



北京交通大学《深度学习》课件

# 实验1

# PyTorch实践与前馈神经网络



主讲教师：丛润民      助教：张禹墨

北京交通大学 《深度学习》课程组





北京交通大学《深度学习》课件

# 实验1-2 前馈神经网络实验

主讲教师：丛润民      助教：张禹墨

北京交通大学 《深度学习》课程组





## 1. 基本概念

- 人工神经元
- 激活函数
- 前馈神经网络的组成
- 优化器的使用

## 2. 实现前馈神经网络

- 手动实现前馈神经网络
- Torch.nn实现前馈神经网络

## 3. 模型调优

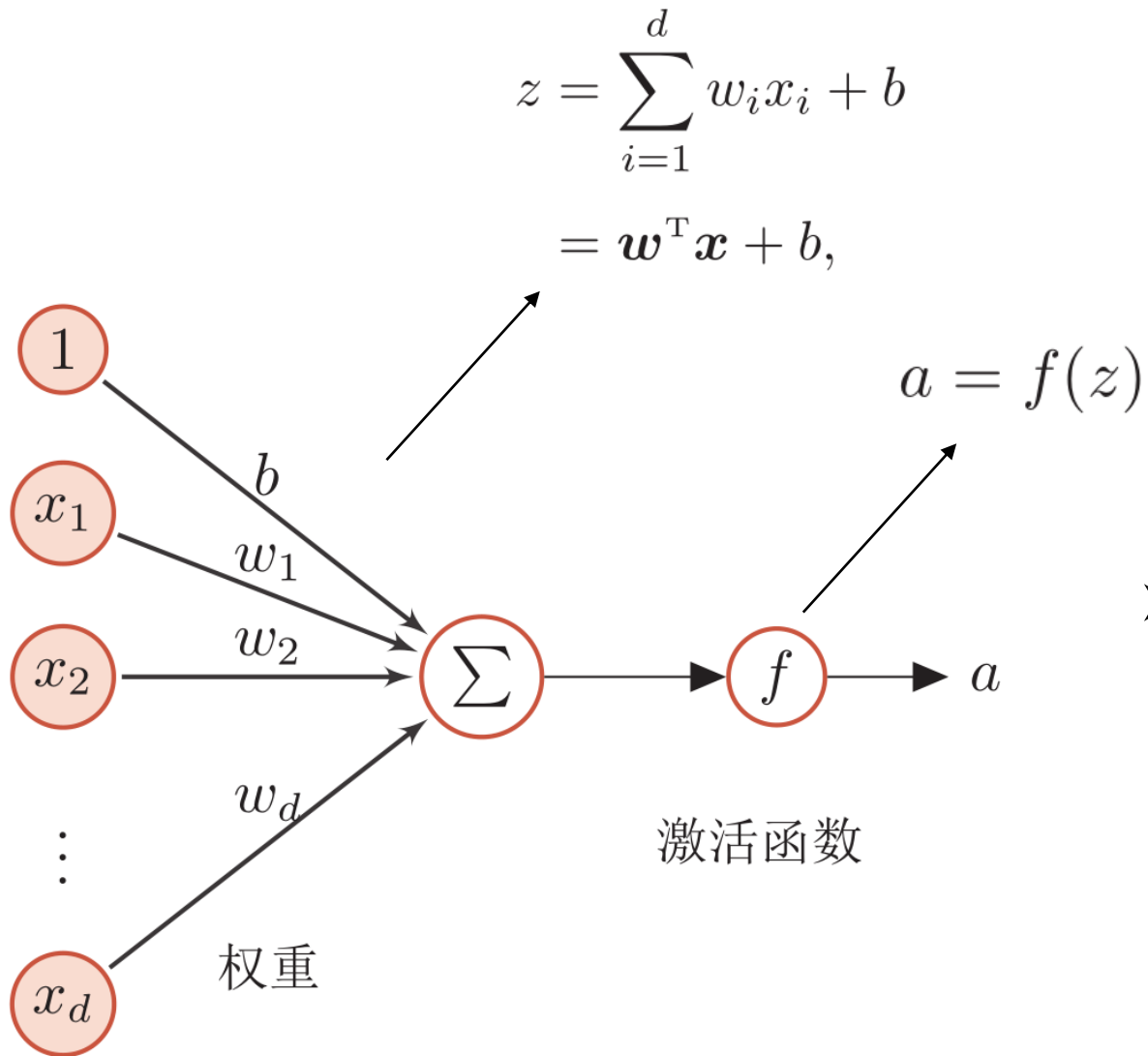
- 模型选择、过拟合、欠拟合
- 探究导致过拟合、欠拟合的因素
- 过拟合解决办法：正则化、dropout

## 4. 实验要求

- 数据集介绍
- 平台课实验要求
- 专业课实验要求



# 人工神经元



$$\mathbf{x} = [x_1, x_2, \dots, x_d]$$

$$\mathbf{w} = [w_1, w_2, \dots, w_d]$$

$b \in R$  是偏置

非线性函数 $f(\cdot)$ 被称为**激活函数**

➤ 手动实现人工神经元 (以relu激活函数为例)

```
def neurons(X, w, b):  
    z=torch.mm(X, w) + b  
    return F.relu(z)
```



## ■ 常见的激活函数

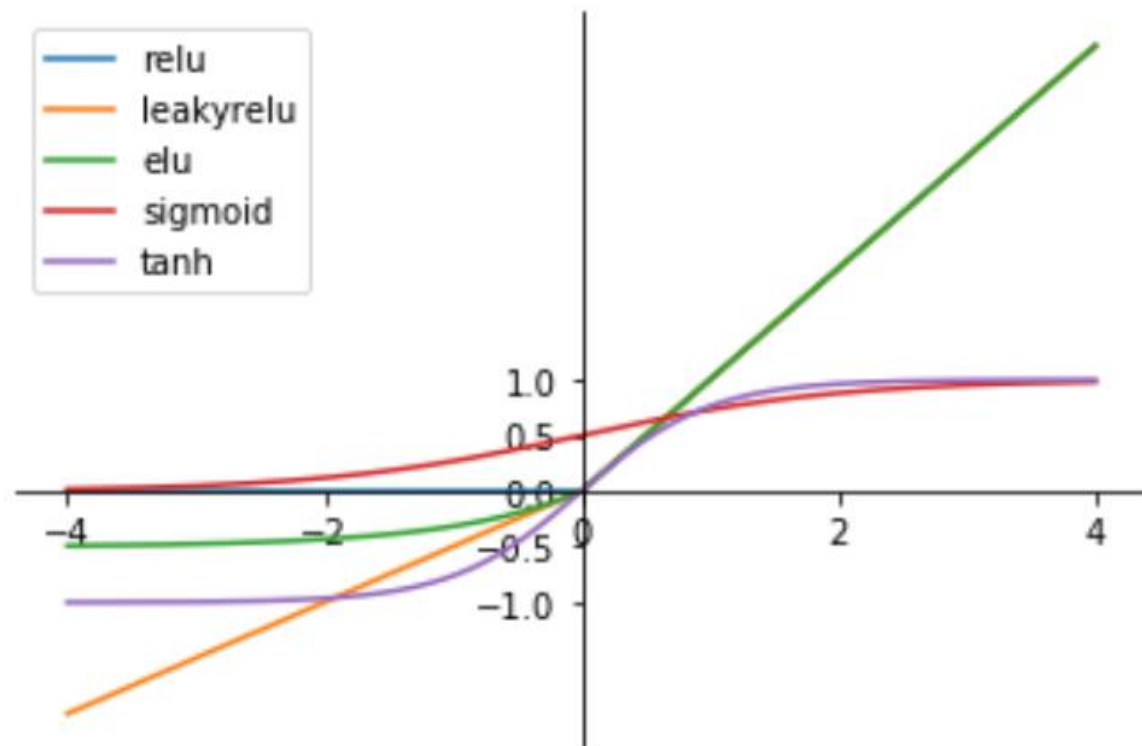
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

$$\text{ReLU}(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

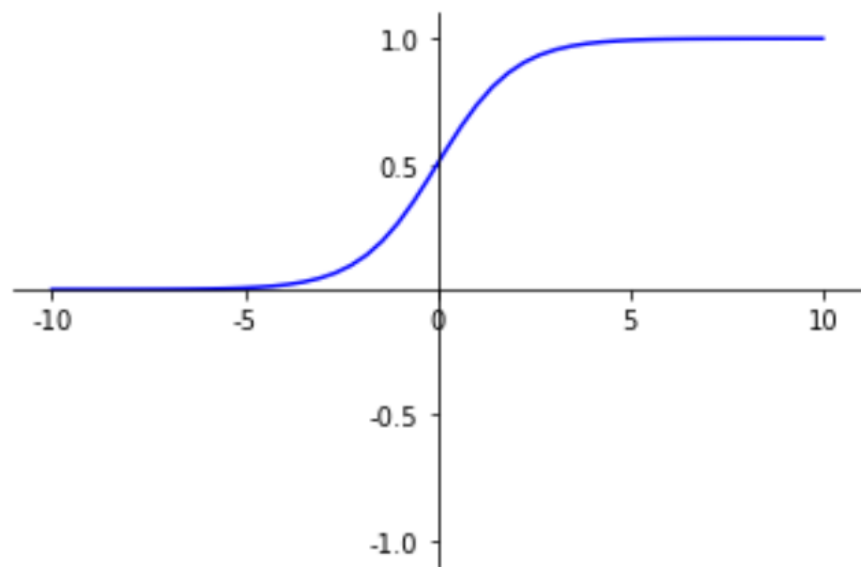
$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \gamma x & \text{if } x \leq 0 \end{cases}$$

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \gamma(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$





## ■ sigmoid激活函数



$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

### ➤ 手动实现sigmoid激活函数

```
input_x= torch.randn(4)
input_x

tensor([-0.6806,  0.8624,  1.6567,  2.3298])
```

```
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))
sigmoid(input_x)
```

```
tensor([0.3361,  0.7032,  0.8398,  0.9113])
```

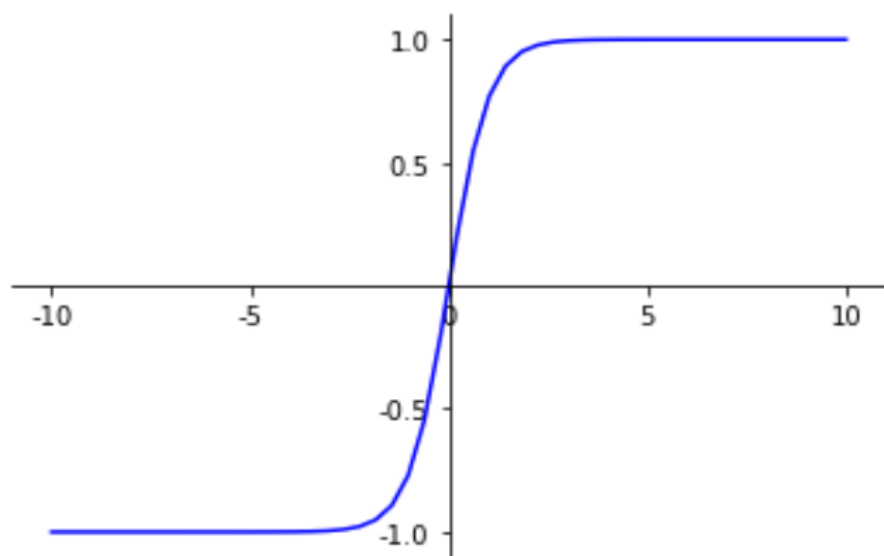
### ➤ 调用pytorch实现的sigmoid激活函数

```
output=torch.sigmoid(input_x)
print(output)
```

```
tensor([0.3361,  0.7032,  0.8398,  0.9113])
```



## ■ tanh激活函数



$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

## ➤ 手动实现tanh激活函数

```
input_x = torch.randn(4)
input_x
```

```
tensor([ 0.5387, -0.2507,  0.2622, -0.0470])
```

```
def fun_tanh(x):
    return (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))
fun_tanh(input_x)
```

```
tensor([ 0.4920, -0.2456,  0.2564, -0.0470])
```

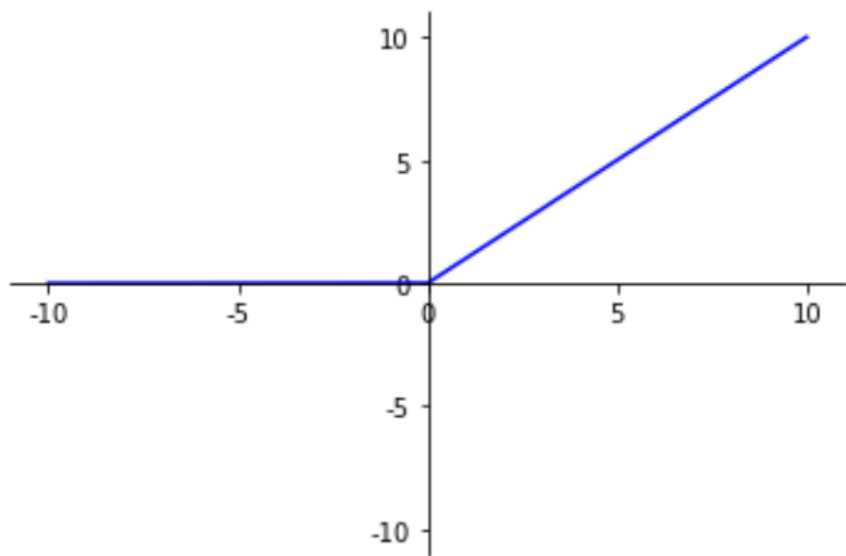
## ➤ 调用pytorch实现的tanh激活函数

```
output = torch.tanh(input_x)
print(output)
```

```
tensor([ 0.4920, -0.2456,  0.2564, -0.0470])
```



## ■ ReLU激活函数



$$\text{ReLU}(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

### ➤ 手动实现ReLU激活函数

```
input_x=torch.randn(4)
input_x
```

```
tensor([ 1.2941,  1.3352,  0.7952, -0.0587])
```

```
def fun_relu(x):
    x=np.where(x>=0, x, 0)
    return torch.tensor(x)
fun_relu(input_x)
```

```
tensor([1.2941, 1.3352, 0.7952, 0.0000])
```

### ➤ 调用pytorch实现的ReLU激活函数

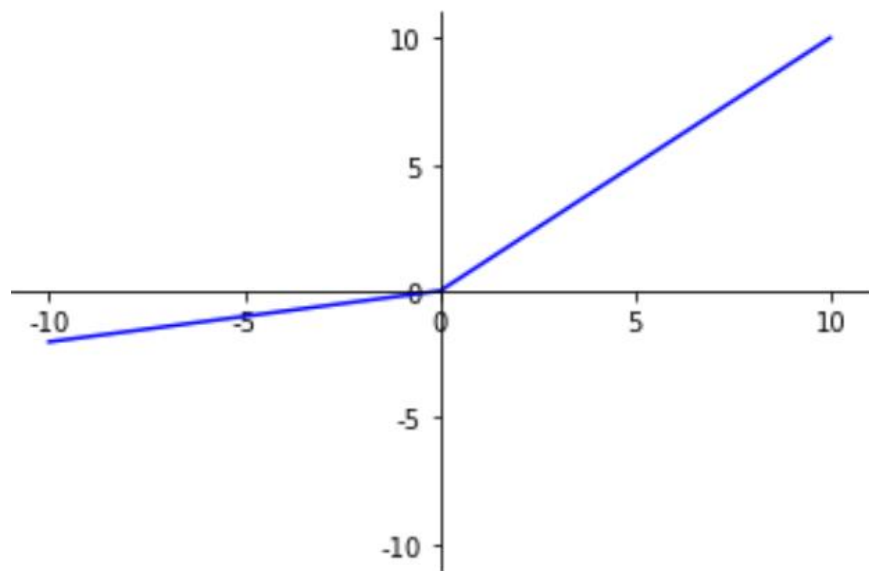
```
output=torch.nn.functional.relu(input_x)
print(output)
```

```
tensor([1.2941, 1.3352, 0.7952, 0.0000])
```





## ■ leakyReLU激活函数



$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \gamma x & \text{if } x \leq 0 \end{cases}$$

## ➤ 手动实现leakyReLU激活函数

```
input_x=torch.randn(4)
input_x
```

```
tensor([-1.0628, -0.3291,  1.2930, -0.5396])
```

```
def fun_leakyrelu(x, gamma):
    x=np.where(x>0, x, x*gamma)
    return torch.tensor(x)
fun_leakyrelu(input_x, 0.2)
```

```
tensor([-0.2126, -0.0658,  1.2930, -0.1079])
```

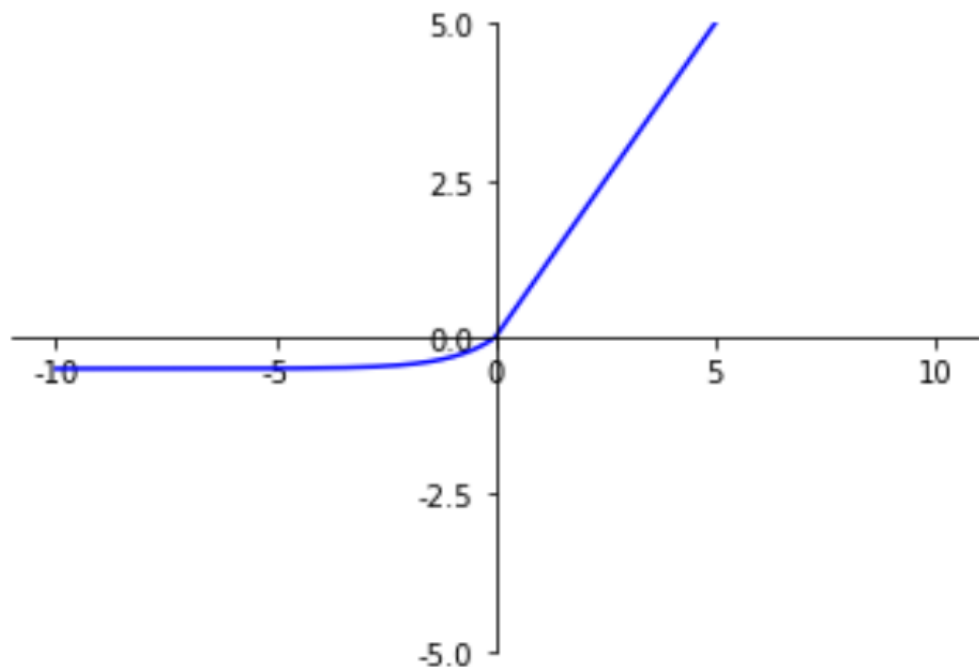
## ➤ 调用pytorch实现的leakyReLU激活函数

```
output=torch.nn.functional.leaky_relu(input_x, 0.2)
print(output)
```

```
tensor([-0.2126, -0.0658,  1.2930, -0.1079])
```



## ■ eLU激活函数



$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \gamma(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

## ➤ 手动实现eLU激活函数

```
input_x=torch.randn(4)
input_x
```

```
tensor([-0.7859,  1.2788, -0.5068, -0.7251])
```

```
def fun_elu(x, gamma):
    x=np.where(x>0, x, gamma*(np.exp(x)-1))
    return torch.tensor(x)
fun_elu(input_x, 0.2)
```

```
tensor([-0.1089,  1.2788, -0.0795, -0.1031])
```

## ➤ 调用pytorch实现的eLU激活函数

```
output=torch.nn.functional.elu(input_x, 0.2)
output
```

```
tensor([-0.1089,  1.2788, -0.0795, -0.1031])
```



## ■ 如何选择激活函数

通常来说，很少会把各种激活函数串起来在一个网络中使用的。

如果使用 ReLU，那么一定要小心设置 learning rate，而且要注意不要让网络出现很多 “dead” 神经元，如果这个问题不好解决，那么可以试试 LeakyReLU、PReLU。

最好不要用 sigmoid，可以试试 tanh，不过预期它的效果会比不上 ReLU。

**根据具体的模型，进行具体的分析和选择**



# 前馈神经网络的组成

## ■ 组成结构

- 输入神经元个数:  $d = 4$
- 隐藏层神经元个数:  $h = 5$
- 输出层神经元个数:  $q = 3$
- 给的小批量样本  $X \in R^{n*d}$
- 隐藏层的输出为  $H \in R^{n*h}$
- 输出层的输出为  $O \in R^{n*q}$

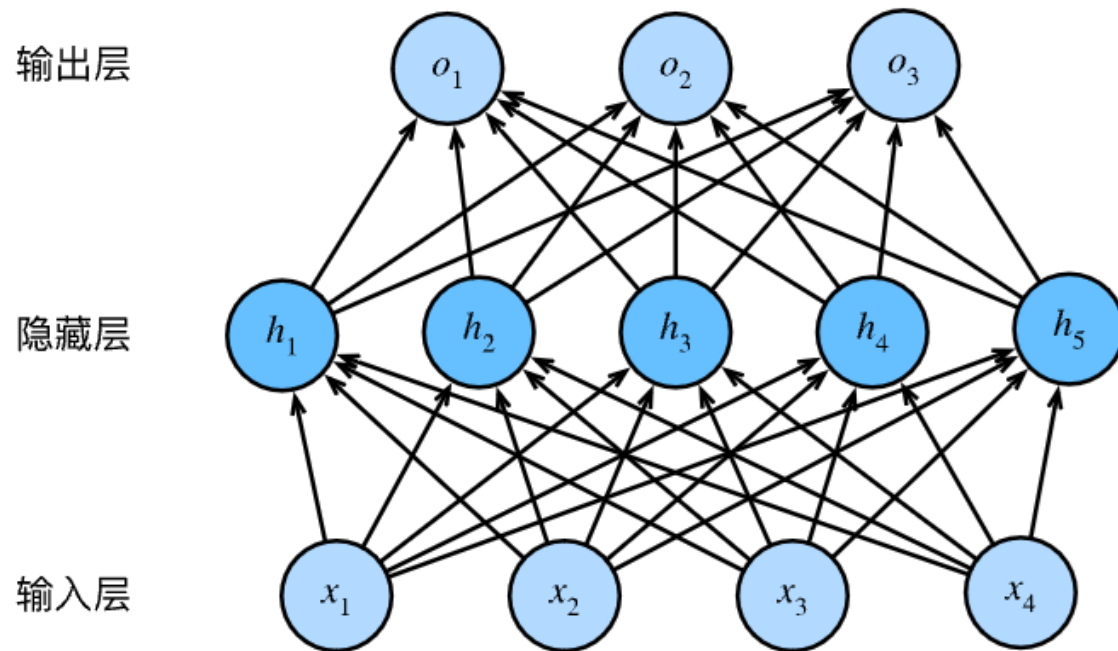
## ■ 计算公式

$$H = XW_h^T + b_h$$

$$O = HW_o^T + b_o$$

$$W_h \in R^{h*d}, b_h \in R^{1*h}$$

$$W_o \in R^{q*h}, b_o \in R^{1*q}$$



包含一个隐藏层的前馈神经网络



# 优化器的使用

## ■ 常用的优化器：SGD、AdaGrad、RMSProp等

### ➤ 手动实现SGD

```
def SGD(params, lr):  
    for param in params:  
        param.data -= lr * param.grad
```

### ➤ pytorch如何使用SGD

torch.optim实现了各种优化算法

```
optimizer = torch.optim.SGD(model.parameters(), lr)  
for input, target in dataset:  
    optimizer.zero_grad()  
    output = model(input)  
    loss = loss_fn(output, target)  
    loss.backward()  
    optimizer.step()
```

其他优化器的使用可以参考pytorch官方文档



## 1. 基本概念

- 人工神经元
- 激活函数
- 前馈神经网络的组成
- 优化器的使用

## 2. 实现前馈神经网络

- 手动实现前馈神经网络
- Torch.nn实现前馈神经网络

## 3. 模型调优

- 模型选择、过拟合、欠拟合
- 探究导致过拟合、欠拟合的因素
- 过拟合解决办法：正则化、dropout

## 4. 实验要求

- 数据集介绍
- 平台课实验要求
- 专业课实验要求



# 手动实现前馈神经网络

## ■ 数据集介绍

采用实验1中介绍的Fashion-MNIST数据集完成多分类任务。

训练集：60,000    测试集：10,000

每个样本的数据格式为：28\*28\*1（高\*宽\*通道）

类别（10类）：dress（连衣裙）、coat（外套）、sandal（凉鞋）、shirt（衬衫）、sneaker（运动鞋）、bag（包）和ankle boot（短靴）

## ➤ Fashion-MNIST数据集下载

```
import torch
import torchvision
import torchvision.transforms as transforms
```

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



# 手动实现前馈神经网络

## ➤ 批量读取数据

采用`torch.utils.data.DataLoader`读取小批量数据

```
batch_size = 64  #小批量数目
train_iter = torch.utils.data.DataLoader(mnist_train, batch_size=batch_size, shuffle = True, num_workers = 0)
#num_workers=0, 不开启多线程读取。
test_iter = torch.utils.data.DataLoader(mnist_test, batch_size = batch_size, shuffle=False, num_workers=0)
```

`batch_size`是超参数，表示一轮训练多少个样本

`shuffle`是否打乱数据，`True`表示打乱数据

`num_workers=0`表示不开启多线程读取数据





# 手动实现前馈神经网络

## ■ 模型设计（包含一个隐藏层）

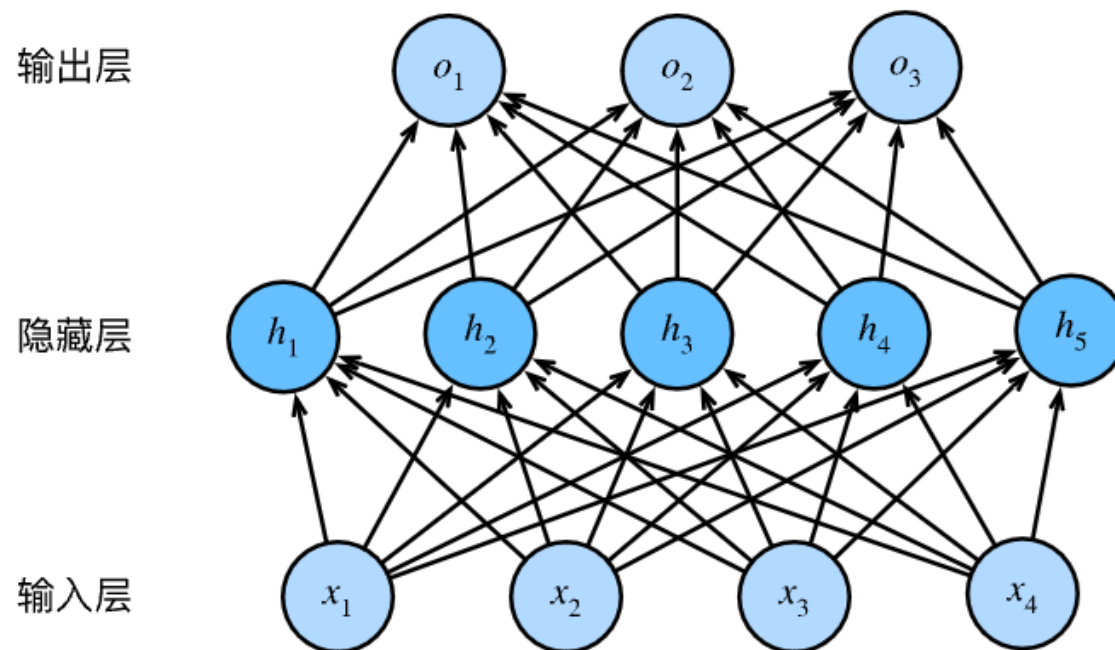
输出层神经元个数：10  
(对应10个类别)

$$W_2 \in \mathbb{R}^{10 \times 256}$$

隐藏层神经元个数：256

$$W_1 \in \mathbb{R}^{256 \times 784}$$

输入神经元个数：784  
(每个样本是 $28 \times 28 \times 1$ )





# 手动实现前馈神经网络

## ■ 代码实现

### ➤ 模型参数定义及初始化

```
##定义模型参数
num_inputs, num_outputs, num_hiddens = 784, 10, 256

W1 = torch.tensor(np.random.normal(0, 0.01, (num_hiddens, num_inputs)), dtype=torch.float)
b1 = torch.zeros(num_hiddens, dtype=torch.float)
W2 = torch.tensor(np.random.normal(0, 0.01, (num_outputs, num_hiddens)), dtype=torch.float)
b2 = torch.zeros(num_outputs, dtype=torch.float)

params = [W1, b1, W2, b2]
for param in params:
    param.requires_grad_(requires_grad=True)
```

### ➤ 定义激活函数

我们选用ReLU作为激活函数

```
def relu(x):
    x=np.where(x>0, x, 0)
    return x
```



# 手动实现前馈神经网络

## ➤ 定义模型

```
#定义模型
def net(X):
    X = X.view((-1, num_inputs))
    H = relu(torch.matmul(X, W1.t()) + b1)
    return torch.matmul(H, W2.t()) + b2
```

## ➤ 定义交叉熵损失函数

```
loss = torch.nn.CrossEntropyLoss()
```

## ➤ 定义随机梯度下降函数

```
def SGD(params, lr):
    for param in params:
        param.data -= lr * param.grad
```

## ➤ 计算模型在某个数据集上的准确率

```
def evaluate_accuracy(data_iter, net):
    acc_sum, n = 0.0, 0
    for X, y in data_iter:
        acc_sum += (net(X).argmax(dim=1) == y).float().sum().item()
        n += y.shape[0]
    return acc_sum / n
```



# 手动实现前馈神经网络

## ➤ 定义模型训练函数

```
def train(net, train_iter, test_iter, loss, num_epochs, batch_size, params=None, lr=None, optimizer=None):  
    for epoch in range(num_epochs): 外层循环控制训练轮次  
        train_l_sum, train_acc_sum, n = 0.0, 0.0, 0  
        for X, y in train_iter: 内层循环控制训练批次  
            y_hat = net(X)  
            l = loss(y_hat, y).sum() 第一步:计算模型输出和loss  
            # 梯度清零  
            if optimizer is not None:  
                optimizer.zero_grad()  
            elif params is not None and params[0].grad is not None:  
                for param in params:  
                    param.grad.data.zero_()  
            l.backward() 第二步:反向传播  
            if optimizer is None:  
                SGD(params, lr)  
            else:  
                optimizer.step()  
            train_l_sum += l.item()  
            train_acc_sum += (y_hat.argmax(dim=1) == y).sum().item() 第三步:计算准确率  
            n += y.shape[0]  
        test_acc = evaluate_accuracy(test_iter, net)  
        print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f'  
              % (epoch + 1, train_l_sum / n, train_acc_sum / n, test_acc))
```



# 手动实现前馈神经网络

## ➤ 训练模型

- 设置训练轮次为5轮
- 学习率为0.1
- 优化器采用默认的SGD函数

```
num_epochs=5 #训练轮次  
lr=0.1 #学习率  
train(net, train_iter, test_iter, loss, num_epochs, batch_size, params, lr)
```

```
epoch 1, loss 0.0313, train acc 0.510, test acc 0.650  
epoch 2, loss 0.0278, train acc 0.708, test acc 0.722  
epoch 3, loss 0.0273, train acc 0.729, test acc 0.727  
epoch 4, loss 0.0271, train acc 0.735, test acc 0.731  
epoch 5, loss 0.0270, train acc 0.739, test acc 0.737
```



# Torch.nn实现前馈神经网络

## ➤ 实现FlattenLayer层

```
class FlattenLayer(torch.nn.Module):
    def __init__(self):
        super(FlattenLayer, self).__init__()
    def forward(self, x):
        return x.view(x.shape[0], -1)
```

## ➤ 模型定义和参数初始化

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256

net = nn.Sequential(
    FlattenLayer(),
    nn.Linear(num_inputs, num_hiddens),
    nn.ReLU(),
    nn.Linear(num_hiddens, num_outputs),
)

for params in net.parameters():
    init.normal_(params, mean=0, std=0.01)
```

## ➤ 设置训练轮次、学习率、损失函数和优化器

```
num_epochs=50 #训练轮次
lr=0.01 #学习率
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr)
```

## ➤ 计算模型在某个数据集上的准确率和loss

```
def evaluate_accuracy(data_iter, net, loss):
    acc_sum, n = 0.0, 0
    test_l_sum=0.0
    for X, y in data_iter:
        acc_sum += (net(X).argmax(dim=1) == y).float().sum().item()
        l = loss(net(X), y).sum()
        test_l_sum += l.item()
        n += y.shape[0]
    return acc_sum / n, test_l_sum/n
```



# Torch.nn实现前馈神经网络

## ➤ 模型训练，保存训练集和验证集的loss

```
def train(net, mnist_train, mnist_test, loss, num_epochs, batch_size, params=None, lr=None, optimizer=None):
    train_loss=[]
    test_loss=[]
    for epoch in range(num_epochs):
        train_l_sum, train_acc_sum, n = 0.0, 0.0, 0
        for X, y in train_iter:
            y_hat = net(X)
            l = loss(y_hat, y).sum()
            # 梯度清零
            if optimizer is not None:
                optimizer.zero_grad()
            elif params is not None and params[0].grad is not None:
                for param in params:
                    param.grad.data.zero_()
            l.backward()
            if optimizer is None:
                SGD(params, lr)
            else:
                optimizer.step()
            train_l_sum += l.item()
            train_acc_sum += (y_hat.argmax(dim=1) == y).sum().item()
            n += y.shape[0]
        test_acc, test_l = evaluate_accuracy(test_iter, net, loss)
        train_loss.append(train_l_sum/n)
        test_loss.append(test_l)
        print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f'
              % (epoch + 1, train_l_sum / n, train_acc_sum / n, test_acc))
    return train_loss, test_loss
```



# Torch.nn实现前馈神经网络

## ➤ 训练结果和loss曲线绘制

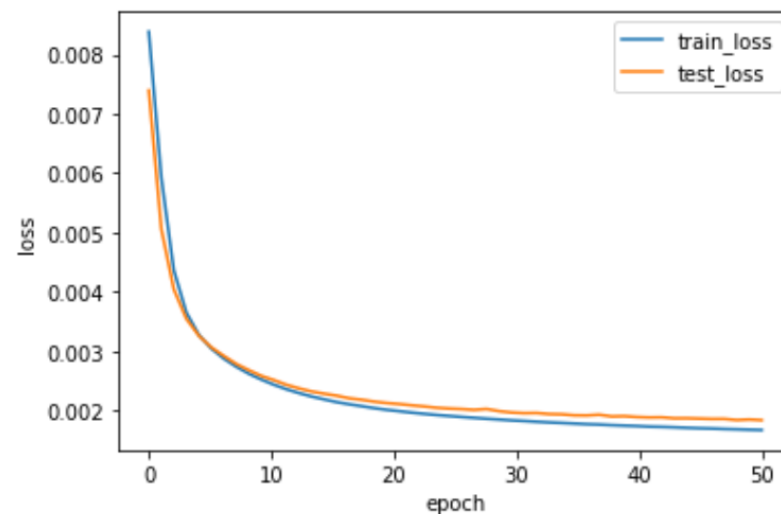
```
train_loss, test_loss = train(net, train_iter, test_iter, loss, num_epochs, batch_size, net.parameters(), lr, optimizer)
```

```
epoch 1, loss 0.0084, train acc 0.353, test acc 0.464
epoch 2, loss 0.0059, train acc 0.561, test acc 0.623
epoch 3, loss 0.0044, train acc 0.640, test acc 0.648
epoch 4, loss 0.0037, train acc 0.670, test acc 0.672
epoch 5, loss 0.0033, train acc 0.691, test acc 0.693
epoch 6, loss 0.0030, train acc 0.715, test acc 0.708
epoch 7, loss 0.0029, train acc 0.736, test acc 0.731
epoch 8, loss 0.0027, train acc 0.754, test acc 0.752
epoch 9, loss 0.0026, train acc 0.767, test acc 0.761
epoch 10, loss 0.0025, train acc 0.779, test acc 0.774
epoch 11, loss 0.0024, train acc 0.789, test acc 0.779
epoch 12, loss 0.0024, train acc 0.796, test acc 0.785
epoch 13, loss 0.0023, train acc 0.802, test acc 0.792
epoch 14, loss 0.0022, train acc 0.807, test acc 0.797
epoch 15, loss 0.0022, train acc 0.812, test acc 0.800
epoch 16, loss 0.0021, train acc 0.816, test acc 0.802
epoch 17, loss 0.0021, train acc 0.819, test acc 0.809
epoch 18, loss 0.0021, train acc 0.821, test acc 0.811
epoch 19, loss 0.0020, train acc 0.824, test acc 0.812
epoch 20, loss 0.0020, train acc 0.826, test acc 0.813
epoch 21, loss 0.0020, train acc 0.828, test acc 0.816
epoch 22, loss 0.0020, train acc 0.829, test acc 0.817
epoch 23, loss 0.0019, train acc 0.830, test acc 0.815
epoch 24, loss 0.0019, train acc 0.833, test acc 0.820
epoch 25, loss 0.0019, train acc 0.833, test acc 0.822
```

## ➤ 绘制loss曲线

```
import matplotlib.pyplot as plt

x=np.linspace(0, len(train_loss), len(train_loss))
plt.plot(x, train_loss, label="train_loss", linewidth=1.5)
plt.plot(x, test_loss, label="test_loss", linewidth=1.5)
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.show()
```







## 1. 基本概念

- 人工神经元
- 激活函数
- 前馈神经网络的组成
- 优化器的使用

## 2. 实现前馈神经网络

- 手动实现前馈神经网络
- Torch.nn实现前馈神经网络

## 3. 模型调优

- 模型选择、过拟合、欠拟合
- 探究导致过拟合、欠拟合的因素
- 过拟合解决办法：正则化、dropout

## 4. 实验要求

- 数据集介绍
- 平台课实验要求
- 专业课实验要求



## ■ K折交叉验证

- 将数据集分层采样划分为k个大小相似的互斥子集，每次用k-1个子集的并集作为训练集，余下的子集作为验证集，最终返回k个测试结果的均值，k最常用的取值是10。

## ■ 手动实现K折交叉验证

### ➤ 创建数据集

```
# 导入模块
import numpy as np
import random
# 创建一个数据集
X = torch.rand(100, 32, 32)
Y = torch.rand(100, 1)
# random shuffle
index = [i for i in range(len(X))]
random.shuffle(index)
X = X[index]
Y = Y[index]
```



# 模型选择、欠拟合、过拟合

## ➤ 获取k折交叉验证某一折的训练集和验证集

```
def get_kfold_data(k, i, X, y):  
    # 返回第 i+1 折 (i = 0 -> k-1) 交叉验证时所需要的训练和验证数据, X_train为训练集, X_valid为验证集  
    fold_size = X.shape[0] // k # 每份的个数:数据总条数/折数(组数)  
  
    val_start = i * fold_size  
    if i != k - 1:  
        val_end = (i + 1) * fold_size  
        X_valid, y_valid = X[val_start:val_end], y[val_start:val_end]  
        X_train = torch.cat((X[0:val_start], X[val_end:]), dim = 0)  
        y_train = torch.cat((y[0:val_start], y[val_end:]), dim = 0)  
    else: # 若是最后一折交叉验证  
        X_valid, y_valid = X[val_start:], y[val_start:] # 若不能整除, 将多的样本放在最后一折里  
        X_train = X[0:val_start]  
        y_train = y[0:val_start]  
  
    return X_train, y_train, X_valid, y_valid
```

分为k份后每份的个数



# 模型选择、欠拟合、过拟合

## ➤ 依次对每一折数据进行训练和测试，并计算k折平均值

```
def k_fold(k, X_train, y_train):
```

```
    train_loss_sum, valid_loss_sum = 0, 0  
    train_acc_sum, valid_acc_sum = 0, 0
```

循环K次，取平均值

```
    for i in range(k):  
        print('第', i + 1, '折验证结果')  
        data = get_kfold_data(k, i, X_train, y_train)  # 获取k折交叉验证的训练和验证数据  
        net = Net()  # 实例化模型 (某已经定义好的模型)  
        # 对每份数据进行训练  
        train_loss, val_loss, train_acc, val_acc = train(net, *data)  
  
        train_loss_sum += train_loss  
        valid_loss_sum += val_loss  
        train_acc_sum += train_acc  
        valid_acc_sum += val_acc
```

```
    print('\n', '最终k折交叉验证结果: ')
```

```
    print('average train loss:{:.4f}, average train accuracy:{:.3f}%'.format(train_loss_sum/k, train_acc_sum/k))  
    print('average valid loss:{:.4f}, average valid accuracy:{:.3f}%'.format(valid_loss_sum/k, valid_acc_sum/k))
```

```
    return
```



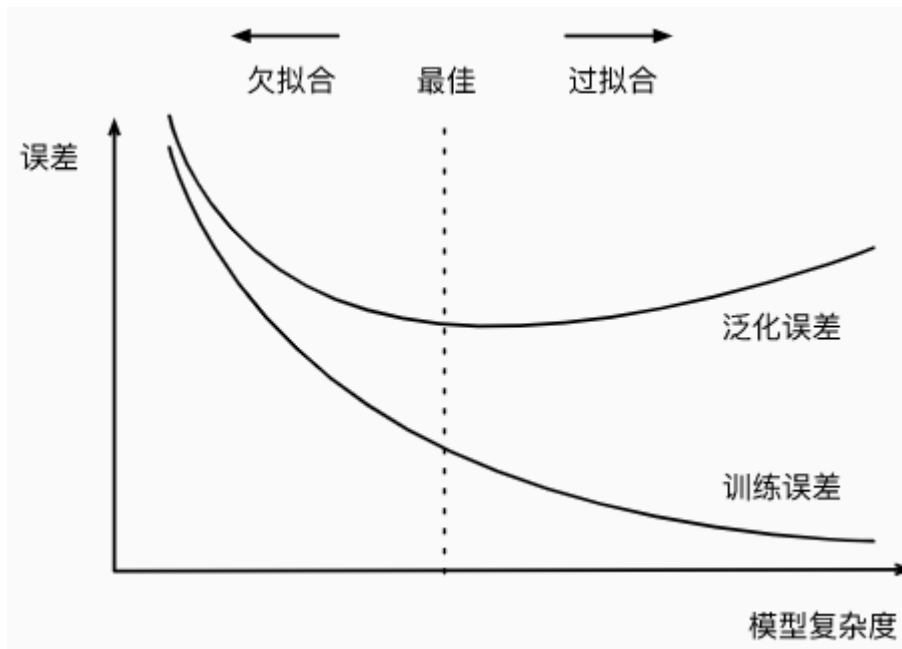
# 模型选择、欠拟合、过拟合

## ■ 过拟合

- 表现：模型在训练集上正确率**很高**，但是在测试集上正确率**很低**
- 造成原因：由于**训练数据少**和**噪声**以及**模型能力强**等原因造成的过拟合
- 解决办法：优化目标加正则项；Dropout；早停机制

## ■ 欠拟合

- 表现：模型在训练集和测试集上的正确率**都很低**
- 造成原因：由于**模型能力不足**造成的
- 解决办法：增加模型复杂度





# 多项式函数拟合实验探究影响欠拟合、过拟合的因素

## ■ 给定样本特征，使用如下的三阶多项式函数来生成样本的标签

$$y = 1.2x - 3.4x^2 + 5.6x^3 + 5 + \epsilon$$

### ➤ 设置噪声项 $\epsilon$ 服从均值为0、标准差为0.1的正态分布。训练数据集和测试数据集的样本数都设为100

```
n_train, n_test, true_w, true_b = 100, 100, [1.2, -3.4, 5.6], 5
features = torch.randn((n_train + n_test, 1))
poly_features = torch.cat((features, torch.pow(features, 2), torch.pow(features, 3)), 1)
labels = (true_w[0] * poly_features[:, 0] + true_w[1] * poly_features[:, 1]
          + true_w[2] * poly_features[:, 2] + true_b)
labels += torch.tensor(np.random.normal(0, 0.01, size=labels.size()), dtype=torch.float)
print(features[0], labels[0])

tensor([0.3509]) tensor(5.2411)
```

构造成 $[x, x^2, x^3]$ 的形式

### ➤ 定义作图函数Draw\_Loss\_Curve

```
def Draw_Loss_Curve(x_vals, y_vals, x_label, y_label, x2_vals=None, y2_vals=None,
                    legend=None, figsize=(3.5, 2.5)):
    display.set_matplotlib_formats('svg')
    plt.rcParams['figure.figsize'] = figsize
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.semilogy(x_vals, y_vals)
    if x2_vals and y2_vals:
        plt.semilogy(x2_vals, y2_vals, linestyle=':')
    plt.legend(legend)
```



# 多项式函数拟合实验探究导致欠拟合、过拟合的因素

## ➤ 模型定义和训练函数定义

```
num_epochs, loss = 100, torch.nn.MSELoss()
def fit_and_plot(train_features, test_features, train_labels, test_labels):
    #参数形状由输入数据的形状决定，由此来控制模型不同函数对原函数的拟合
    net = torch.nn.Linear(train_features.shape[-1], 1)
    #数据划分
    batch_size = min(10, train_labels.shape[0])
    dataset = torch.utils.data.TensorDataset(train_features, train_labels)
    train_iter = torch.utils.data.DataLoader(dataset, batch_size, shuffle=True)
    #训练模型
    optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
    train_ls, test_ls = [], []
    for _ in range(num_epochs):
        for X, y in train_iter:
            l = loss(net(X), y.view(-1, 1))
            optimizer.zero_grad()
            l.backward()
            optimizer.step()
        train_labels = train_labels.view(-1, 1)
        test_labels = test_labels.view(-1, 1)
        train_ls.append(loss(net(train_features), train_labels).item())
        test_ls.append(loss(net(test_features), test_labels).item())
    print('final epoch: train loss', train_ls[-1], 'test loss', test_ls[-1])
    Draw_Loss_Curve(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
                    range(1, num_epochs + 1), test_ls, ['train', 'test'])
    print('weight:', net.weight.data,
          '\nbias:', net.bias.data)
```

由传的参数来控制  
构造不同的模型



# 多项式函数拟合实验探究导致欠拟合、过拟合的因素

## ■ 使用三阶多项式函数拟合

- 使用与数据生成函数同阶的**三阶多项式函数**拟合，学习到的模型参数**接近真实值**

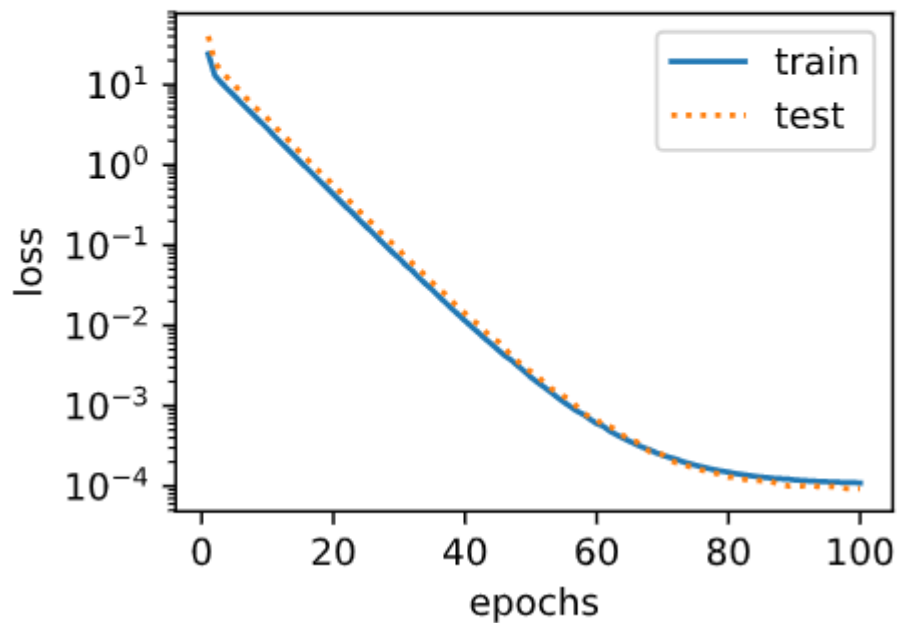
```
fit_and_plot(poly_features[:n_train, :], poly_features[n_train:, :],  
             labels[:n_train], labels[n_train:])
```

final epoch: train loss 0.00010961104999296367 test loss 9.270678856410086e-05

weight: tensor([[ 1.2058, -3.4001, 5.5984]])

bias: tensor([4.9980])

与真实值接近  $y = 1.2x - 3.4x^2 + 5.6x^3 + 5 + \epsilon$







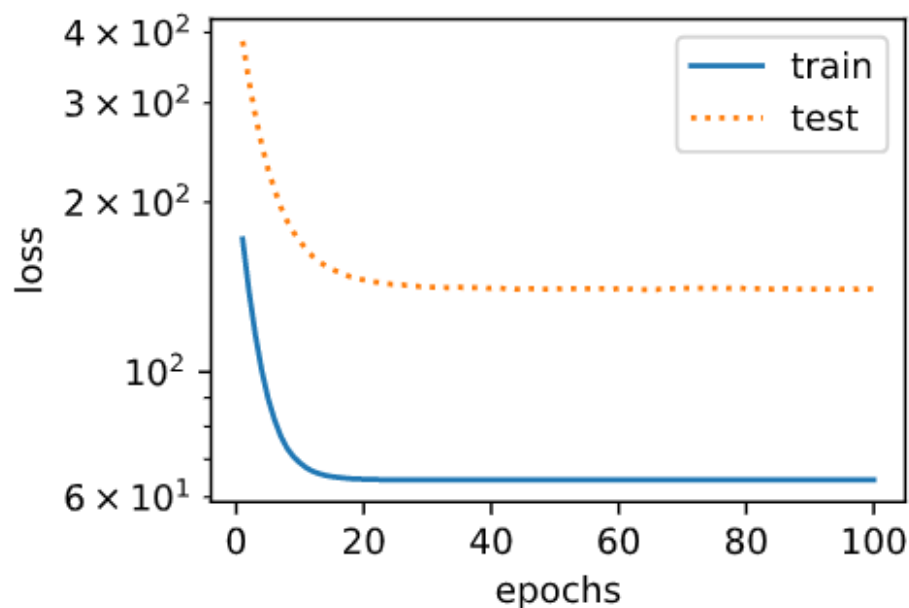
# 多项式函数拟合实验探究导致欠拟合、过拟合的因素

## ■ 使用线性函数拟合（欠拟合）

- 将模型复杂度降低：使用**线性函数**。训练集和测试集的loss在后期均很难下降，出现**欠拟合**

```
fit_and_plot(features[:n_train, :], features[n_train:, :], labels[:n_train],  
             labels[n_train:])
```

```
final epoch: train loss 64.31674194335938 test loss 140.50250244140625  
weight: tensor([[12.6037]])  
bias: tensor([1.7815])
```





# 多项式函数拟合实验探究导致欠拟合、过拟合的因素

## ■ 训练样本过少（过拟合）

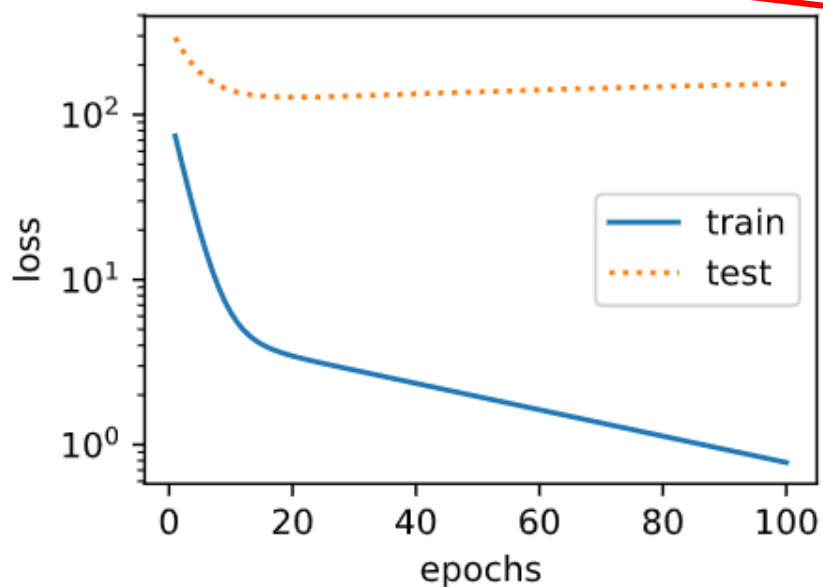
- 只使用**两个样本**来训练模型，训练集loss持续下降，测试集loss上升，出现了**过拟合**

```
fit_and_plot(poly_features[0:2, :], poly_features[n_train:, :], labels[0:2],  
             labels[n_train:])
```

final epoch: train loss 0.7790262699127197 test loss 153.86997985839844

weight: tensor([[2.0786, 1.6235, 2.4390]])

bias: tensor([2.9667])



与真实值相差较多  $y = 1.2x - 3.4x^2 + 5.6x^3 + 5 + \epsilon$



# 过拟合问题的常用方法—— $L_2$ 范数正则化

- 在模型原损失函数基础上添加 $L_2$ 范数惩罚项，通过惩罚绝对值较大的模型参数为需要学习的模型增加限制，来应对过拟合问题。带有 $L_2$ 范数惩罚项的模型的新损失函数为：

$$\ell_o + \frac{\lambda}{2n} \|\mathbf{w}\|^2$$

其中 $\mathbf{w}$ 是参数向量， $\ell_o$ 是模型原损失函数， $n$ 是样本个数， $\lambda$ 是超参数

## ■ 以高维线性回归为例

设数据样本特征的维度为 $p$ ，使用如下函数生成样本的标签

$$y = 0.05 + \sum_{i=1}^p 0.01x_i + \epsilon$$

其中噪声项 $\epsilon$ 服从均值为0、标准差为0.01的正态分布。设 $p=200$ ，设置训练集样本数为20，测试集样本数为100来引入过拟合的情况。



# 应对过拟合问题的常用方法—— $L_2$ 范数正则化

## ➤ 生成数据集

```
In [30]: %matplotlib inline
import torch
import torch.nn as nn
from torch.utils import data
import numpy as np
import sys
sys.path.append("../")
from matplotlib import pyplot as plt
from IPython import display
```

```
In [16]: n_train, n_test, num_inputs = 20, 100, 200
true_w, true_b = torch.ones(num_inputs, 1) * 0.01, 0.05
#生成数据集
features = torch.randn((n_train + n_test, num_inputs))
labels = torch.matmul(features, true_w) + true_b
labels += torch.tensor(np.random.normal(0, 0.01, size=labels.size()), dtype=torch.float)
train_features, test_features = features[:n_train, :], features[n_train:, :]
train_labels, test_labels = labels[:n_train], labels[n_train:]
print(train_features[0][:5]) #输出第一个样本特征向量的前五维的元素
print(train_labels[0])
```

$$y = 0.05 + \sum_{i=1}^p 0.01x_i + \delta$$

```
tensor([ 0.1348,  0.3261, -1.4309, -1.4814,  0.4257])
tensor([0.2408])
```



# 应对过拟合问题的常用方法—— $L_2$ 范数正则化

## ■ 手动实现 $L_2$ 范数正则化

### ➤ 定义随机初始化模型参数的函数

```
def init_params():  
    w = torch.randn((num_inputs, 1), requires_grad=True)  
    b = torch.zeros(1, requires_grad=True)  
    return [w, b]
```

### ➤ 定义 $L_2$ 范数惩罚项

```
def l2_penalty(w):  
    return (w**2).sum() / 2
```

### ➤ 定义模型

```
def linear(X, w, b):  
    return torch.mm(X, w) + b
```

### ➤ 定义均方误差

```
def squared_loss(y_hat, y):  
    # 返回的是向量, 注意: pytorch里的MSELoss并没有除以 2  
    return ((y_hat - y.view(y_hat.size())) ** 2) / 2
```

### ➤ 定义随机梯度下降函数

```
def SGD(params, lr):  
    for param in params:  
        # 注意这里参数赋值用的是param.data  
        param.data -= lr * param.grad
```



# 应对过拟合问题的常用方法—— $L_2$ 范数正则化

## ➤ 定义训练函数

```
batch_size, num_epochs, lr = 1, 100, 0.003
net, loss = linear, squared_loss
#划分数据集
dataset = torch.utils.data.TensorDataset(train_features, train_labels)
train_iter = torch.utils.data.DataLoader(dataset, batch_size, shuffle=True)
#训练模型
def fit_and_plot(lambd):
    w, b = init_params()
    train_ls, test_ls = [], []
    for _ in range(num_epochs):
        for X, y in train_iter:
            # 添加了L2范数惩罚项
            l = loss(net(X, w, b), y) + lambd * l2_penalty(w)
            l = l.sum()

            if w.grad is not None:
                w.grad.data.zero_()
                b.grad.data.zero_()
            l.backward()
            SGD([w, b], lr)
        train_ls.append(loss(net(train_features, w, b), train_labels).mean().item())
        test_ls.append(loss(net(test_features, w, b), test_labels).mean().item())
    Draw_Loss_Curve(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
                    range(1, num_epochs + 1), test_ls, ['train', 'test'])
    print('L2 norm of w:', w.norm().item())
```

添加惩罚项，用lambd  
控制惩罚权重

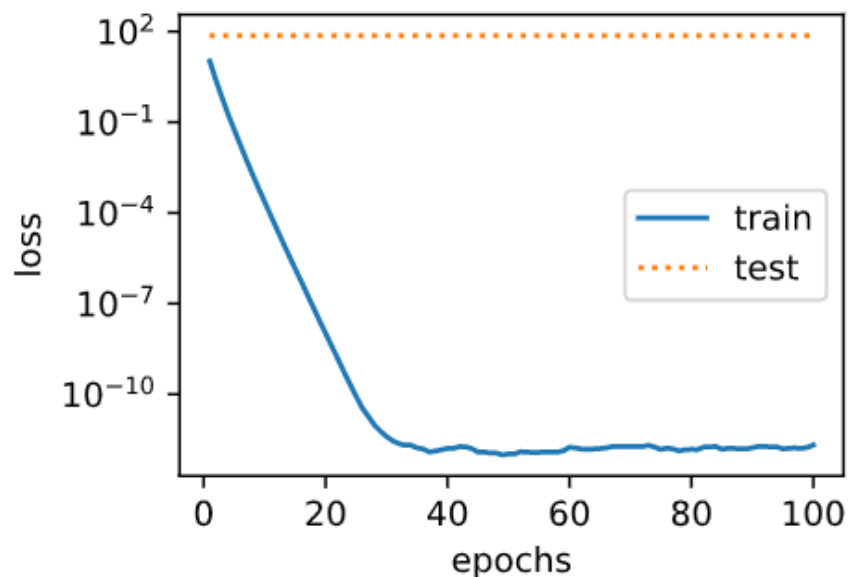


# 应对过拟合问题的常用方法—— $L_2$ 范数正则化

- $\lambda = 0$  (即不使用 $L_2$ 范数正则化) 时的实验结果, 出现了**过拟合**的现象。

```
fit_and_plot(lambd=0)
```

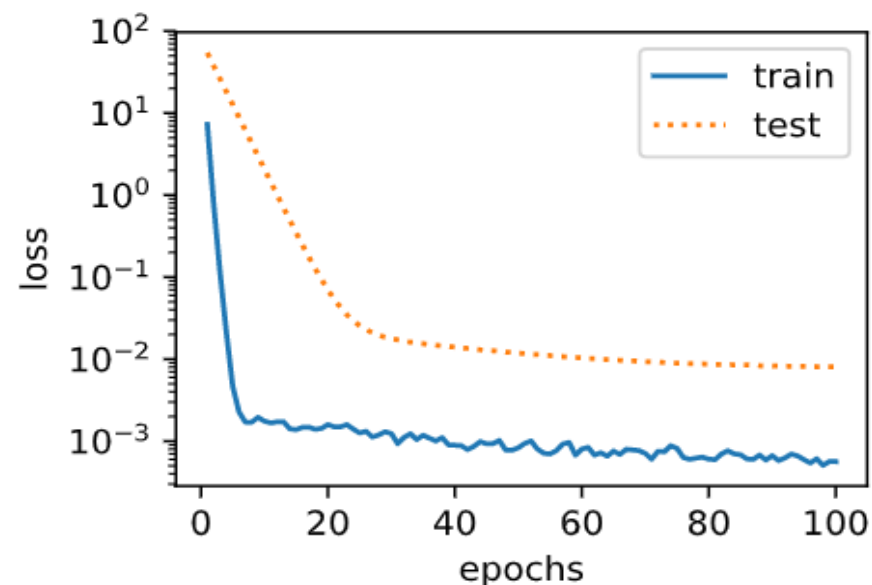
L2 norm of w: 12.559598922729492



- $\lambda = 3$  (即使用 $L_2$ 范数正则化) 时的实验结果, 一定程度的**缓解了过拟合**。同时可以看到参数 $L_2$ 范数变小, **参数更接近0**。

```
fit_and_plot(lambd=3)
```

L2 norm of w: 0.04879558086395264





# 应对过拟合问题的常用方法—— $L_2$ 范数正则化

## ■ 利用torch.optim的weight\_decay参数实现 $L_2$ 范数正则化

### ➤ 定义训练函数

```
def fit_and_plot_pytorch(wd):  
    # 对权重参数衰减。权重名称一般是以weight结尾  
    net = nn.Linear(num_inputs, 1)  
    nn.init.normal_(net.weight, mean=0, std=1)  
    nn.init.normal_(net.bias, mean=0, std=1)  
    # 使用weight_decay参数实现L2范数正则化  
    optimizer_w = torch.optim.SGD(params=[net.weight], lr=lr, weight_decay=wd)  
    optimizer_b = torch.optim.SGD(params=[net.bias], lr=lr)  
  
    train_ls, test_ls = [], []  
    for _ in range(num_epochs):  
        for X, y in train_iter:  
            l = loss(net(X), y).mean()  
            optimizer_w.zero_grad()  
            optimizer_b.zero_grad()  
  
            l.backward()  
  
            # 对两个optimizer实例分别调用step函数，从而分别更新权重和偏差  
            optimizer_w.step()  
            optimizer_b.step()  
  
            train_ls.append(loss(net(train_features), train_labels).mean().item())  
            test_ls.append(loss(net(test_features), test_labels).mean().item())  
    Draw_Loss_Curve(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',  
                    range(1, num_epochs + 1), test_ls, ['train', 'test'])  
    print('L2 norm of w:', net.weight.data.norm().item())
```



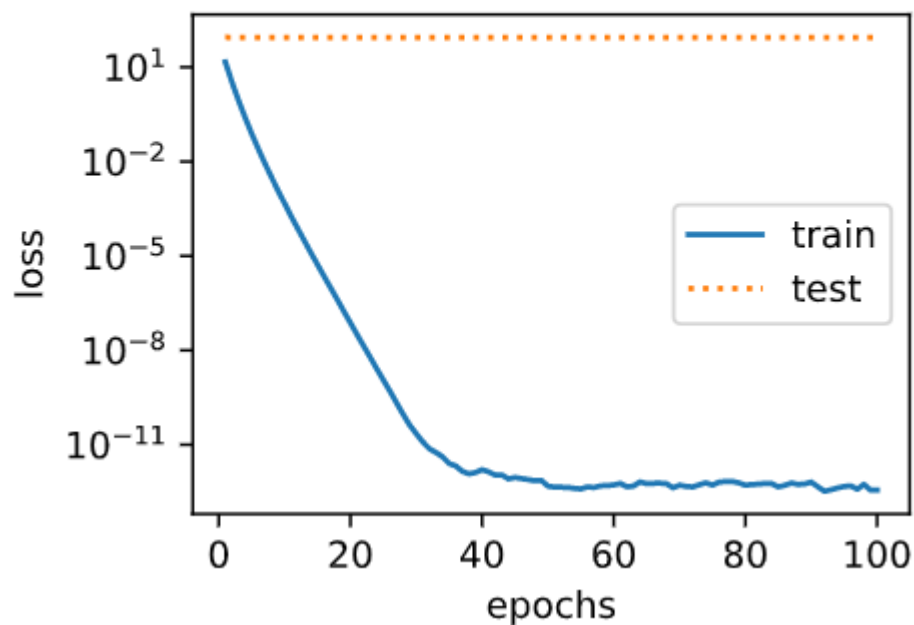


# 应对过拟合问题的常用方法—— $L_2$ 范数正则化

- $\lambda = 0$  (即不使用 $L_2$ 范数正则化) 时的实验结果, 出现了过拟合的现象。

```
fit_and_plot_pytorch(0)
```

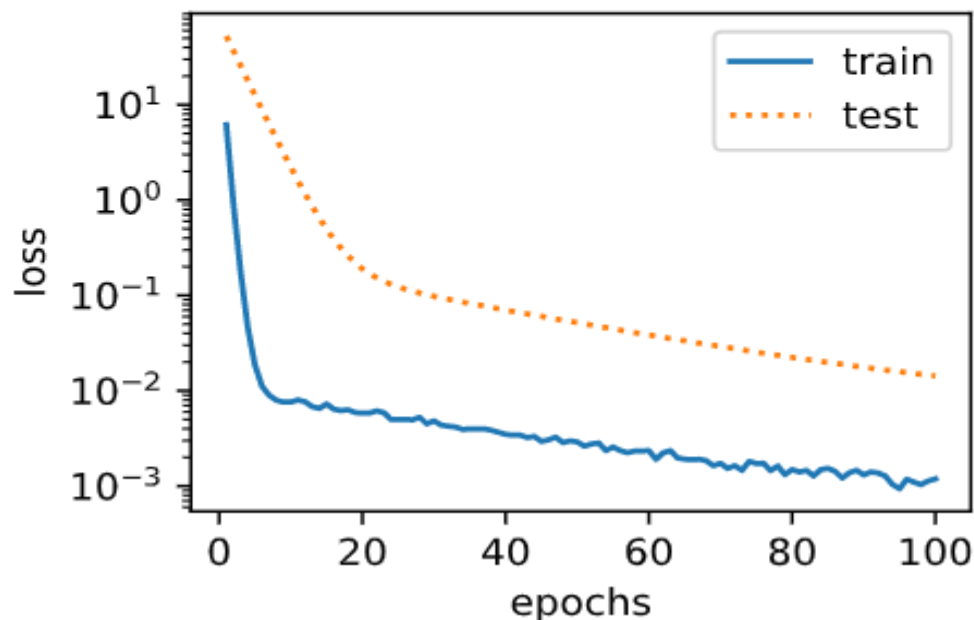
L2 norm of w: 13.343997955322266



- $\lambda = 3$  (即使用 $L_2$ 范数正则化) 时的实验结果, 一定程度的缓解了过拟合。同时可以看到参数 $L_2$ 范数变小, 参数更接近0。

```
fit_and_plot_pytorch(3)
```

L2 norm of w: 0.060846149921417236





# 应对过拟合问题的常用方法——Dropout

## ■ 手动实现dropout

以前馈神经网络为例，当使用dropout时，前馈神经网络隐藏层中的隐藏单元 $h_i$ 有一定概率被丢弃掉。

- 设丢弃概率为 $p$ ，那么有 $p$ 的概率 $h_i$ 会被清零，有 $1-p$ 的概率 $h_i$ 会除以 $1-p$ 做拉伸。由此定义进行dropout操作的函数

```
def dropout(X, drop_prob):  
    X = X.float()  
    #检查丢弃概率是否在0到1之间  
    assert 0 <= drop_prob <= 1  
    keep_prob = 1 - drop_prob  
    # 这种情况下把全部元素都丢弃  
    if keep_prob == 0:  
        return torch.zeros like(X)  
    #生成mask矩阵 (向量)  
    mask = (torch.rand(X.shape) < keep_prob).float()  
    #按照mask进行对X进行变换  
    return mask * X / keep_prob
```

- 初始化一个向量X，对X进行dropout，分别设置丢弃率为0、0.5、1，实验结果如下：

```
X = torch.arange(10).view(2, 5)  
print(dropout(X, 0), '\n')  
print(dropout(X, 0.5), '\n')  
print(dropout(X, 1))  
  
tensor([[0., 1., 2., 3., 4.],  
        [5., 6., 7., 8., 9.]])  
  
tensor([[ 0.,  2.,  0.,  0.,  8.],  
        [ 0., 12.,  0., 16., 18.]])  
  
tensor([[0., 0., 0., 0., 0.],  
        [0., 0., 0., 0., 0.]])
```



# 应对过拟合问题的常用方法——Dropout

## ➤ 定义模型参数（使用Fashion-MNIST数据集进行实验）

```
num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256

W1 = torch.tensor(np.random.normal(0, 0.01, size=(num_hiddens1, num_inputs)), dtype=torch.float, requires_grad=True)
b1 = torch.zeros(num_hiddens1, requires_grad=True)
W2 = torch.tensor(np.random.normal(0, 0.01, size=(num_hiddens2, num_hiddens1)), dtype=torch.float, requires_grad=True)
b2 = torch.zeros(num_hiddens2, requires_grad=True)
W3 = torch.tensor(np.random.normal(0, 0.01, size=(num_outputs, num_hiddens2)), dtype=torch.float, requires_grad=True)
b3 = torch.zeros(num_outputs, requires_grad=True)

params = [W1, b1, W2, b2, W3, b3]
```

## ➤ 定义使用dropout的网络模型，两个隐藏层的丢弃率分别为0.2和0.5

```
drop_prob1, drop_prob2 = 0.2, 0.5

def net(X, is_training=True):
    X = X.view(-1, num_inputs)
    H1 = (torch.matmul(X, W1.t()) + b1).relu()
    if is_training: # 如果是在训练则使用dropout
        H1 = dropout(H1, drop_prob1) # 在第一层全连接后进行dropout
    H2 = (torch.matmul(H1, W2.t()) + b2).relu()
    if is_training:
        H2 = dropout(H2, drop_prob2) # 在第二层全连接后进行dropout
    return torch.matmul(H2, W3.t()) + b3
```



# 应对过拟合问题的常用方法——Dropout

## ➤ 定义计算准确率的函数

```
def evaluate_accuracy(data_iter, net):  
    acc_sum, n = 0.0, 0  
    for X, y in data_iter:  
        acc_sum += (net(X, is_training=False).argmax(dim=1) == y).float().sum().item()  
        n += y.shape[0]  
    return acc_sum / n
```

测试时不使用dropout

## ➤ 训练模型的部分与1.5中相同，此处不再赘述。实验结果如下

```
num_epochs, lr, batch_size = 5, 0.1, 128  
loss = torch.nn.CrossEntropyLoss()  
train(net, train_iter, test_iter, loss, num_epochs, batch_size, params, lr, None)
```

```
epoch 1, loss 0.0105, train acc 0.492, test acc 0.689  
epoch 2, loss 0.0052, train acc 0.759, test acc 0.802  
epoch 3, loss 0.0042, train acc 0.810, test acc 0.827  
epoch 4, loss 0.0037, train acc 0.831, test acc 0.825  
epoch 5, loss 0.0034, train acc 0.844, test acc 0.849
```



# 应对过拟合问题的常用方法——Dropout

## ■ 利用torch.nn.Dropout层实现dropout

### ➤ 定义模型

```
class FlattenLayer(torch.nn.Module):  
    def __init__(self):  
        super(FlattenLayer, self).__init__()  
    def forward(self, x):  
        return x.view(x.shape[0], -1)
```

```
net_pytorch = nn.Sequential(  
    FlattenLayer(),  
    nn.Linear(num_inputs, num_hiddens1),  
    nn.ReLU(),  
    nn.Dropout(drop_prob1),  
    nn.Linear(num_hiddens1, num_hiddens2),  
    nn.ReLU(),  
    nn.Dropout(drop_prob2),  
    nn.Linear(num_hiddens2, 10)  
)  
  
for param in net_pytorch.parameters():  
    nn.init.normal_(param, mean=0, std=0.01)
```



# 应对过拟合问题的常用方法——Dropout

## ➤ 定义计算准确率的函数 (eval()和train()来切换模型的状态)

```
def evaluate_accuracy(data_iter, net):  
    acc_sum, n = 0.0, 0  
    for X, y in data_iter:  
        if isinstance(net, torch.nn.Module):  
            net.eval() # 评估模式, 不使用dropout  
            acc_sum += (net(X).argmax(dim=1) == y).float().sum().item()  
            net.train() # 改回训练模式  
        n += y.shape[0]  
    return acc_sum / n
```

先用eval()函数切换模式, 再进行测试

## ➤ 实验结果

```
optimizer = torch.optim.SGD(net_pytorch.parameters(), lr=0.1)  
train(net_pytorch, train_iter, test_iter, loss, num_epochs, batch_size, None, None, optimizer)
```

```
epoch 1, loss 0.0104, train acc 0.490, test acc 0.746  
epoch 2, loss 0.0051, train acc 0.764, test acc 0.801  
epoch 3, loss 0.0042, train acc 0.809, test acc 0.825  
epoch 4, loss 0.0037, train acc 0.830, test acc 0.840  
epoch 5, loss 0.0034, train acc 0.843, test acc 0.837
```



## 1. 基本概念

- 人工神经元
- 激活函数
- 前馈神经网络的组成
- 优化器的使用

## 2. 实现前馈神经网络

- 手动实现前馈神经网络
- Torch.nn实现前馈神经网络

## 3. 模型调优

- 模型选择、过拟合、欠拟合
- 探究导致过拟合、欠拟合的因素
- 过拟合解决办法：正则化、dropout

## 4. 实验要求

- 数据集介绍
- 平台课实验要求
- 专业课实验要求



# 数据集介绍——回归、二分类、多分类任务数据集

## ■ 手动生成回归任务的数据集，要求：

- 生成单个数据集。
- 数据集的大小为10000且训练集大小为7000，测试集大小为3000。
- 数据集的样本特征维度p为500，且服从如下的高维线性函数：
$$y = 0.028 + \sum_{i=1}^p 0.0056x_i + \delta$$

## ■ 手动生成二分类任务的数据集，要求：

- 共生成两个数据集。
- 两个数据集的大小均为10000且训练集大小为7000，测试集大小为3000。
- 两个数据集的样本特征x的维度均为200，且分别服从均值互为相反数且方差相同的正态分布。
- 两个数据集的样本标签分别为0和1。

## ■ MNIST手写体数据集介绍：

- 该数据集包含60,000个用于训练的图像样本和10,000个用于测试的图像样本。
- 图像是固定大小(28x28像素)，其值为0到1。为每个图像都被平展并转换为784(28 \* 28)个特征的一维numpy数组。





# 数据集介绍——多分类任务数据集

## ➤ MNIST数据集下载和读取：

#下载MNIST手写数字数据集

```
train_dataset = torchvision.datasets.MNIST(root='./Datasets/MNIST', train=True, transform= transforms.ToTensor(), download=True)
test_dataset = torchvision.datasets.MNIST(root='./Datasets/MNIST', train=False, transform= transforms.ToTensor())
```

Downloading <http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz>

Downloading <http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz>

Downloading <http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz>

Downloading <http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz>

Processing...

Done!

```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=False)
```

```
for X, y in train_loader:
    print(X.shape, y.shape)
    break
```

torch.Size([32, 1, 28, 28]) torch.Size([32])



# 平台课实验要求

## ■ 手动实现前馈神经网络解决上述回归、二分类、多分类任务

- 分析实验结果并绘制训练集和测试集的loss曲线

## ■ 利用torch.nn实现前馈神经网络解决上述回归、二分类、多分类任务

- 分析实验结果并绘制训练集和测试集的loss曲线

## ■ 在多分类实验的基础上使用至少三种不同的激活函数

- 对比使用不同激活函数的实验结果

## ■ 在多分类任务实验中手动实现一种解决过拟合的方法 ( $L_2$ 正则化或dropout)

- 探究使用前后模型效果的变化, 并完成下述任务中的一项
- $L_2$ 正则化: 探究惩罚项的权重对实验结果的影响 (可用loss曲线进行展示)
- dropout: 探究不同丢弃率对实验结果的影响 (可用loss曲线进行展示)



# 实验提交要求

## ■ 实验报告

- 需要完成之前的任务要求，写清楚实验结果，对实验结果的分析和相关实验参数设置

## ■ 实验报告中涉及的代码

- 完整代码，确保能够跑出实验报告中的结果（允许由于网络随机性出现的误差）

## ■ 实验报告中使用的数据集

- 无需提交的情况
  - 代码中包含生成数据的函数
  - 按照PPT中所述下载数据集，未进行更改

## ■ 所有文件放到一个文件夹内压缩

- 名称例如：20120332\_张三\_实验1.zip

## ■ 8月2号晚上24:00之前提交

### 文件部署情况应如下：

- 20120332\_张三\_实验1
  - ▼ ● 实验1-1
    - 代码和数据集放这里
  - 实验1-2
    - 代码和数据集放这里
- 20120332\_张三\_实验1.docx(实验报告)