# 1 机器学习基础

机器学习可以分为监督学习(Supervised Learning)和非监督学习(Unsupervised Learning)，其中，监督学习又有两类任务：回归和分类。

## 1.1 线性回归

线性回归是处理回归任务的最简单模型，对于简单线性回归来说，我们可以作出如下假设：

$h_\theta(x^{(i)}) = \theta_0 + \theta_1 x^{(i)}$

其中的$\theta_0$和$\theta_1$代表模型的参数。线性回归的目标是求得最适合的$\theta_0$和$\theta_1$使得模型效果最好。

**数据集**

我们通常收集一系列的真实数据，例如多栋房屋的真实售出价格和它们对应的面积。我们希望在这个数据上面寻找模型参数来使模型的预测价格与真实价格的误差最小。在机器学习术语里，该数据集被称为训练数据集（training data set）或训练集（training set），一栋房屋被称为一个样本（sample），其真实售出价格叫作标签（label），用来预测标签的两个因素叫作特征（feature）。特征用来表征样本的特点。

**代价函数（损失函数）**

衡量模型效果好坏的函数叫代价函数(Cost Function)，其中，均方误差(Mean Squared Error)是最简单的代价函数。

均方误差就是求预测值与真实值之间的差值的平方，即：

$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$

**梯度下降**

我们的目标是找到合适的参数$\theta$，使代价函数最小。因为梯度的反方向是函数值减小最快的方向，故我们更新自变量的策略为：

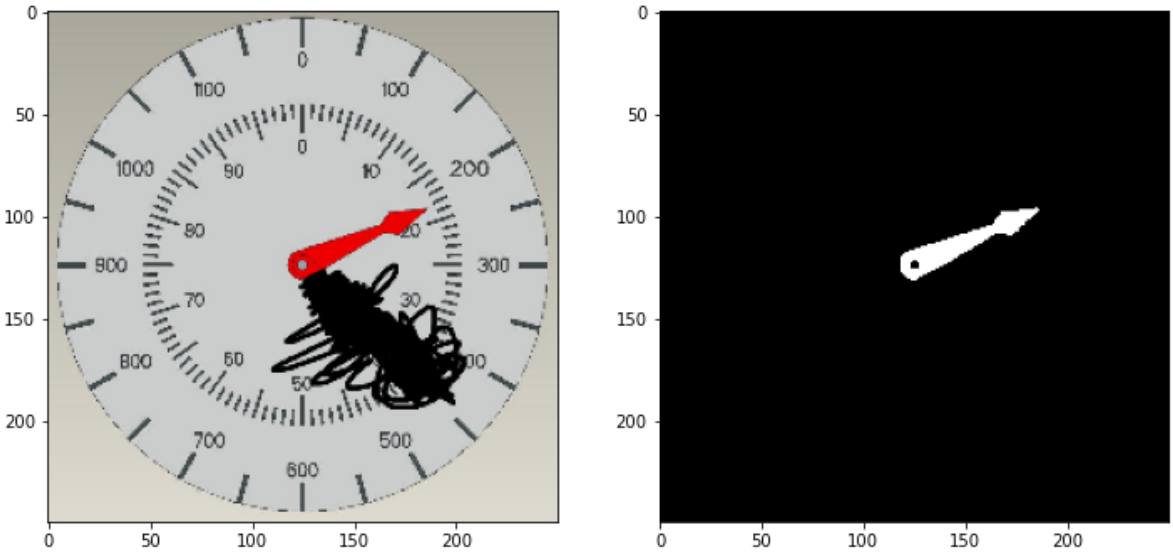
$$\begin{aligned}\theta_j &= \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \\ &= \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}x_j^{(i)}\end{aligned}$$

其中$\alpha$是学习率，即参数更新的步长，$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$

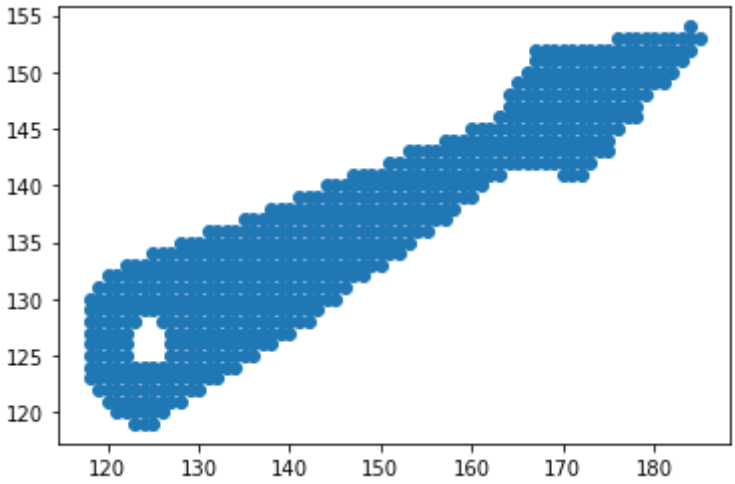
**例子：线性回归检测指针读数**

```
import numpy as np
from matplotlib import pyplot as plt
import cv2

def extract_red(image):
    red_lower1 = np.array([0, 43, 46])
    red_upper1 = np.array([10, 255, 255])
    red_lower2 = np.array([156, 43, 46])
    red_upper2 = np.array([180, 255, 255])
    dst = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    mask1 = cv2.inRange(dst, lowerb=red_lower1, upperb=red_upper1)
    mask2 = cv2.inRange(dst, lowerb=red_lower2, upperb=red_upper2)
```

```
    mask = cv2.add(mask1, mask2)
    return mask

img = cv2.imread('clock.png')  # 原图
img = cv2.resize(img, (250, 250))
plt.figure(figsize=(12, 12))
plt.subplot(121)
plt.imshow(img[:, :, ::-1])
mask = extract_red(img)  # 提取红色
plt.subplot(122)
plt.imshow(mask, cmap='gray')
plt.show()
```



```
X = np.argwhere(mask==255)[:, 1]
Y = np.abs(mask.shape[0] - np.argwhere(mask==255)[:, 0])
plt.scatter(X, Y)
plt.show()
```

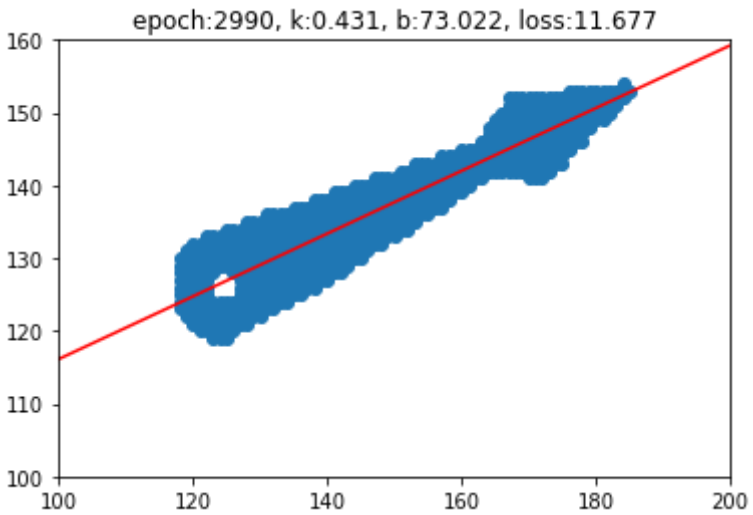


**sklearn**

```
from sklearn import linear_model      # 引入线性回归方法
lin_reg = linear_model.LinearRegression()       # 创建线性回归的类
X1 = X.reshape(-1, 1)
lin_reg.fit(X1,Y)         # 输入特征X和因变量y进行训练
print("模型系数：",lin_reg.coef_)                 # 输出模型的系数
print("模型得分：",lin_reg.score(X1,Y))     # 输出模型的决定系数R^2
print("模型偏置：", lin_reg.intercept_)    # 输出模型偏置b
```

```
模型系数： [0.43111815]
模型得分： 0.848103775789403
模型偏置： 73.02176077297048
```

```
from IPython import display
# y = kx + b

iters = 3000
k = 1
b = 1
lr = 0.00005
size = X.shape[0]
for epoch in range(iters):
    y_pred = k * X + b
    loss = np.mean((y_pred - Y) ** 2)
    k = k - (lr/size) * np.dot(y_pred - Y, X.T)
    b = b - 10000 * (lr/size) * np.sum(y_pred - Y)
    if epoch % 10 == 0:
        display.clear_output(wait=True)
        x_line = np.linspace(100, 200, 1000)
        y_line = k * x_line + b
        plt.plot(x_line, y_line, 'r')
        plt.title(f"epoch:{epoch}, k:{round(k, 3)}, b:{round(b, 3)}, loss:
{round(loss, 3)}")
        plt.xlim(100, 200)
        plt.ylim(100, 160)
        plt.scatter(X, Y)
        plt.pause(0.1)
```



## 1.2 逻辑回归

逻辑回归虽然名字中有“回归”，但其实际上是分类模型，并常用于二分类。首先我们定义sigmoid函数为：

$y = \frac{1}{1+e^{-x}}$

其定义域为：$(-\infty, +\infty)$,值域为：$(0, 1)$,导数为：$y' = y(1-y)$

假设我们的模型为：

$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$

其中$x:[1, x_1, x_2]^T$　　$\theta:[b, w_1, w_2]^T$

逻辑回归的代价函数与线性回归不同，其主要方法是极大似然估计。其推导如下：

设$p\{y=1|x;\theta\} = h_\theta(x)$，设$p\{y=0|x;\theta\} = 1 - h_\theta(x)$

则似然函数：$L = \prod_{i=1}^m p^{(i)} = \prod_{i=1}^m h_\theta^{y^{(i)}}(x^{(i)})(1-h_\theta(x^{(i)}))^{1-y^{(i)}}$

两边取对数：$log(L) = \sum_{i=1}^m \{y^{(i)} log h_\theta(x^{(i)}) + (1-y^{(i)}) log[1-h_\theta(x^{(i)})]\}$

我们的目标是使L最大，即事件发生概率最大，所以定义代价函数如下：

$J(\theta) = -\frac{1}{m}\sum_{i=1}^m \left[y^{(i)} log(h_\theta(x^{(i)})) + (1-y^{(i)}) log(1-h_\theta(x^{(i)}))\right]$

逻辑回归梯度下降：

$\frac{\partial J}{\partial \theta} = -\frac{1}{m}\sum_{i=1}^m [y^{(i)} \frac{\frac{\partial h_\theta(x^{(i)})}{\partial \theta}}{h_{\theta(x^{(i)})}} + (1-y^{(i)}) \frac{\frac{\partial h_\theta(x^{(i)})}{\partial \theta}}{1-h_{\theta(x^{(i)})}}]$

其中$\frac{\partial h(x)}{\partial \theta} = [h(x)[1-h(x)]x_j]$,代入上式化简可得：

$\frac{\partial J}{\partial \theta} = \frac{1}{m}\sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$

逻辑回归一般解决二分类问题，当遇到多分类时，通常有两种方法：

- 一对多：

例如现在有三个类别：A,B,C，我们可以训练一个模型分出A与B,C，另一个模型分出B与A,C，最后一个模型分出C与A,B

优点：需要训练的模型较少

缺点：样本不均衡

- 一对一：

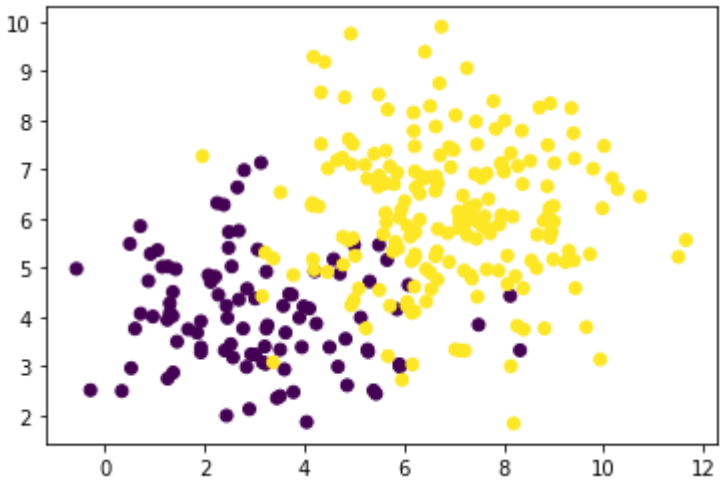
假如需要分k类，我们需要训练$C_k^2$个模型，即每两个类训练一个模型

优点：数据少、样本均衡

缺点：需要训练的模型较多

```
import numpy as np
import matplotlib.pyplot as plt
import random
from tqdm import tqdm
from IPython import display
```

```
## 创建数据集
np.random.seed(1115)

data_0 = np.random.multivariate_normal(mean=[3, 4], cov=[[3, 0], [0, 1]],
size=100)  # 第0类:100个样本
data_1 = np.random.multivariate_normal(mean=[7, 6], cov=[[3, 0], [0, 2]],
size=200)  # 第1类:200个样本
data_x = np.vstack((data_0, data_1))
data_y = np.hstack((np.array([0]*100), np.array([1]*200)))
plt.scatter(data_x[:, 0], data_x[:, 1], c=data_y)
plt.show()
```




```
data = list(zip(data_x, data_y))  # 将x，y打包
random.shuffle(data)  # 打乱
data_x, data_y = zip(*data) # 解包
data_x = np.array(data_x)
data_y = np.array(data_y)
```

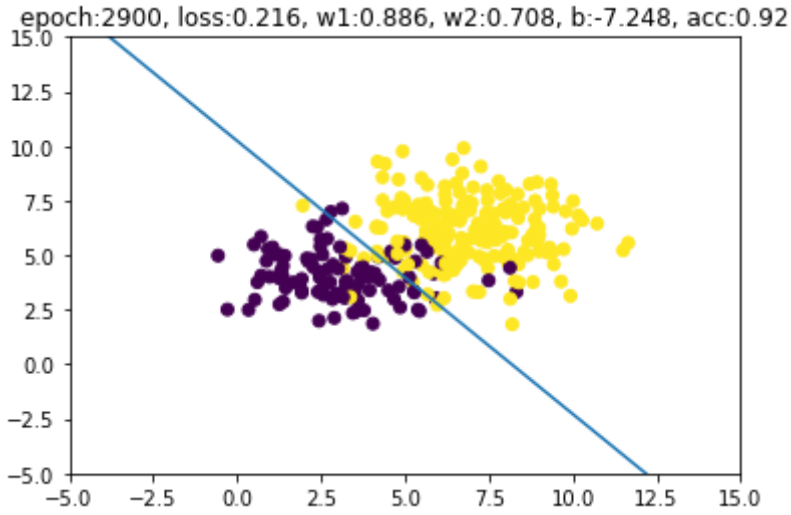
```
def sigmoid(x):
    return 1/(1+np.exp(-x))
```

```
def cal_loss(y_pred, data_y):
    return -np.mean(data_y*np.log(y_pred) + (1-data_y)*np.log(1-y_pred))
```

```
def cal_acc(y_pred, data_y):
    return np.mean((y_pred >= 0.5) == data_y)
```

```
iters = 3000
w1 = 1
w2 = 1
b = 1
lr = 0.1
size = data_x.shape[0]
for epoch in range(iters):
    y_pred = sigmoid(w1*data_x[:, 0] + w2*data_x[: ,1] + b)
```

```
    loss = cal_loss(y_pred, data_y)
    w1 = w1 - (lr/size) * np.dot((y_pred - data_y), data_x[:, 0].T)
    w2 = w2 - (lr/size) * np.dot((y_pred - data_y), data_x[:, 1].T)
    b = b - (lr/size) * np.sum(y_pred - data_y)
    if epoch % 100 == 0 :
        display.clear_output(wait=True)
        line_x = np.linspace(-5, 15, 1000)
        line_y = (-b-w1*line_x)/w2
        plt.xlim(-5, 15)
        plt.ylim(-5, 15)
        plt.plot(line_x, line_y)
        plt.title(f'epoch:{epoch}, loss:{round(loss, 3)}, w1:{round(w1 ,3)}, w2:
{round(w2, 3)}, b:{round(b, 3)}, acc:{round(cal_acc(y_pred, data_y), 3)}')
        plt.scatter(data_x[:, 0], data_x[:, 1], c=data_y)
        plt.savefig('lr.jpg')
        plt.show()
        plt.pause(0.5)
```




**sklean**

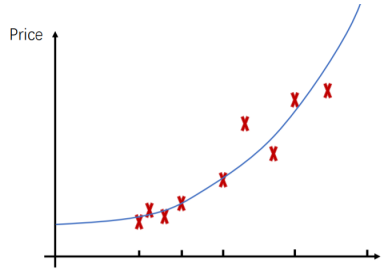
```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(data_x, data_y)


print(model.coef_)
print(model.intercept_)
print(model.score(data_x, data_y))
```

```
[[1.18105812 1.13321831]]
[-10.82334604]
0.91
```

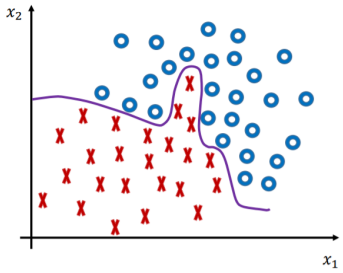
# 1.3 神经网络（Neural Network）

## 1.3.1 为什么需要神经网络?

与前面介绍的线性回归和逻辑回归不同，神经网络通常解决非线性问题，如下图（上）所示的非线性回归问题和下图（下）所示的非线性分类问题：




$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_1 x^2$$
$$= \theta_0 + \theta_1(size) + \theta_1(size)^2$$
$$= \theta_0 + \theta_1 x_1 + \theta_1 x_2$$

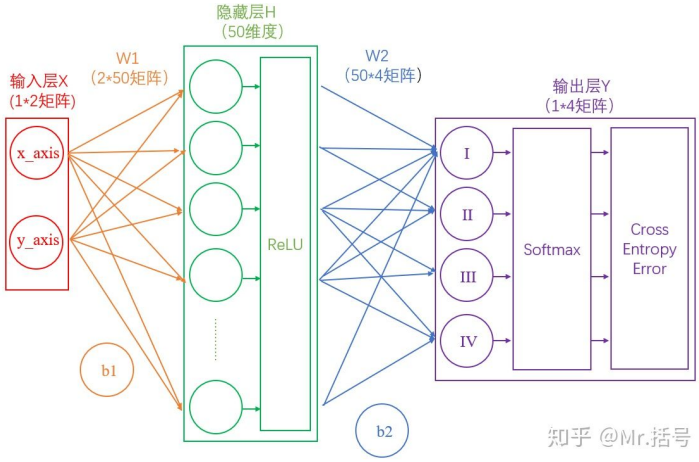



$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_1 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 x_2 + \theta_5 x_1^3 x_2 + \theta_6 x_1 x_2^2 + \cdots)$$

线性回归和逻辑回归就不能很好地处理这两种问题。

## 1.3.2 什么是神经网络?

神经网络的基本结构如下图所示：




设输入为x维，隐藏层为h维，输出为y维，激活函数为g(x)

- 输入层

输入层维度数就是样本的特征数，例如一个二维坐标有两个特征：x和y，所以输入层为二维；一张32×32的图片有$32^2$个特征，故输入层的维度数为$32^2$

- 输入层到隐藏层

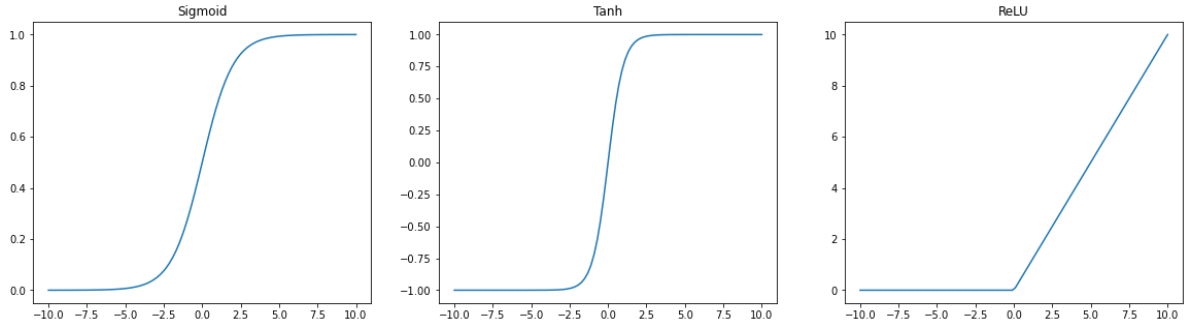
输入层到隐藏层其实就是h个线性回归模型，表达式为：

$$H_{(1\times h)} = X_{(1\times x)} \cdot W_{1(x\times h)} + b_{1(1\times h)}$$

- 激活层

简而言之，激活层是为矩阵运算的结果添加非线性的。常用的激活函数有Sigmoid、Tanh和ReLU。它们的函数图像如下所示：

```
x = np.linspace(-10, 10, 100)
y1 = 1/(1+np.exp(-x))
plt.figure(figsize=(20, 5))
plt.subplot(131)
plt.plot(x, y1)
plt.title('Sigmoid')
plt.savefig('sigmoid.jpg')
plt.subplot(132)
y2 = np.tanh(x)
plt.plot(x, y2)
plt.title('Tanh')
plt.subplot(133)
y3 = list(map(lambda x:x if x > 0 else 0, x))
plt.plot(x, y3)
plt.title('ReLU')
plt.savefig('activation.jpg', dpi=500)
plt.show()
```




- 从隐藏层到输出层

连接隐藏层和输出层的是$W_2$和$b_2$。同样是通过矩阵运算进行的：

$$Y = H \cdot W_2 + b_2$$

- Softmax

假设有一个数组Y，$Y_i$表示$Y$中的第$i$个元素，那么这个元素的Softmax值为：

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

该元素的softmax值，就是该元素的指数与所有元素指数和的比值。但是如果$z$的值很大时，会出现数据溢出的情况，优化版的softmax可以防止这种情况，其表达式如下：

$$S_i = \frac{e^{z_i - max(z)}}{\sum_{j=1}^{y} e^{z_j - max(z)}}$$

- 交叉熵损失函数

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{c=1}^{C} y_c^i ln p_c^i$$

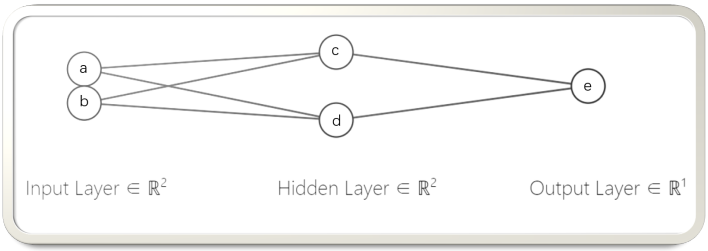
对于分类任务来说，$y = (0, 0, \cdots, 1, \cdots, 0)^T$，设第$k$个元素为1，则：

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} ln p_k^i$$

其中$p_i = \frac{e^{z_k}}{\sum e^{z_j}}$，

### 1.3.3 怎么实现神经网络?

神经网络最核心的部分就是bp算法，其主要思想就是高数中的复合函数求导，下面举一个简单的例子：




$$ 其中，\left{
\begin{aligned}
c & = a+b \\
d & = b +1 \\
e & = c \times d
\end{aligned}
\right.

$$故，\begin{cases}
\frac{\partial e}{\partial d} = c \\
\frac{\partial e}{\partial c} = d \\
\frac{\partial e}{\partial b} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial b} + \frac{\partial e}{\partial c}\frac{\partial c}{\partial b} = d + c = a + 2b + 1 \\
\frac{\partial e}{\partial a} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial a} + \frac{\partial e}{\partial c}\frac{\partial c}{\partial a} = d = b + 1
\end{cases}

下面完整推导一次神经网络的反向传播：

| 变量 | 含义 |
| --- | --- |
| x | 输入样本 |
| h | 第一层输出 |
| a | 对第一层输出激活 |
| z | 第二层输出 |
| p | 对第二层输出进行softmax |
| L | 对p求损失 |
| w1 | 第一层权重 |
| w2 | 第二层权重 |
| b1 | 第一层偏置 |
| b2 | 第二层偏置 |
| y | 标签 |

这里以一个样本为例：
设$y = (0, 0, \cdots, 1, \cdots, 0)^T$，其中第k个元素为1

1. softmax - loss
   $L = -lnp_k$

$$\frac{\partial L_k}{\partial p_i} = \begin{cases} -\frac{1}{p_k} & i = k \\ 0 & i \neq k \end{cases}$$

2. z - softmax

$p_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$

当$i = k$时

$\frac{\partial p_k}{\partial z_k} = \frac{\sum_j e^{z_j} e^{z_k} - e^{z_k} e^{z_k}}{(\sum_j e^{e_j})^2} = p_k(1 - p_k)$

此时，$\frac{\partial L_k}{\partial z_k} = p_k - 1$

当$i \neq k$时

$\frac{\partial p_k}{\partial z_i} = -\frac{e^{z_k} e^{z_i}}{(\sum_j e^{e_j})^2} = -p_k p_i$

此时，$\frac{\partial L_k}{\partial z_i} = p_i$

前面几层参考线性回归即可，至此，神经网络全部介绍完了，下面用代码实现一下

```
from tensorbay import GAS
from tensorbay.dataset import Segment

# Authorize a GAS client.
gas = GAS('<your GAS ak>')

# Get a dataset client.
dataset_client = gas.get_dataset("Fashion-MNIST-PyTorch")

# List dataset segments.
segments = dataset_client.list_segment_names()

# Get a segment by name
segment = Segment("data", dataset_client)
for data in segment:
    with open("FashionMNIST.zip", "wb") as fp:
        fp.write(data.open().read())
```

```
from torch import nn
from torchvision import transforms
from torch.utils.data import DataLoader
from torchvision import datasets
from torch import optim
from tqdm import tqdm
import numpy as np
from torch import save

mnist_train = datasets.FashionMNIST(root='.', train=True, download=False,
transform=transforms.ToTensor())
mnist_test = datasets.FashionMNIST(root='.', train=False, download=False,
transform=transforms.ToTensor())
```

```
train_iter = DataLoader(mnist_train, batch_size=64, shuffle=True, num_workers=0)
test_iter = DataLoader(mnist_test, batch_size=64, shuffle=False, num_workers=0)
print(mnist_train.data[0].shape)  # 图片大小
print(mnist_train.targets.unique())  # 标注信息
```

```
torch.Size([28, 28])
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
class Fashion(nn.Module):
    def __init__(self):
        super(Fashion, self).__init__()
        self.net = nn.Sequential(nn.Linear(28*28, 1024),
                                 nn.ReLU(),
                                 nn.Linear(1024, 10))
    def forward(self, x):
        x = x.view(x.shape[0], -1)
        y = self.net(x)
        return y

def evaluate_accuracy(data_iter, net):
    test_acc =[]
    for X, y in data_iter:
        test_acc.append((net(X).argmax(dim=1) == y).float().sum().item() /
y.shape[0])
    return np.mean(test_acc)

epochs = 5
net = Fashion()
optimizer = optim.SGD(net.parameters(), lr=0.1)
criterion = nn.CrossEntropyLoss()
for epoch in range(epochs):
    train_loss = []
    train_acc = []
    for x, y in train_iter:
        y_pred = net(x)
        loss = criterion(y_pred, y).sum()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_loss.append(loss.item())
        train_acc.append((y_pred.argmax(dim=1)==y).sum().item() / y.shape[0])
    test_acc = evaluate_accuracy(test_iter, net)
    print(f'epoch:{epoch} train_loss:{np.mean(train_loss)} train_acc:
{np.mean(train_acc)} test_acc:{test_acc}')
save(net.state_dict(), 'last.pt')
```

```
epoch:0 train_loss:0.6012519435810127 train_acc:0.790961487206823
test_acc:0.8362858280254777
epoch:1 train_loss:0.4248647812777745 train_acc:0.8482476012793176
test_acc:0.8228503184713376
epoch:2 train_loss:0.3802225527319827 train_acc:0.8643223614072495
test_acc:0.8525079617834395
epoch:3 train_loss:0.35324129092890316 train_acc:0.8715851545842217
test_acc:0.8462380573248408
epoch:4 train_loss:0.33386651988127336 train_acc:0.8788146321961621
test_acc:0.8605692675159236
```

# 线上训练

```
from PIL import Image
from torch import nn
from torchvision import transforms
from torch.utils.data import DataLoader, Dataset
from torchvision import datasets
from torch import optim
from tqdm import tqdm
import numpy as np
from torch import save

from tensorbay import GAS
from tensorbay.dataset import Dataset as TensorBayDataset


class MNISTSegment(Dataset):
    """class for wrapping a MNIST segment."""

    def __init__(self, gas, segment_name, transform):
        super().__init__()
        self.dataset = TensorBayDataset("FashionMNIST-PyTorch", gas)
        self.segment = self.dataset[segment_name]
        self.category_to_index =
self.dataset.catalog.classification.get_category_to_index()
        self.transform = transform

    def __len__(self):
        return len(self.segment)

    def __getitem__(self, idx):
        data = self.segment[idx]
        with data.open() as fp:
            image_tensor = self.transform(Image.open(fp))

        return image_tensor,
self.category_to_index[data.label.classification.category]
```

```
ACCESS_KEY = "Accesskey-dac95e0d8e685ef4d5f8b80d51e38499"

to_tensor = transforms.ToTensor()
normalization = transforms.Normalize(mean=[0.485], std=[0.229])
my_transforms = transforms.Compose([to_tensor, normalization])

train_segment = MNISTSegment(GAS(ACCESS_KEY), segment_name="train",
transform=my_transforms)
train_dataloader = DataLoader(train_segment, batch_size=4, shuffle=True)
test_segment = MNISTSegment(GAS(ACCESS_KEY), segment_name="test",
transform=my_transforms)
test_dataloader = DataLoader(test_segment, batch_size=4, shuffle=False)
```

```
class Fashion(nn.Module):
    def __init__(self):
        super(Fashion, self).__init__()
        self.net = nn.Sequential(nn.Linear(28*28, 1024),
                                 nn.ReLU(),
                                 nn.Linear(1024, 10))
    def forward(self, x):
        x = x.view(x.shape[0], -1)
        y = self.net(x)
        return y

def evaluate_accuracy(data_iter, net):
    test_acc =[]
    for X, y in data_iter:
        test_acc.append((net(X).argmax(dim=1) == y).float().sum().item() /
y.shape[0])
    return np.mean(test_acc)

epochs = 5
net = Fashion()
optimizer = optim.SGD(net.parameters(), lr=0.1)
criterion = nn.CrossEntropyLoss()
for epoch in range(epochs):
    train_loss = []
    train_acc = []
    for x, y in tqdm(train_dataloader):
        y_pred = net(x)
        loss = criterion(y_pred, y).sum()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_loss.append(loss.item())
        train_acc.append((y_pred.argmax(dim=1)==y).sum().item() / y.shape[0])
    test_acc = evaluate_accuracy(test_dataloader, net)
    print(f'epoch:{epoch} train_loss:{np.mean(train_loss)} train_acc:
{np.mean(train_acc)} test_acc:{test_acc}')
save(net.state_dict(), 'last.pt')
```