Chapter 2 Regression

Some people worry that artificial intelligence will make us feel inferior, but then, anybody in his right mind should have an inferiority complex every time he looks at a flower.

– Alan Kay

2.1 Neuron model

An adult brain contains about 100 billion neurons. Each neuron obtains input signals through dendrites and transmits output signals through axons. The neurons are interconnected to form a huge neural network, thus forming the human brain, the basis of perception and consciousness. Figure 0.1 is a typical biological neuron structure. In 1943, the psychologist Warren McCulloch and mathematical logician Walter Pitts proposed a mathematical model of artificial neural networks to simulate the mechanism of biological neurons [1]. This research was further developed by the American neurologist Frank Rosenblatt into the Perceptron model [2], which is also the cornerstone of modern deep learning.

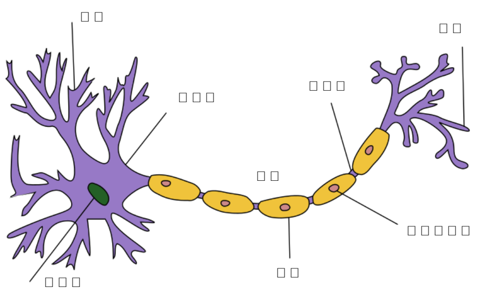

Figure 0.1 Typical biological neuron structure[①]

Starting from the structure of biological neurons, we will revisit the exploration of scientific pioneers and gradually unveil the mystery of automatic learning machines.

First, we can abstract the Neuron model into the mathematical structure as shown in Figure 0.2 (a). The neuron input vector $\boldsymbol{x} = [x_1,\ x_2,\ x_3, \dots, x_n]^\mathrm{T}$ maps to y through function $f_\theta: \boldsymbol{x} \to y$, where θ represents the parameters in the function f. Consider a simplified case such as linear transformation: $f(\boldsymbol{x}) = \boldsymbol{w}^\mathrm{T}\boldsymbol{x} + b$, the expended form is：

$$f(\boldsymbol{x}) = w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b$$

The above calculation logic can be intuitively shown in Figure 0.2 (b).

[①] Source:https://commons.wikimedia.org/wiki/File:Neuron_Hand-tuned.svg

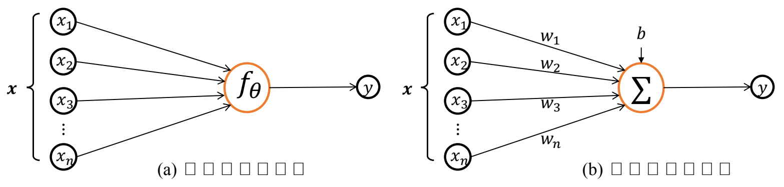

Figure 0.2 Mathematical neuron model

The parameters $\theta = \{w_1, w_2, w_3, \ldots, w_n, b\}$ determine the state of the neuron, and the processing logic of this neuron can be determined by fixing those parameters. When the number of input nodes $n = 1$ (single input), the neuron model can be further simplified as

$$y = wx + b$$

Then we can plot the change of y as a function of x as shown in Figure 0.3. As the input signal x increases, the output y also increases linearly. Here parameter w can be understood as the slope of the straight line and b is the bias of the straight line.

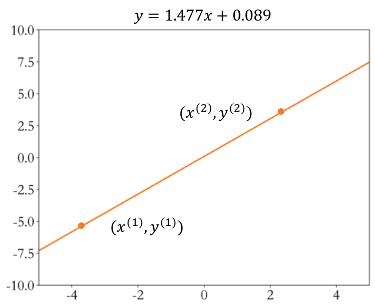

Figure 0.3 Single-input linear neuron model

For a certain neuron, the mapping relationship $f_{w,b}$ between x and y is unknown but fixed. Two points can determine a straight line. In order to estimate the value of w and b, we only need to sample any two data points $\left(x^{(1)}, y^{(1)}\right)$ and $\left(x^{(2)}, y^{(2)}\right)$ from the straight line in Figure 0.3, where the superscript indicates the data point number:

$$y^{(1)} = wx^{(1)} + b$$

$$y^{(2)} = wx^{(2)} + b$$

If $\left(x^{(1)}, y^{(1)}\right) \neq \left(x^{(2)}, y^{(2)}\right)$, we can solve above equations to get the value of w and b. Let’s consider a specific example: $x^{(1)} = 1,\ y^{(1)} = 1.567,\ x^{(2)} = 2,\ y^{(2)} = 3.043$. Substituting the numbers in the above formula gives:

$$1.567 = w \cdot 1 + b$$

$$3.043 = w \cdot 2 + b$$

This is the system of binary linear equations that we learned in junior or high school. The analytical solution can be easily calculated using the elimination method, that is, $w = 1.477,\ b = 0.089$.

You can see that we only need to two different data points to perfectly solve the parameters of a single-input linear neuron model. For linear neuron models with N input, we only need to sample $N+1$ different data points. It seems that the linear neuron models can be perfectly resolved. So what's wrong with the above method? Considering that there may be observation errors for any sampling point, we assume that the observation error variable ϵ follows a normal distribution $\mathcal{N}(\mu,\sigma^2)$ with μ as mean and σ^2 as variance, then the samples meet:

$$y = wx + b + \epsilon, \epsilon \sim \mathcal{N}(\mu,\sigma^2)$$

Once the observation error is introduced, even if it is as simple as a linear model, if only two data points are sampled, it may bring a large estimation bias. As shown in Figure 0.4, the data points all have observation errors. If the estimation is based on the two blue rectangular data points, the estimated blue dotted line would have a large deviation from the true orange straight line. In order to reduce the estimation bias introduced by observation errors, we can sample multiple data points $\mathbb{D} = \{(x^{(1)},y^{(1)}),(x^{(2)},y^{(2)}),\dots,(x^{(n)},y^{(n)})\}$ and then find a "best" straight line, so that it minimizes the sum of errors between all sampling points and the straight line.

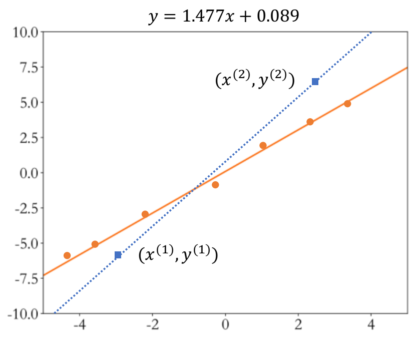

Figure 0.4 Model with observation errors

Due to the existence of observation errors, there may not be a straight line that perfectly passes through all the sampling points $\mathbb{D}$. Therefore, we hope to find a “good” straight line close to all sampling points. How to measure "good" and "bad"? A natural idea is to use the sum of the squared difference between the predicted value $wx^{(i)}+b$ and the true value $y^{(i)}$ at all sampling points as the total error, i.e.:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\left(wx^{(i)}+b-y^{(i)}\right)^2$$

Then search a set of parameters w^* and b^* to minimize the total error $\mathcal{L}$. And the straight line corresponding to the minimal total error is the optimal straight line we are looking for, that is

$$w^*, b^* = \underset{w,b}{\arg\min}\frac{1}{n}\sum_{i=1}^{n}\left(wx^{(i)}+b-y^{(i)}\right)^2$$

Here n represents the number of sampling points. The total error is called Mean Squared Error (MSE).

2.2 Optimization method

Now let's summarize the above solution: we need to find the optimal parameters w^* and b^*, so that the input and output meet a linear relationship $y^{(i)} = wx^{(i)} + b,\ i \in [1, n]$. However, due to the existence of observation errors ϵ, it is necessary to sample a data set $\mathbb{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(n)}, y^{(n)})\}$, composed of a sufficient number of data samples, to find an optimal set of parameters w^* and b^* to minimize the mean square error $\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\left(wx^{(i)} + b - y^{(i)}\right)^2$.

For a single-input neuron model, only two samples are needed to obtain the exact solution of the equations by the elimination method. This exact solution derived by a strict formula is called an analytical solution. However, in the case of multiple data points ($n \gg 2$), there is probably no analytical solution. We can only use numerical optimization methods to to obtain an approximate numerical solution. Why is it called optimization? This is because the computer's calculation speed is very fast, we can use the powerful computing power to "search" and "try" multiple times, thereby reducing the error $\mathcal{L}$ step by step. The simplest optimization method is brute force search or random experiment. For example, to find the most suitable w^* and b^*, we can randomly sample any w and b from the real number space, and calculate the error value $\mathcal{L}$ of the corresponding model. Pick out the smallest error $\mathcal{L}^*$ from all the experiments $\{\mathcal{L}\}$, and its corresponding w^* and b^* are the optimal paramaters we are looking for.

This brute force algorithm is simple and straightforward, but it is extremely inefficient when faced with large-scale, high-dimensional optimization problems, which is basically not feasible. Gradient descent is the most commonly used optimization algorithm in neural network training. With the parallel acceleration capability of powerful Graphics Processing Unit (GPU) chips, it is very suitable for optimizing neural network models with massive data. Naturally it is also suitable for optimizing our simple linear neuron model. Since the gradient descent algorithm is the core algorithm of deep learning, we will first apply the gradient descent algorithm to solve simple neuron models and then detail its application in neural networks in Chapter 7.

With the concept of derivative, if we want to solve the maximum and minimum values of a function, we can simply set the derivative function to be 0 and find the corresponding independent variable values i.e. the stagnation point, and then check the stagnation type. Taking the function $f(x) = x^2 \cdot si\,n(x)$ as an example, we can plot the function and its derivative in the interval $x \in [-10,10]$, where the blue solid line is $f(x)$ and the yellow dotted line is $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$ as shown in Figure 0.5. It can be seen that the points where the derivative (dashed line) is 0 are the stagnation points, and both the maximum and minimum values of $f(x)$ appear in the stagnation points.

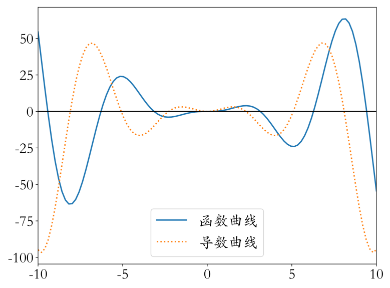

Figure 0.5 Function $f(x) = x^2 \cdot sin\,(x)$ and its derivative

The gradient of a function is defined as a vector of partial derivatives of the function on each independent variable. Considering a 3-dimensional function $z = f(x, y)$, the partial derivative of the function with respect to the independent variable x is $\frac{\partial z}{\partial x}$, the partial derivative of the function with respect to the independent variable y is recorded as $\frac{\partial z}{\partial y}$, and the gradient ∇f is a vector $(\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y})$. Let's look at a specific function $f(x, y) = -(\cos^2 x + \cos^2 y)^2$. As shown in Figure 0.6, the length of the red arrow in the plane represents the modulus of the gradient vector, and the direction of the arrow represents the direction of the gradient vector. It can be seen that the direction of the arrow always points to the direction where the function value increases at the current position. The steeper the function surface, the longer the length of the arrow, and the larger the modulus of the gradient.

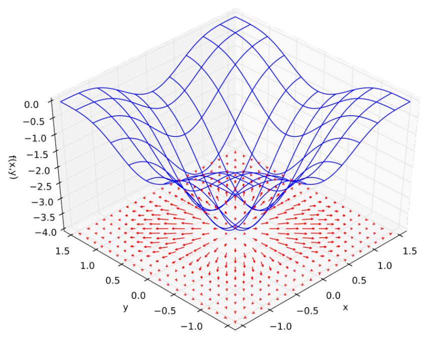

Figure 0.6 A function and its gradient[②]

Through the above example, we can intuitively feel that the gradient direction of the function always points to the direction in which the function value increases, then the opposite direction of the gradient should point to the direction in which the function value decreases.

[②] Picture source: https://en.wikipedia.org/wiki/Gradient?oldid=747127712

$$\boldsymbol{x}' = \boldsymbol{x} - \eta \cdot \nabla f \tag{0.1}$$

To take advantage of this, we just need to follow the above equation to iteratively update $\boldsymbol{x}'$, then we can get smaller and smaller function values. η is used to scale the gradient vector, which is generally set to a smaller value, such as 0.01 or 0.001. In particular, for one-dimensional functions, the above vector form can be written into a scalar form:

$$x' = x - \eta \cdot \frac{\mathrm{d}y}{\mathrm{d}x}$$

By iterating and updating x'several times through the above formula, the function value y' at x' is always more likely to be smaller than the function value at x.

The method of optimizing parameters by the formula (2.1) is called gradient descent algorithm. It calculates the gradient ∇f of the function f and iteratively updates the parameters θ to obtain the optimal numerical solution of the parameters θ when the function f reaches its minimum value. It should be noted that model input in deep learning is generally represented as $\boldsymbol{x}$, and the parameters to be optimized are generally represented by θ、w、b.

Now we will apply the gradient descent algorithm to calculate the optimal parameters w^* and b^* in the beginning of this session. What is to be minimized here is the mean square error function:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\left(wx^{(i)} + b - y^{(i)}\right)^2$$

The model parameters that need to be optimized are w and b, so we update them iteratively using the following equations.

$$w' = w - \eta\frac{\partial \mathcal{L}}{\partial w}$$

$$b' = b - \eta\frac{\partial \mathcal{L}}{\partial b}$$

2.3 Linear model in action

Let's actually train a single-input linear neuron model using the gradient descent algorithm. First we need to sample multiple data points. For a toy example with a known model, we directly sample from the specified real model:

$$y = 1.477x + 0.089$$

1. **Sampling data**

In order to simulate the observation errors, we add an independent error variable ϵ to the model. ϵ follows a Gaussian distribution with a mean value of 0 and a standard deviation of 0.01:

$$y = 1.477x + 0.089 + \epsilon, \epsilon \sim \mathcal{N}(0, 0.01^2)$$

By randomly sampling $n = 100$ times, we obtain a training data set $\mathbb{D}^{\text{train}}$ using the following code.

```
data = [] # A list to save data samples
```

```
for i in range(100): # repeat 100 times
    # Randomly sample x from a uniform distribution
    x = np.random.uniform(-10., 10.)
    # Randomly sample from Gaussian distribution
    eps = np.random.normal(0., 0.01)
   # Calculate model output with random errors
    y = 1.477 * x + 0.089 + eps
    data.append([x, y]) # save to data list
data = np.array(data) # convert to 2D Numpy array
```

In above code, we perform 100 samples in a loop, randomly sampling one data x from the uniform $U(-10, 10)$ distribution each time, and randomly sampling noise ϵ from the Gaussian distribution $\mathcal{N}(0, 0.1^2)$, and then generate the data using the real model and random noise ϵ, and save it as a Numpy array.

2. Calculate the mean square error

Now let's calculate the mean sqaure error on the training set by averaging the squared difference between the predicted value and the true value at each data point. We can achieve this using the following function.

```
def mse(b, w, points):
   # Calculate MSE based on current w and b
   totalError = 0
    # Loop through all points
   for i in range(0, len(points)):
      x = points[i, 0] # Get ith input
      y = points[i, 1] # Get ith output
      # Calculate the total sqaured error
      totalError += (y - (w * x + b)) ** 2
   # Calculate the mean of the total sqaured error
   return totalError / float(len(points))
```

3. **Calculate gradient**

According to the gradient descent algorithm, we need to calculate the gradient at each data point $(\frac{\partial \mathcal{L}}{\partial w}, \frac{\partial \mathcal{L}}{\partial b})$. First, consider expanding the mean square error function $\frac{\partial \mathcal{L}}{\partial w}$:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \frac{1}{n}\sum_{i=1}^{n}\left(wx^{(i)} + b - y^{(i)}\right)^2}{\partial w} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial\left(wx^{(i)} + b - y^{(i)}\right)^2}{\partial w}$$

Because

$$\frac{\partial g^2}{\partial w} = 2 \cdot g \cdot \frac{\partial g}{\partial w}$$

We have

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{1}{n}\sum_{i=1}^{n} 2\left(wx^{(i)} + b - y^{(i)}\right) \cdot \frac{\partial\left(wx^{(i)} + b - y^{(i)}\right)}{\partial w}$$

$$= \frac{1}{n}\sum_{i=1}^{n} 2\left(wx^{(i)} + b - y^{(i)}\right) \cdot x^{(i)}$$

$$= \frac{2}{n}\sum_{i=1}^{n} \left(wx^{(i)} + b - y^{(i)}\right) \cdot x^{(i)} \qquad (0.2)$$

If it is difficult to understand the above derivation, you can review the gradient-related courses in mathematics. At the same time, it will be introduced in detail in Chapter 7 of this book. We can remember the final expression of $\frac{\partial \mathcal{L}}{\partial w}$ for now. In the same way, we can derive the expression of the partial derivative $\frac{\partial \mathcal{L}}{\partial b}$:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \frac{1}{n}\sum_{i=1}^{n}\left(wx^{(i)} + b - y^{(i)}\right)^2}{\partial b} = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial\left(wx^{(i)} + b - y^{(i)}\right)^2}{\partial b}$$

$$= \frac{1}{n}\sum_{i=1}^{n} 2\left(wx^{(i)} + b - y^{(i)}\right) \cdot \frac{\partial\left(wx^{(i)} + b - y^{(i)}\right)}{\partial b}$$

$$= \frac{1}{n}\sum_{i=1}^{n} 2\left(wx^{(i)} + b - y^{(i)}\right) \cdot 1$$

$$= \frac{2}{n}\sum_{i=1}^{n} \left(wx^{(i)} + b - y^{(i)}\right) \qquad (0.3)$$

According to the expressions (2.2) and (2.3), we only need to calculate the mean value of $\left(wx^{(i)} + b - y^{(i)}\right) \cdot x^{(i)}$ and $\left(wx^{(i)} + b - y^{(i)}\right)$ at each data point. The implementation is as follows:

```
def step_gradient(b_current, w_current, points, lr):

    # Calculate gradient and update w and b.

    b_gradient = 0

    w_gradient = 0

    M = float(len(points)) # total number of samples

    for i in range(0, len(points)):
        x = points[i, 0]
        y = points[i, 1]
        # dL/db: grad_b = 2(wx+b-y) from equation (2.3)
        b_gradient += (2/M) * ((w_current * x + b_current) - y)
        # dL/dw: grad_w = 2(wx+b-y)*x from equation (2.2)
        w_gradient += (2/M) * x * ((w_current * x + b_current) - y)
    # Update w',b' according to gradient descent algorithm
    # lr is learning rate
    new_b = b_current - (lr * b_gradient)
    new_w = w_current - (lr * w_gradient)
    return [new_b, new_w]
```

4. **Gradient update**

After calculating the gradient of the error function at w and b, we can update the value of w and b according to equation (2.1). Training all samples of the data set once is knowns as one epoch. We can iterate multiple epochs using previous defined functions. The implementation is as follows:

```
def gradient_descent(points, starting_b, starting_w, lr, num_iterations):
    # Update w, b multiple times
    b = starting_b # initial value for b
    w = starting_w # initial value for w
    # Iterate num_iterations time
    for step in range(num_iterations):
        # Update w, b once
        b, w = step_gradient(b, w, np.array(points), lr)
        # Calcualte current loss
        loss = mse(b, w, points)
        if step%50 == 0: # print loss and w, b
            print(f"iteration:{step}, loss:{loss}, w:{w}, b:{b}")
    return [b, w] # return the final value of w and b
```

The main training function is defined as follows:

```
def main():
    # Load training dataset
    data = []
    for i in range(100):
        x = np.random.uniform(3., 12.)
        # mean=0, std=0.1
        eps = np.random.normal(0., 0.1)
        y = 1.477 * x + 0.089 + eps
        data.append([x, y])
    data = np.array(data)
    lr = 0.01 # learning rate
    initial_b = 0 # initialize b
    initial_w = 0 # initialize w
    num_iterations = 1000
    # Train 1000 times and return optimal w*,b* and corresponding loss
    [b, w]= gradient_descent(data, initial_b, initial_w, lr, num_iterations)
    loss = mse(b, w, data) # Calculate MSE
    print(f'Final loss:{loss}, w:{w}, b:{b}')
```

Comment [XZ1]: Add code for generating "data" varaible

After 1000 iterative updates, the final w and b are the "optimal" solution we are looking for. The results are as follows:

```
iteration:0, loss:11.437586448749, w:0.88955725981925, b:0.02661765516748428
iteration:50, loss:0.111323083882350, w:1.48132089048970, b:0.58389075913875
iteration:100, loss:0.02436449474995, w:1.479296279074, b:0.78524532356388
…
iteration:950, loss:0.01097700897880, w:1.478131231919, b:0.901113267769968
Final loss:0.010977008978805611, w:1.4781312318924746, b:0.901113270434582
```

It can be seen that at the 100th iteration, the values of w and b are already close to the real model values. The w and b obtained after 1000 updates is very close to the real model. The mean square error of the training process is shown in Figure 0.7.

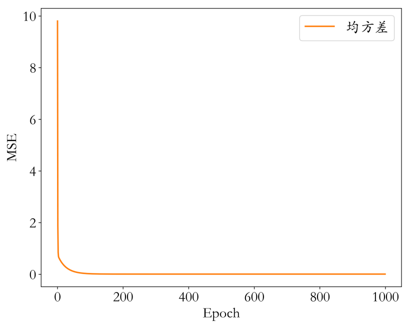

Figure 0.7 MSE change during training process

The above example shows the power of the gradient descent algorithm in solving model parameters. It should be noted that for complex nonlinear models, the parameters solved by the gradient descent algorithm may be a local minimum solution instead of a global minimum solution, which is determined by the function non-convexity. However, we found in practice that the performance of the numerical solution obtained by the gradient descent algorithm can often be optimized very well, and the corresponding solution can be directly used to approximate the optimal solution.

2.4 Linear regression

A brief review of our exploration: we first assume that the neuron model with n input is a linear model, and then we can calculate the extract solution of $\boldsymbol{w}$ and $\boldsymbol{b}$ through $n+1$ samples. After introducing the observation error, we can sample multiple sets of data points and optimize it through the gradient descent algorithm to obtain the numerical solution of $\boldsymbol{w}$ and $\boldsymbol{b}$.

If we look at this problem from another angle, it can actually be understood as a set of continuous value (vector) prediction problems. Given a data set $\mathbb{D}$, we need to learn a model from the data set in order to predict the output value of an unseen sample. After assuming the type of model, the learning process becomes a problem of searching for model parameters. For example, if we assume that the neuron is a linear model, then the training process is the process of searching the linear model parameters $\boldsymbol{w}$ and $\boldsymbol{b}$. After training, we can use the model output value as an approximation of the real value for any new input. From this perspective, it is a continuous value prediction problem.

In real life, continuous value prediction problems are very common, such as the prediction of stock price trends, the prediction of temperature and humidity in weather forecasts, the prediction of age, the prediction of traffic flow, and so on. We call it a Regression problem if its predictions are in a continuous range of real numbers, or belong to a certain continuous range of real numbers. In particular, if a linear model is used to approximate the real model, then we call it Linear Regression, which is a specific implementation of regression problems.

In addition to the continuous value prediction problem, is there a discrete value prediction problem? For example, the prediction of the front and back of a coin can only have two types of predictions: front and back. Given a picture, the type of objects in this picture can only be some discrete categories such as cats or dogs. Problems like those are known as Classification problems, which will be introduced in next chapter.

2.5 References

[1] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics,* 5, pp. 115-133, 01 12 1943.

[2] F. Rosenblatt, The Perceptron, a Perceiving and Recognizing Automaton Project Para, Cornell Aeronautical Laboratory, 1957.