model, if we use a linear function with lower capacity than the real model to fit the data, it is difficult for the model to perform well. The specific performance is that the learned linear model has a larger error (such as the mean square error) on the training set, and the error on the test set is also larger.

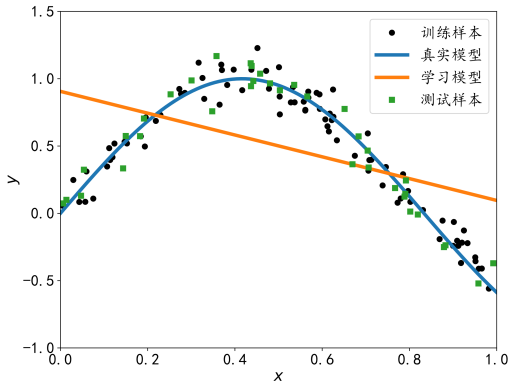


Figure 9.6 A typical underfitting model

When we find that the current model has maintained a high error on the training set, it is difficult to optimize and reduce the error, and it also performs poorly on the test set, we can consider whether there is a phenomenon of underfitting. The problem of underfitting can be solved by increasing the number of layers of the neural network or increasing the size of the intermediate dimension. However, because modern deep neural network models can easily reach deeper layers, the capacity of the model used for learning is generally sufficient. In real applications, more overfitting phenomena occur.

9.2.2 Overfitting

Consider the same problem, the black dots of the training set and the green rectangles of the test machine are independently sampled from a parabolic model with the same distribution. When we set the hypothesis space of the model to 25th polynomial, it is much larger than the functional capacity of the real model. It is found that the learned model is likely to overfit the training sample, resulting in the error of the learning model on the training sample is very small, even smaller than the error of the real model on the training set. But for the test sample, the model performance drops sharply, and the generalization ability is very poor, as shown in Figure 9.7.

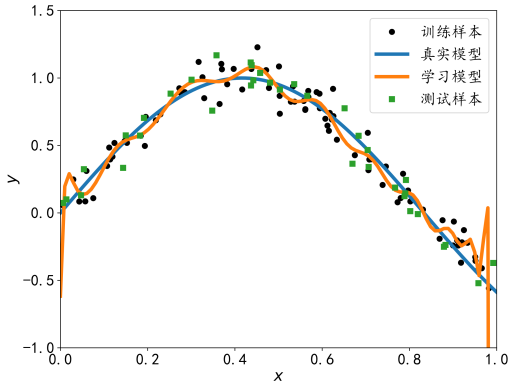


Figure 9.7 A typical overfitting model

The phenomenon of overfitting in modern deep neural networks is very easy to occur, mainly because the neural network has a very strong expressive ability and the number of samples in the training set is not enough, it is easy to appear that the capacity of the neural network is too large. So how to effectively detect and reduce overfitting?

Next, we will introduce a series of methods to help detect and suppress overfitting.

9.3 Data set division

Earlier we introduced that the data set needs to be divided into a training set and a test set. In order to select model hyperparameters and detect overfitting, it is generally necessary to split the original training set into a new training set and a validation set, that is, the data set needs to be divided into three subsets: training set, validation set and test set.

9.3.1 Validation set and hyperparameters

The difference between the training set and the test set has been introduced earlier. The training set $\mathbb{D}^{\text{train}}$ is used to train model parameters, and the test set $\mathbb{D}^{\text{test}}$ is used to test the generalization ability of the model. The samples in the test set cannot participate in the model training, preventing the model from "memorizing" the characteristics of the data and damaging the generalization ability of the model. Both the training set and the test set are sampled from the same data distribution. For example, the MNIST handwritten digital picture set has a total of 70,000 sample pictures, of which 60,000 pictures are used as the training set, and the remaining 10,000 pictures are used for the test set. The seperation ratio of the training set and the test set can be defined by the user. For example, 80% of the data is used for training, and the remaining 20% is used for testing. When the size of the data set is small, in order to test the generalization ability of the model more accurately, the proportion of the test set can be increased appropriately. Figure 9.8 below demonstrates the division of the MNIST handwritten digital picture collection: 80% is used for training, and the remaining 20% is used for testing.

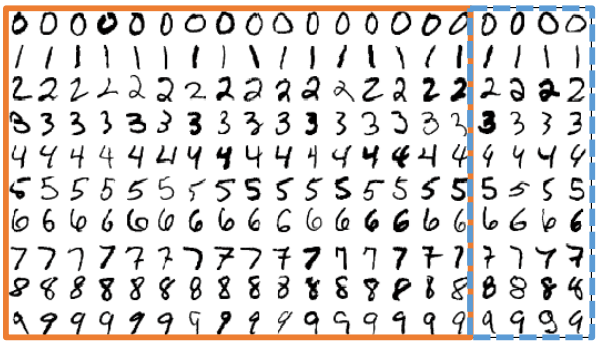


Figure 9.8 Training and testing dataset division

However, it is not enough to divide the data set into only the training set and the test set. Because the performance of the test set cannot be used as feedback for model training, we need to be able to pick out more suitable model hyperparameters during model training to determine whether the model is overfitting. Therefore, we need to divide the training set into training set and validation set, as shown in Figure 9.9. The divided training set has the same function as the original training set and is used to train the parameters of the model, while the validation set is used to select the hyperparameters of the model. Its functions include:

- Adjust the learning rate, weight decay coefficient, training times, etc. according to the performance of the validation set.
- Readjusts the network topology according to the performance of the validation set.
- According to the performance of the validation set, determine whether it is overfitting or underfitting.

Similar to the division of the training set-test set, the training set, validation set and test set can be divided according to a custom ratio, such as the common 60% -20% -20% division. Figure 9.9 shows the MNIST handwriting data set Schematic diagram of the division.

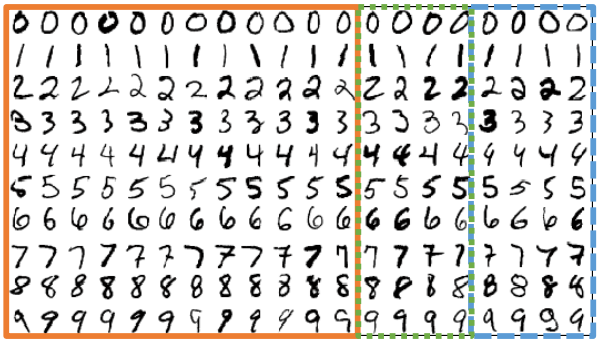


Figure 9.9 Training, validation and test dataset

The difference between the validation set and the test set is that the algorithm designer can adjust the settings of various hyperparameters of the model according to the performance of the validation set to improve the generalization ability of the model, but the performance of the test set cannot be used to adjust the model. Otherwise, the functions of the test set and the validation set will overlap, so the performance on the test set will not represent the generalization ability of the model.

In fact, some developers will incorrectly use the test set to select the best model, and then use it as a model generalization performance report. For those cases, the test set is actually the validation set, so the "generalization performance" reported is essentially the performance on the

validation set, not the real generalization performance. In order to prevent this kind of “cheating”, you can choose to generate multiple test sets, so that even if the developer uses one of the test sets to select the model, we can also use other test sets to evaluate the model, which is also commonly used in Kaggle competitions.

9.3.2 Early stopping

Generally, we call one Batch updating in the training set one Step, and iterating through all the samples in the training set once is called an Epoch. The validation set can be used after several Steps or Epochs to calculate the validation performance of the model. If the validation steps are too frequent, it can accurately observe the training status of the model, but it also introduces additional computation costs. It is generally recommended to perform a validation operation after several Epochs.

Taking the classification task as an example, the training performance indicators include training error, training accuracy, etc. Correspondingly, there are also validation error and validation accuracy during validation process, and test error and test accuracy during testing process. The training accuracy and validation accuracy can roughly infer whether the model is overfitting and underfitting. If the training error of the model is low and the training accuracy is high, but the validation error is high and the validation accuracy rate is low, overfitting may occur. If the errors on both the training set and the validation set are high and the accuracy is low, then underfitting may occur.

When overfitting is observed, the capacity of the network model can be redesigned, such as reducing the number of layers of the network, reducing the number of parameters of the network, adding regularization methods, adding constraints on the hypothesis space, etc., so that the actual capacity of the model reduces to solve the overfitting phenomenon. When the underfitting phenomenon is observed, you can try to increase the capacity of the network, such as deepening the number of layers of the network, increasing the number of network parameters, and trying more complicated network structures.

In fact, since the actual capacity of the network can change as the training progresses, even with the same network settings, different overfitting and underfitting conditions may be observed. Figure 9.10 shows a typical training curve for classification problems. The red curve is the training accuracy and the blue curve is the test accuracy. As we can see from the figure, as the training progresses in the early stage of training, the training accuracy and test accuracy of the model show an increasing trend, and there is no overfitting phenomenon at this time. In the later stage of training, even with the same network structure, due to the change in the actual capacity of the model, we observed the phenomenon of overfitting. That is, the training accuracy continues to improve, but the generalization ability becomes weaker (the test accuracy decreases).

This means that for neural networks, even if the network hyperparameters amount remains unchanged (that is, the maximum capacity of the network is fixed), the model may still appear to be overfitting, because the effective capacity of the neural network is closely related to the state of the network parameters. The effective capacity of the neural network can be very large, and the effective capacity can also be reduced by means of sparse parameters and regularization. In the early and middle stages of training, the phenomenon of overfitting did not appear. As the number

of training Epochs increased, the overfitting became more and more serious. In Figure 9.10, the vertical dotted line is in the best state of the network, there is no obvious overfitting phenomenon, and the generalization ability of the network is the best.

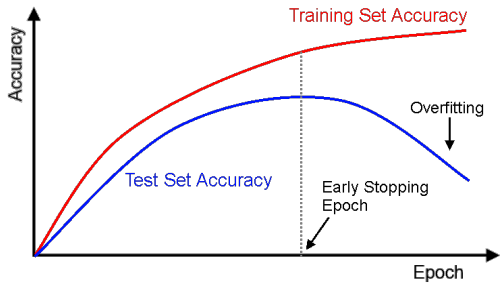


Figure 9.10 Training process diagram

So how to choose the right Epoch to stop training early (Early Stopping) to avoid overfitting? We can predict the possible position of the most suitable Epoch by observing the change of the validation metric. Specifically, for the classification problem, we can record the validation accuracy of the model and monitor its change. When it is found that the validation accuracy has not decreased for successive Epochs, we can predict that the most suitable Epoch may have been reached, so we can stop training. Figure 9.11 plots the curve of training and validation accuracy with training Epoch during a specific training process. It can be observed that when Epoch is around 30, the model reaches its optimal state and we can stop training in advance.

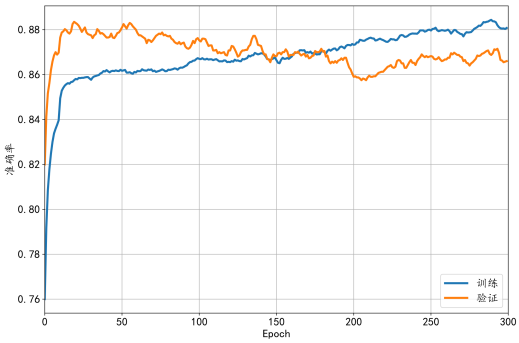


Figure 9.11 Traning curve example

Algorithm 1 is a pseudo-code that uses an early-stop model training algorithm.

Algorithm 1：Network training with Early stopping

Initialize parameter θ

repeat

  for $step = 1, \ldots, N$ **do**

    random select Batch $\{(\boldsymbol{x}, y)\} \sim \mathbb{D}^{\text{train}}$

    $\theta \leftarrow \theta - \eta \nabla_{\theta} \mathcal{L}(f(\boldsymbol{x}), y)$

  end

---

if every nth Epoch **do**

Calculate validation set $\{(\boldsymbol{x}, y)\}\sim\mathbb{D}^{\text{val}}$ performance

if validation performance doesn't increase for certain successive steps **do**

save the network and stop training

end

do

until training reaches maximum Epoch

Use the saved network to calculate test set $\{(\boldsymbol{x}, y)\}\sim\mathbb{D}^{\text{test}}$ performance

Output: Network parameter θ and testing accuracy

9.4 Model design

The validation set can determine whether the network model is overfitting or underfitting, which provides a basis for adjusting the capacity of the network model. For neural networks, the number of layers and parameters of the network are very important reference indicators for network capacity. By reducing the number of layers and reducing the size of the network parameters in each layer, the network capacity can be effectively reduced. Conversely, if the model is found to be underfitting, we can increase the capacity of the network by increasing the number of layers and the amount of parameters in each layer.

To demonstrate the effect of the number of network layers on network capacity, we visualized the decision boundary of a classification task. Figure 9.12, Figure 9.13, Figure 9.14, and Figure 9.15 respectively demonstrate the decision boundary map for training 2-category classification task under different network layers, where the red rectangular block and the blue circular block respectively represent the 2 types of samples on the training set. Only adjust the number of layers of the network while keeping other hyperparameters consistent. As shown in the figure, you can see that as the number of network layers increases, the learned model decision boundary is more and more close to training samples, indicating overfitting. For this task, the two-layer neural network can obtain good generalization ability. The deeper layer of the network does not improve the overall model performance. Instead, it can lead to overfitting, and the generalization ability becomes worse, and the computation cost is also higher.

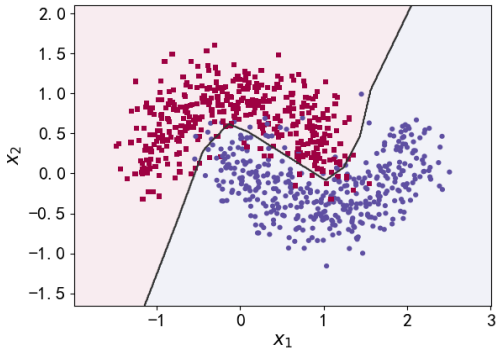


Figure 9.12 2 layers

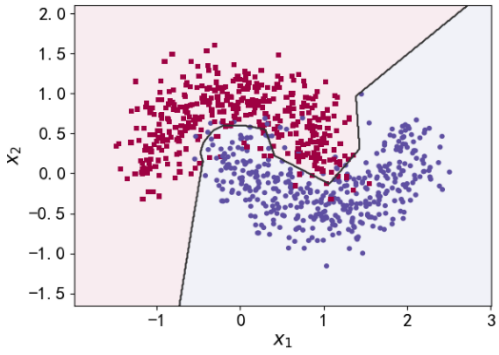


Figure 9.13 3 layers

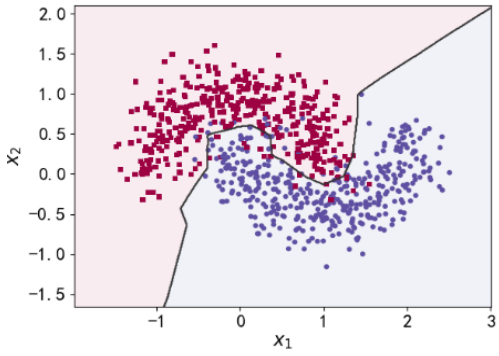


Figure 9.14 4 layers

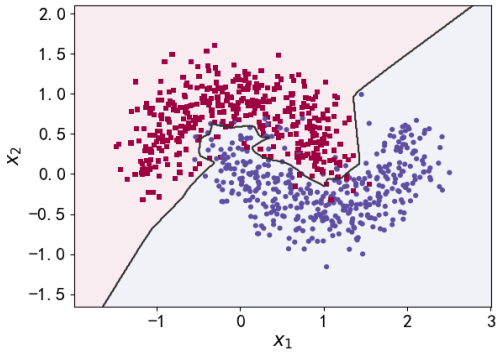


Figure 9.15 6 layers

9.5 Regularization

By designing network models with different layers and sizes, the initial function hypothesis space can be provided for the optimization algorithm, but the actual capacity of the model can change as the network parameters are optimized and updated. Take the polynomial function model as an example:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \cdots + \beta_n x^n + \varepsilon$$

The capacity of the above model can be simply measured through n. During the training process, if the network parameters $\beta_{k+1}, \cdots, \beta_n$ are all 0, then the actual capacity of the network degenerates to the function capacity of the kth polynomial. Therefore, by limiting the sparsity of network parameters, the actual capacity of the network can be constrained.

This constraint is generally achieved by adding additional parameter sparsity penalties to the loss function. The optimization goal before the constraint added is

$$min\ \mathcal{L}(f_\theta(\boldsymbol{x}), y), (\boldsymbol{x}, y) \in \mathbb{D}^{\text{train}}$$

After adding additional constraints to the parameters of the model, the goal of optimization becomes

$$min\ \mathcal{L}(f_\theta(\boldsymbol{x}), y) + \lambda \cdot \Omega(\theta), (\boldsymbol{x}, y) \in \mathbb{D}^{\text{train}}$$

Where $\Omega(\theta)$ represents the sparsity constraint function on the network parameters θ. Generally, the sparsity constraint of the parameter θ is achieved by constraining the L norm of the parameter, i.e.

$$\Omega(\theta) = \sum_{\theta_i} \|\theta_i\|_l$$

where $\|\theta_i\|_l$ represents the l norm of the parameter θ_i.

In addition to minimizing the original loss function $\mathcal{L}(\boldsymbol{x}, y)$, the new optimization goal also needs to constrain the sparsity $\Omega(\theta)$ of the network parameters. The optimization algorithm will reduce the network parameter sparsity $\Omega(\theta)$ as much as possible while reducing $\mathcal{L}(\boldsymbol{x}, y)$. Here λ is the weight parameter to balance the importance of $\mathcal{L}(\boldsymbol{x}, y)$ and $\Omega(\theta)$. Larger λ means that the sparsity of the network is more important; smaller λ means that the training error of the network is more important. By selecting the appropriate λ, you can get better training performance, while ensuring the sparsity of the network, which lead to a good generalization ability.

Commonly used regularization methods are L0, L1, and L2 regularization.

9.5.1 L0 regularization

L0 regularization refers to the regularization calculation method using the L0 norm as the sparsity penalty term $\Omega(\theta)$, namely

$$\Omega(\theta) = \sum_{\theta_i} \|\theta_i\|_0$$

The L0 norm $\|\theta_i\|_0$ is defined as the number of non-zero elements in θ_i. The constraint of $\sum_{\theta_i} \|\theta_i\|_0$ can force the connection weights in the network to be mostly 0, thereby reducing the actual amount of network parameters and network capacity. However, because the L0 norm is not derivable, gradient descent algorithm cannot be used for optimization. L0 norm is not often used in neural networks.

9.5.2 L1 regularization

The regularization calculation method using the L1 norm as the sparsity penalty term $\Omega(\theta)$ is called L1 regularization, that is

$$\Omega(\theta) = \sum_{\theta_i} \|\theta_i\|_1$$

The L1 norm $\|\theta_i\|_1$ is defined as the sum of the absolute values of all elements in the tensor θ_i. L1 regularization is also called Lasso Regularization, which is continuously derivable and widely used in neural networks.

L1 regularization can be implemented as follows:

```
# Create weights w1,w2
w1 = tf.random.normal([4,3])
w2 = tf.random.normal([4,2])
# Calculate L1 regularization term
loss_reg = tf.reduce_sum(tf.math.abs(w1))\
```

```
    + tf.reduce_sum(tf.math.abs(w2))
```

9.5.3 L2 regularization

The regularization calculation method using the L2 norm as the sparsity penalty term $\Omega(\theta)$ is called L2 regularization, that is

$$\Omega(\theta) = \sum_{\theta_i} \|\theta_i\|_2$$

The L2 norm $\|\theta_i\|_2$ is defined as the sum of squares of all elements in the tensor θ_i. L2 regularization is also called Ridge Regularization, which is continuous and derivable like L1 regularization, and is widely used in neural networks.

The L2 regularization term is implemented as follows:

```
# Create weights w1,w2
w1 = tf.random.normal([4,3])
w2 = tf.random.normal([4,2])
# Calculate L2 regularization term
loss_reg = tf.reduce_sum(tf.square(w1))\
    + tf.reduce_sum(tf.square(w2))
```

9.5.4 Regularization effect

Continue to take the crescent-shaped 2-class data as an example. Under the condition that the other hyperparameters such as the network structure remain unchanged, the L2 regularization term is added to the loss function, and different regularization hyperparameter λ are used to obtain regularization effects of different degrees.

After training 500 Epochs, we obtain the classification decision boundaries of the learning model, as shown in Figure 9.16, Figure 9.17, Figure 9.18, and Figure 9.19. The distribution represents the classification effect when the regularization coefficient $\lambda = 0.00001, 0.001, 0.1, and\ 0.13$ is used. It can be seen that as the regularization coefficient increases, the network penalties for parameter sparsity become larger, thus forcing the optimization algorithm to search for models that make the network capacity smaller. When $\lambda = 0.00001$, the regularization effect is relatively weak, and the network is overfitting. However, when 在$\lambda = 0.1$, the network has been optimized to the appropriate capacity, and there is no obvious overfitting or underfitting.

In actual training, it is generally preferred to try smaller regularization coefficients to observe whether the network is overfitting. Then try to increase the parameter λ gradually to increase the sparsity of the network parameters and improve the generalization ability. However, excessively large λ may cause the network to not converge, and need to be adjusted according to the actual task.

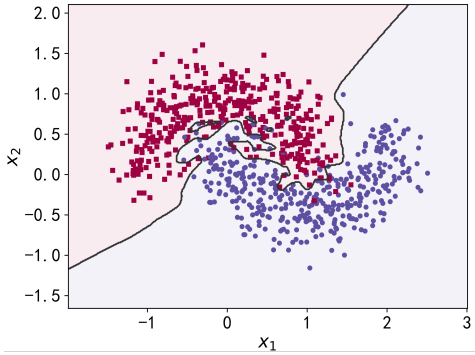


Figure 9.16 Regularization parameter：0.00001

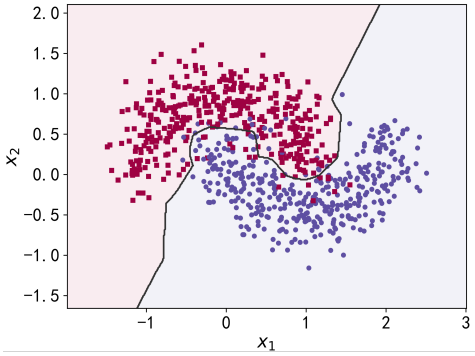


Figure 9.17 Regularization parameter：0.001

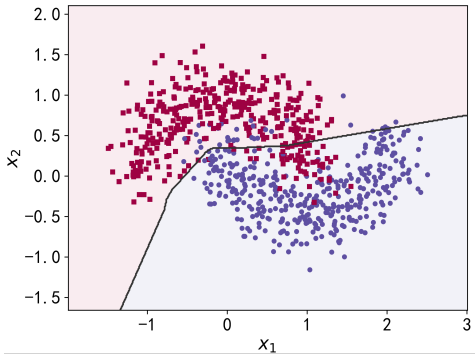


Figure 9.18 Regularization parameter：0.1

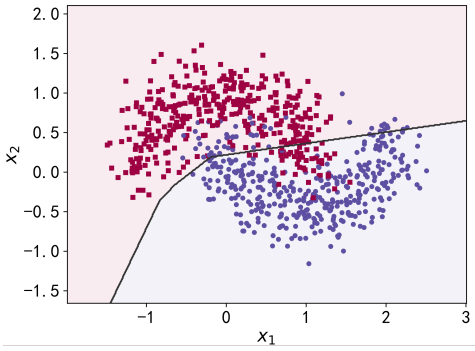


Figure 9.19 Regularization parameter：0.13

Under different regularization coefficients, we counted the value range of each connection weight in the network. Consider the weight matrix $\boldsymbol{W}$ of 2nd layer of the network, whose shape is [256,256], that is, to convert a vector with an input length of 256 to an output vector of 256. From the perspective of the weight connection of the fully connected layer, the weight $\boldsymbol{W}$ include $256 \cdot 256$ connection lines. We correspond them to the XY grids in Figure 9.20, Figure 9.21, Figure 9.22, and Figure 9.23, where the X axis range is $[0,255]$ and the range of the Y axis is [0,255]. All integer points of the XY grid respectively represent each position of the weight tensor $\boldsymbol{W}$ of shape $[256,256]$, and each grid point indicates the weight of the current connection. From the figure, we can see the influence of different degrees of regularization constraints on the network weights. When $\lambda = 0.00001$, the effect of regularization is relatively weak, and the weight values in the network are relatively large, and are mainly distributed in interval $[-1.6088,1.1599]$. After increasing the value to $\lambda = 0.13$, the network weight values are constrained in a smaller range $[-0.1104,0.0785]$. As shown in 9.1, the sparseness of the weights after regularization can also be observed.

Table 9.1 Weight variation after regularization

$\boldsymbol{\lambda}$	$\boldsymbol{min(W)}$	$\boldsymbol{max(W)}$	$\boldsymbol{mean(W)}$
0.00001	-1.6088	1.1599	0.0026
0.001	-0.1393	0.3168	0.0003

0.1	-0.0969	0.0832	0
0.13	-0.1104	0.0785	0

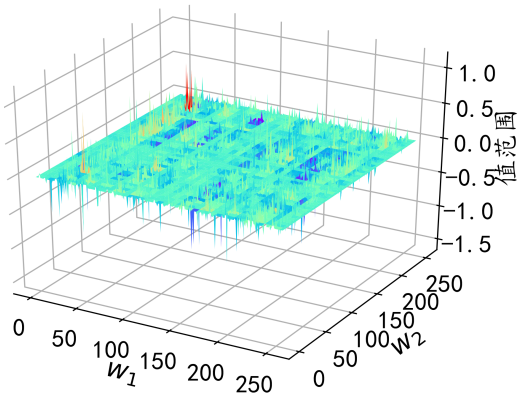


Figure 9.20 Regularization parameter：0.00001

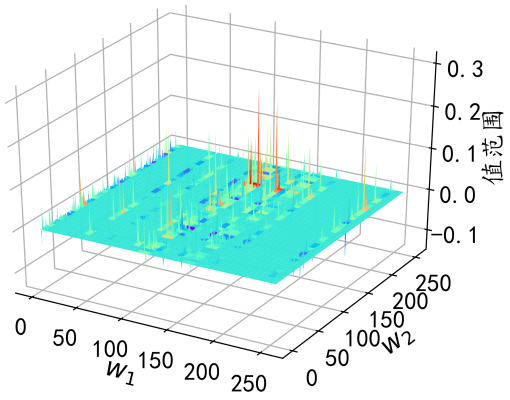


Figure 9.21 Regularization parameter：0.001

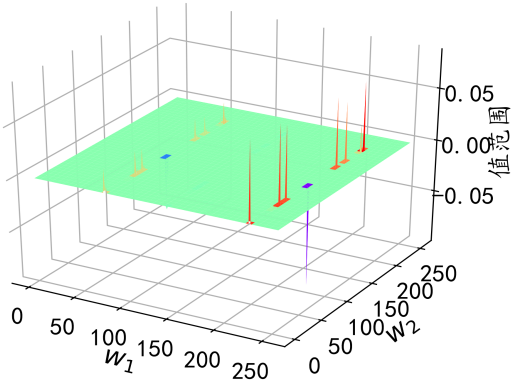


Figure 9.22 Regularization parameter：0.1

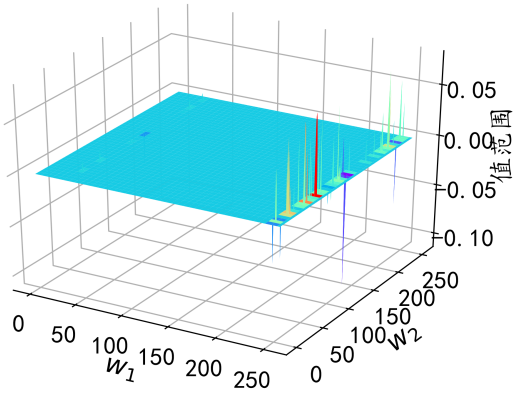


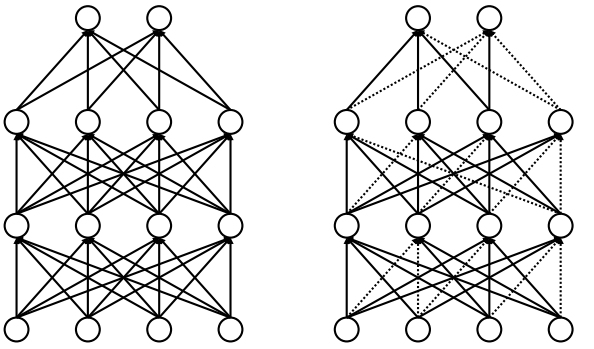
Figure 9.23 Regularization parameter：0.13

9.6 Dropout

In 2012, Hinton et al. used the Dropout method in their paper "Improving neural networks by preventing co-adaptation of feature detectors" to improve model performance. Dropout method reduces the number of parameters of the model that actually participates in the calculation during each training by randomly disconnecting the neural network. However, during testing, Dropout method will restore all connections to ensure the best performance during model testing.

Figure 9.24 is a schematic diagram of the connection status of a fully connected layer network during a certain forward calculation. Figure 9.24(a) is a standard fully connected neural network. The current node is connected to all input nodes in the previous layer. In the network layer to which the Dropout function is added, as shown in Figure 9.24(b), whether each connection is disconnected conforms to a certain preset probability distribution, such as a Bernoulli distribution with a disconnect probability Figure 9.24(b) shows a specific sampling result. The dotted line indicates that the sampling result is a disconnected line, and the solid line

indicates the sampling result is not disconnected.



(a) □ □ □ □ □ □ □ □ (b) □ Dropout □ □ □ □ □ □

Figure 9.24 Dropout diagram

In TensorFlow, you can implement the Dropout function through the tf.nn.dropout(x, rate) function, where the rate parameter sets the probability p of disconnection. E.g:

```
# Add dropout operation with disconnection rate of 0.5
x = tf.nn.dropout(x, rate=0.5)
```

You can also use Dropout as a network layer and insert a Dropout layer in the middle of the network. E.g:

```
# Add Dropout layer with disconnection rate of 0.5
model.add(layers.Dropout(rate=0.5))
```

In order to explore the influence of the Dropout layer on network training, we maintained the hyperparameters such as the number of network layers unchanged, and observed the impact of Dropout on network training by inserting different numbers of Dropout layers in the 5 fully connected layers. As shown in Figure 9.25, Figure 9.26, Figure 9.27, and Figure 9.28, the distribution draws the decision boundary effect of the network model without adding Dropout layers, adding 1, 2, and 4 Dropout layers. It can be seen that when the Dropout layer is not added, the network model has the same result as the previous observation. With the increase of the Dropout layer, the actual capacity of the network model during training decreases and the generalization ability becomes stronger.

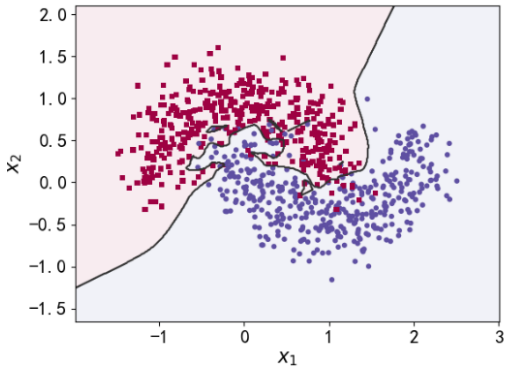


Figure 9.25 Without Dropout layer

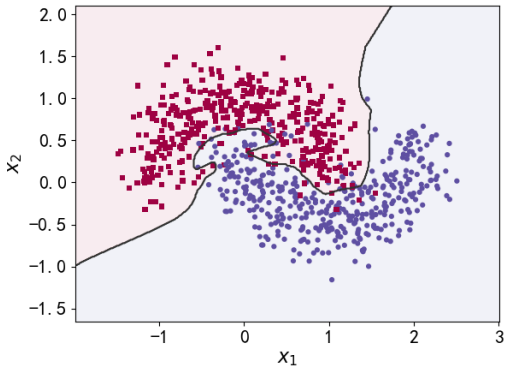


Figure 9.26 With 1 Dropout layer

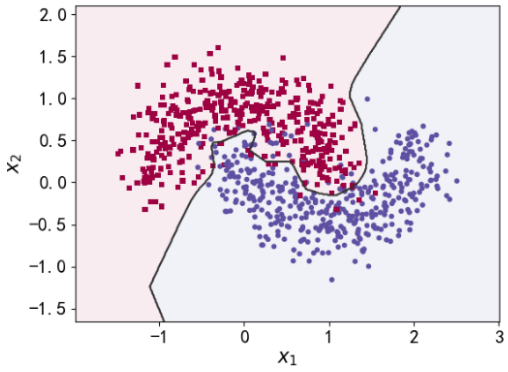


Figure 9.27 With 2 Dropout layer

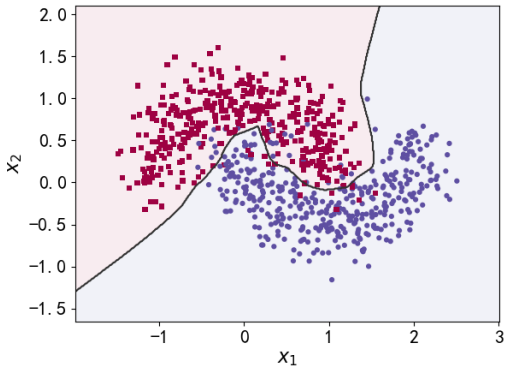


Figure 9.28 With 4 Dropout layer

9.7 Data augmentation

In addition to the methods described above, which can effectively detect and suppress overfitting, increasing the size of the data set is the most important way to solve overfitting problem. However, collecting sample data and labels is often costly. For a limited data set, the number of training samples can be increased through data augmentation technology to obtain a certain degree of performance improvement. Data augmentation refers to changing the characteristics of the sample based on a priori knowledge while keeping the sample label unchanged, so that the newly generated sample also conforms or approximately conforms to the true distribution of the data.

Taking image data as an example, let's introduce how to do data augmentation. The size of the pictures in the data set is often inconsistent. In order to facilitate the processing of the neural network, the pictures need to be rescaled to a fixed size, as shown in Figure 9.29, which is a fixed size 224×224 picture after rescaling. For the person in the picture, according to a priori knowledge, we know that rotation, scaling, translation, cropping, changing the angle of view, and blocking a certain local area will not change the main category label of the picture, so for the picture data, there are a variety of data augmentation methods.

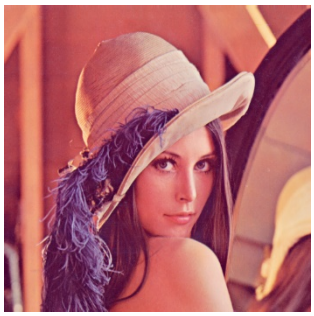


Figure 9.29 A picture after rescaling to 224×224 pixels

TensorFlow provides common image processing functions, located in the tf.image submodule. Through the tf.image.resize function, we can zoom the pictures. We generally implement data augmentation in the preprocessing step. After reading the picture from the file system, the image data augmentation operation can be performed. E.g:

```
def preprocess(x,y):
    # Preprocess function
    # x: picture path, y: picture label
    x = tf.io.read_file(x)
    x = tf.image.decode_jpeg(x, channels=3) # RGBA
    # rescale pictures to 244x244
    x = tf.image.resize(x, [244, 244])
```

9.7.1 Rotation

Rotating pictures is a very common way of augmenting picture data. By rotating the original picture at a certain angle, new pictures at different angles can be obtained, and the label information of these pictures remains unchanged, as shown in Figure 9.30.

Figure 9.30 Image rotation

Through tf.image.rot90(x, k = 1), the picture can be rotated by 90 degrees counterclockwise k times, for example:

```
# Picture rotates 180 degrees counterclockwise
x = tf.image.rot90(x,2)
```

9.7.2 Flip

The flip of the picture is divided into flip along the horizontal axis and along the vertical axis, as shown in Figure 9.31 and Figure 9.32 respectively. In TensorFlow, you can use tf.image.random_flip_left_right and tf.image.random_flip_up_down to randomly flip the image in the horizontal and vertical directions, for example:

```
# Random horizontal flip
x = tf.image.random_flip_left_right(x)
# Random vertical flip
x = tf.image.random_flip_up_down(x)
```

Figure 9.31 Horizontal flip

Figure 9.32 Vertical flip

9.7.3 Cropping

By removing part of the edge pixels in the left, right, or up and down directions of the original image, the main body of the image can be kept unchanged, and new image samples can be obtained at the same time. When actually cropping, the picture is generally scaled to a size slightly larger than the network input size, and then cropped to a suitable size. For example, if the input size of the network is 224×224, then you can use the resize function to rescale the picture to 244×244, and then randomly crop to the size 224×224. The code is implemented as follows:

```
# Rescale picture to larger siz
x = tf.image.resize(x, [244, 244])
# Then randomly crop the picture to the desired size
x = tf.image.random_crop(x, [224,224,3])
```

Figure 9.33 is a picture zoomed to 244×244, Figure 9.34 is an example of random cropping to 244×244, and Figure 9.35 is also an example of random cropping.

Figure 9.33 Before cropping

Figure 9.34 After cropping and rescaling-1

Figure 9.35 After cropping and rescaling -2

9.7.4 Generate data

By training the generative model on the original data and learning the distribution of the real data, the generative model can be used to obtain new samples. This method can also improve network performance to a certain extent. For example, conditional generation adversarial network (Conditional GAN, CGAN for short) can generate labeled sample data, as shown in Figure 9.36.

Figure 9.36 CGAN generated numbers

9.7.5 Other methods

In addition to the typical picture data augmentation methods described above, the picture data can be arbitrarily transformed to obtain new pictures based on a priori knowledge without changing the picture tag information. Figure 9.37 demonstrates the picture data after superimposing Gaussian noise on the original picture, Figure 9.38 demonstrates the new picture obtained by changing the viewing angle of the picture, and Figure 9.39 demonstrates the new picture obtained by randomly blocking parts of the original picture.

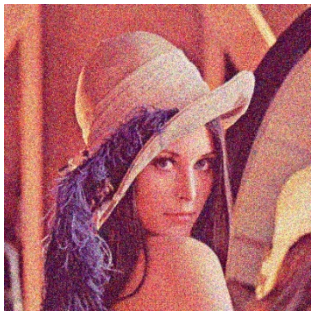


Figure 9.37 Adding Gaussian noise

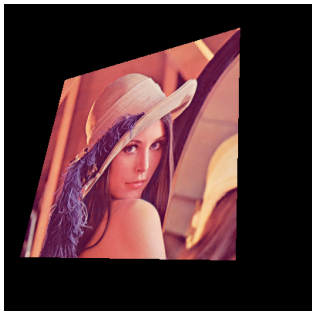


Figure 9.38 Changing viewing angle

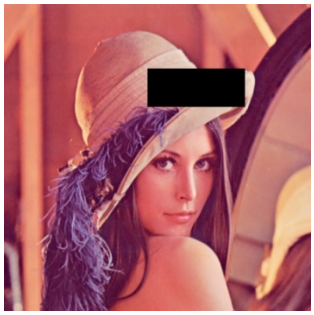


Figure 9.39 Randonly blocking parts

9.8 Hands-on overfitting

Earlier, we used a large amount of crescent-shaped 2-class data sets to demonstrate the performance of the network model under various measures to prevent overfitting. In this section, we will complete the exercise based on the overfitting and underfitting models of the 2 classification data set of crescent shape.

9.8.1 Build the dataset

The feature vector length of the sample data set we used is 2, and the label is 0 or 1, which represents two categories. With the help of the make_moons tool provided in the scikit-learn library, we can generate a training set of any number of data. First open the cmd command terminal and install the scikit-learn library. The command is as follows:

```
# Install scikit-learn library
pip install -U scikit-learn
```

To demonstrate the phenomenon of overfitting, we only sampled 1000 samples, and added Gaussian noise with a standard deviation of 0.25 as below:

```
# Import libraries
from sklearn.datasets import make_moons
# Randomly choose 1000 samples, and split them into training and testing sets
X, y = make_moons(n_samples = N_SAMPLES, noise=0.25, random_state=100)
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                test_size = TEST_SIZE, random_state=42)
```

The make_plot function can easily draw the distribution map of the data according to the coordinate X of the sample and the label y of the sample:

```
def make_plot(X, y, plot_name, file_name, XX=None, YY=None, preds=None):
    plt.figure()
    # sns.set_style("whitegrid")
    axes = plt.gca()
    axes.set_xlim([x_min,x_max])
    axes.set_ylim([y_min,y_max])
```

```
    axes.set(xlabel="$x_1$", ylabel="$x_2$")
    # Plot prediction surface
    if(XX is not None and YY is not None and preds is not None):
        plt.contourf(XX, YY, preds.reshape(XX.shape), 25, alpha = 0.08,
                     cmap=cm.Spectral)
        plt.contour(XX, YY, preds.reshape(XX.shape), levels=[.5],
cmap="Greys",
                    vmin=0, vmax=.6)
    # Plot samples
    markers = ['o' if i == 1 else 's' for i in y.ravel()]
    mscatter(X[:, 0], X[:, 1], c=y.ravel(), s=20,
             cmap=plt.cm.Spectral, edgecolors='none', m=markers)
    # Save the figure
    plt.savefig(OUTPUT_DIR+'/'+file_name)
```

Draw the distribution of 1000 samples for sampling, as shown in Figure 9.40, the red square points are one category, and the blue circles are another category.

```
# Plot data points
make_plot(X, y, None, "dataset.svg")
```

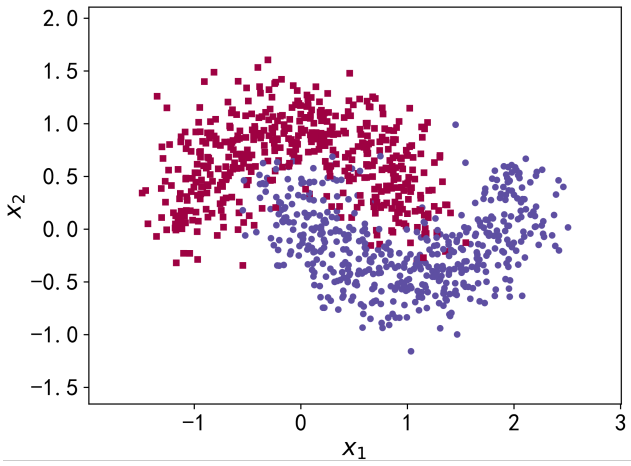


Figure 9.40 Moon-shape two-class data points

9.8.2 Influence of the number of network layers

In order to explore the degree of overfitting at different network depths, we conducted a total of 5 training experiments. When $n \in [0,4]$, build a fully connected layer network with $n+2$ layers, and train 500 Epochs through the Adam optimizer to obtain the separation curve of the network on the training set, as shown in Figures 9.12, 9.13, 9.14, and 9.15 .

```
for n in range(5): # Create 5 different network with different layers
    model = Sequential()
    # Create 1st layer
```

```
    model.add(Dense(8, input_dim=2,activation='relu'))
    for _ in range(n): # Add nth layer
        model.add(Dense(32, activation='relu'))
    model.add(Dense(1, activation='sigmoid')) # Add last layer
    model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy']) # Configure and train
    history = model.fit(X_train, y_train, epochs=N_EPOCHS, verbose=1)
    # Plot boundaries for different network
    preds = model.predict_classes(np.c_[XX.ravel(), YY.ravel()])
    title = "Network layer ({})".format(n)
    file = "NetworkCapacity%f.png"%(2+n*1)
    make_plot(X_train, y_train, title, file, XX, YY, preds)
```

9.8.3 Impact of Dropout

In order to explore the impact of the Dropout layer on network training, we conducted a total of 5 experiments. Each experiment used a 7-layer fully connected layer network for training, but inserted 0 ~ 4 Dropout layers in the fully connected layer at intervals and passed Adam The optimizer trains 500 Epochs. The network training results are shown in Figures 9.25, 9.26, 9.27, and 9.28.

```
for n in range(5): # Create 5 different networks with differen number of
Dropout layers
    model = Sequential()
    # Create 1st layer
    model.add(Dense(8, input_dim=2,activation='relu'))
    counter = 0
    for _ in range(5): # Total number of layers is 5
        model.add(Dense(64, activation='relu'))
        if counter < n: # Add n Dropout layers
            counter += 1
            model.add(layers.Dropout(rate=0.5))
    model.add(Dense(1, activation='sigmoid')) # Output layer
    model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy']) # Configure and train
    # Train
    history = model.fit(X_train, y_train, epochs=N_EPOCHS, verbose=1)
    # Plot decision boundaries for different number of Dropout layers
    preds = model.predict_classes(np.c_[XX.ravel(), YY.ravel()])
    title = "Dropout({})".format(n)
    file = "Dropout%f.png"%(n)
    make_plot(X_train, y_train, title, file, XX, YY, preds)
```

9.8.4 Impact of regularization

In order to explore the influence of regularization coefficients on network model training, we adopted the L2 regularization method to construct a 5-layer neural network, in which the weight tensor W of the second, third, and fourth neural network layers are added with L2 regularization constraints terms as follows:

```
def build_model_with_regularization(_lambda):
    # Create networks with regularization terms
    model = Sequential()
    model.add(Dense(8, input_dim=2,activation='relu')) # without
regularization
    model.add(Dense(256, activation='relu', # With L2 regularization
                kernel_regularizer=regularizers.l2(_lambda)))
    model.add(Dense(256, activation='relu', # With L2 regularization
                kernel_regularizer=regularizers.l2(_lambda)))
    model.add(Dense(256, activation='relu', # With L2 regularization
                kernel_regularizer=regularizers.l2(_lambda)))
    # Output
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy']) # Configure and train
    return model
```

Under the condition of keeping the network structure unchanged, we adjust the regularization coefficient $\lambda = 0.00001, 0.001, 0.1, 0.12, 0.13$ to test the training effect of the network and draw the decision boundary curve of the learning model on the training set, as shown in Figure 9.16, Figure 9.17, Figure 9.18, Figure 9.19 Show.

```
for _lambda in [1e-5,1e-3,1e-1,0.12,0.13]:
    # Create model with regularization term
    model = build_model_with_regularization(_lambda)
    # Train model
    history = model.fit(X_train, y_train, epochs=N_EPOCHS, verbose=1)
    # Plot weight range
    layer_index = 2
    plot_title = "Regularization-[lambda = {}]".format(str(_lambda))
    file_name = " Regularization _" + str(_lambda)
    # Plot weight ranges
    plot_weights_matrix(model, layer_index, plot_title, file_name)
    # Plot decision boundaries
    preds = model.predict_classes(np.c_[XX.ravel(), YY.ravel()])
    title = " regularization ".format(_lambda)
    file = " regularization %f.svg"%_lambda
    make_plot(X_train, y_train, title, file, XX, YY, preds)
```

The plot_weights_matrix code of the matrix 3D plot function is as follows:

```
def plot_weights_matrix(model, layer_index, plot_name, file_name):
    # Plot weight ranges
    # Get weights for certain layers
    weights = model.layers[LAYER_INDEX].get_weights()[0]
    # Get minimum, maximum and mean values
    min_val = round(weights.min(), 4)
    max_val = round(weights.max(), 4)
    mean_val = round(weights.mean(), 4)
    shape = weights.shape
    # Generate grids
    X = np.array(range(shape[1]))
    Y = np.array(range(shape[0]))
    X, Y = np.meshgrid(X, Y)
    print(file_name, min_val, max_val,mean_val)
    # Plot 3D figures
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.xaxis.set_pane_color((1.0, 1.0, 1.0, 0.0))
    ax.yaxis.set_pane_color((1.0, 1.0, 1.0, 0.0))
    ax.zaxis.set_pane_color((1.0, 1.0, 1.0, 0.0))
    # Plot weight ranges
    surf = ax.plot_surface(X, Y, weights, cmap=plt.get_cmap('rainbow'), line
width=0)
    ax.set_xlabel('x', fontsize=16, rotation = 0)
    ax.set_ylabel('y', fontsize=16, rotation = 0)
    ax.set_zlabel('weight', fontsize=16, rotation = 90)
    # save figure
    plt.savefig("./" + OUTPUT_DIR + "/" + file_name + ".svg")
```

9.9 References

[1] I. Goodfellow, Y. Bengio and A. Courville, Deep Learning, MIT Press, 2016.