一、Linux

二、pytorch基础

1、Tensor

张量和NumPy数组通常可以共享相同的底层内存，从而无需复制数据（参见与NumPy的桥接）。

```
import torch
import numpy as np
```

（1）初始化

```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)

np_array = np.array(data)
x_np = torch.from_numpy(np_array)

x_ones = torch.ones_like(x_data) #创建形状相同，但元素都是1的张量
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) #创建形状相同，但元素是随机生成的浮点数，dtype覆盖原来的类型
print(f"Random Tensor: \n {x_rand} \n")


shape = (2,3,)#张量的形状
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)#这三个生成的张量元素都为浮点型
print(f"Random Tensor: \n {rand_tensor} \n")
print(f"Ones Tensor: \n {ones_tensor} \n")
print(f"Zeros Tensor: \n {zeros_tensor}")
```

(2) 张量的属性

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")

//*Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu*//
```

(3) 对张量的操作

```
if torch.accelerator.is_available():
    tensor = tensor.to(torch.accelerator.current_accelerator())#检查当前是
否有加速器可用，如果可用，则将张量 tensor 转移到当
                                    #前可用的加速器上进行处理

tensor = torch.ones(4, 4)
print(f"First row: {tensor[0]}")#第一行
print(f"First column: {tensor[:, 0]}")#第一列
print(f"Last column: {tensor[..., -1]}")#最后一列，...代表省略部分维度
tensor[:,1] = 0#将第二列改为0


t1 = torch.cat([tensor, tensor, tensor], dim=1)#按照第一维连接张量。在
PyTorch中，维度是从0开始计数的，所以第一维是列。
tensor: ([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
t1([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.]])


y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)#矩阵乘法

y3 = torch.rand_like(y1)
torch.matmul(tensor, tensor.T, out=y3)#y1,y2,y3结果相同


z1 = tensor * tensor
z2 = tensor.mul(tensor)#元素乘法

z3 = torch.rand_like(tensor)
```

```
torch.mul(tensor, tensor, out=z3)#z1,z2,z3结果相同


agg = tensor.sum()
agg_item = agg.item()#单个元素的张量可以通过item()转换为python数值
print(agg_item, type(agg_item))


print(f"{tensor} \n")
tensor.add_(5)
print(tensor)#每个元素加5
```

2、数据集

（1）加载训练集和测试集

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt


training_data = datasets.FashionMNIST(
    root="data",#数据存储路径
    train=True,#true为训练数据，false为测试数据
    download=True,#指定路径找不到，就下载
    transform=ToTensor()#将加载的数据转为张量
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

（2）数据集可视化

```
labels_map = {
    0: "T-Shirt",
    1: "Trouser",
    2: "Pullover",
    3: "Dress",
    4: "Coat",
    5: "Sandal",
    6: "Shirt",
    7: "Sneaker",
```

```
    8: "Bag",
    9: "Ankle Boot",
}#标签映射
figure = plt.figure(figsize=(8, 8))#创建8*8英寸的窗口
cols, rows = 3, 3
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(training_data), size=(1,)).item()#随机选择一个样本的索引
    #torch.randint生成随机整数，size=(1,)指定生成单个整数的张量
    img, label = training_data[sample_idx]#获取随机选取的图像及对应的标签
    figure.add_subplot(rows, cols, i)#在窗口添加子图，前两个指定行列数，最后一个指定位置编号
    plt.title(labels_map[label])#设置子图名称
    plt.axis("off")#隐藏坐标轴
    plt.imshow(img.squeeze(), cmap="gray")#去掉多余的维度，以灰度显示
plt.show()
```

(3) 自定义数据集

```
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):#Dataset是基类，用于创建自定义数据集
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
    #第一个是csv文件路径，第二个是存储图像文件的目录路径，第三个是对图像预处理，第四个是对标签预处理，三四为可选
        self.img_labels = pd.read_csv(annotations_file)#读取csv文件并存储在DataFrame中
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)#返回样本数量

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        #path.join用于安全地连接目录和文件名，生成完整文件路径；iloc基于整数位置进行数据选取；0代表第一个元素，
        image = read_image(img_path)#读取图像文件
        label = self.img_labels.iloc[idx, 1]#获取标签
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

(4) 数据加载器

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
#第二个参数指定每个小批量包含64个样本，shuffle==true代表在每个训练周期开始时随机打乱数据，以减少模型过拟合。
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)



# Display image and label.
train_features, train_labels = next(iter(train_dataloader))
#从train_dataloader中获取下一个批次的特征和标签，next获取迭代器的下一个元素，iter创建数据加载器的迭代器
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()
print(f"Label: {label}")
```

(5) 数据的转化

```
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda

ds = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),#ToTensor 将 PIL 图像或 NumPy ndarray 转换为 FloatTensor，并将图像的像素
    #强度值缩放到 [0., 1.] 范围内。
    target_transform=Lambda(lambda y: torch.zeros(10, dtype=torch.float).scatter_(0, torch.tensor(y), value=1))
    #使用 Lambda 定义一个匿名函数，将标签 y 转换为独热编码张量；torch.zeros(10, dtype=torch.float): 创建一个具有10个元素的零张
    #量，表示10个类别；scatter_(0, torch.tensor(y), value=1): 将标签对应的索引位置设置为1，实现独热编码；scatter_() 是
    #PyTorch 的一个内置函数，用于将指定值分配到张量中的指定位置；0 表示沿着第0维（即行）进行操作；torch.tensor(y) 将标签 y 转换为
    #张量，用作索引；value=1 指定要分配的值为1
)
```

3、构建神经网络

```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms


device = torch.accelerator.current_accelerator().type if
torch.accelerator.is_available() else "cpu"
print(f"Using {device} device")


class NeuralNetwork(nn.Module):#nn.Module 是所有神经网络模块的基类
    def __init__(self):
        super().__init__()#调用父类 nn.Module 的构造函数，确保父类的初始化逻辑被
执行
        self.flatten = nn.Flatten()#将输入的多维张量展平成一维张量。通常用于将二
维图像数据转换为一维输入向量
        self.linear_relu_stack = nn.Sequential(#定义一个顺序容器
nn.Sequential，用于将多个层按顺序组合在一起
            nn.Linear(28*28, 512),#定义了线性层（全连接层），输入大小是28*28，
输出大小是512，执行线性变换 y=Wx+b
            nn.ReLU(),#一种常用的激活函数，定义为f(x)=max(0,x)，用于引入非线性
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):#定义前向传播方法，描述数据如何通过网络进行计算
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork().to(device)
print(model)



不要直接调用model.forward()

X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)#指定在第 1 维度上进行 Softmax 操作，
将每个输出向量转换为概率
y_pred = pred_probab.argmax(1)#返回指定维度上最大值的索引。这里是在第 1 维度上找
到概率最大的类别索引，即预测的类别
print(f"Predicted class: {y_pred}")
```

```
print(f"Model structure: {model}\n\n")#模型结构

for name, param in model.named_parameters():#遍历模型所有参数，获取名称和值
    print(f"Layer: {name} | Size: {param.size()} | Values : {param[:2]}
\n")
```

Softmax 是一种常用的激活函数，通常用于多分类问题的输出层。它将一组未归一化的数值（logits）转换为概率分布。Softmax 的输出值介于 0 和 1 之间，且所有输出值的和为 1。

数学定义

对于一个输入向量 $\mathbf{z} = [z_1, z_2, \ldots, z_n]$，Softmax 函数的输出 $\sigma(\mathbf{z})$ 定义为：

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

其中 e 是自然常数，z_i 是输入向量中的元素。

4、使用torch.autograd进行自动微分

（1）基本用法

```
import torch

x = torch.ones(5)  # input tensor
y = torch.zeros(3)  # expected output
w = torch.randn(5, 3, requires_grad=True)#requires_grad=True表示需要对这个
张量计算梯度，方便后续的反向传播
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)#计算二元
分类问题的损失，并自动应用sigmoid函数到输入z
loss.backward()#计算损失函数相对于模型参数的梯度
print(w.grad)
print(b.grad)
```

(2) 禁用梯度更新

```
z = torch.matmul(x, w)+b
print(z.requires_grad)

with torch.no_grad():#禁用梯度跟踪
    z = torch.matmul(x, w)+b
print(z.requires_grad)



z = torch.matmul(x, w)+b
z_det = z.detach()#detach()禁用梯度跟踪
print(z_det.requires_grad)
```

(3) 张量梯度和雅各比积

```
inp = torch.eye(4, 5, requires_grad=True)#生成4*5的单位矩阵
out = (inp+1).pow(2).t()#生成out矩阵
out.backward(torch.ones_like(out), retain_graph=True)#计算 out 对 inp 的梯
度。使用 torch.ones_like(out) 作为权重
print(f"First call\n{inp.grad}")
out.backward(torch.ones_like(out),
retain_graph=True)#torch.ones_like(out)用来指定元素
print(f"\nSecond call\n{inp.grad}")
inp.grad.zero_()#将梯度清零
out.backward(torch.ones_like(out), retain_graph=True)
print(f"\nCall after zeroing gradients\n{inp.grad}")

//*First call
tensor([[4., 2., 2., 2., 2.],
        [2., 4., 2., 2., 2.],
        [2., 2., 4., 2., 2.],
        [2., 2., 2., 4., 2.]])

Second call
tensor([[8., 4., 4., 4., 4.],
        [4., 8., 4., 4., 4.],
        [4., 4., 8., 4., 4.],
        [4., 4., 4., 8., 4.]])

Call after zeroing gradients
tensor([[4., 2., 2., 2., 2.],
        [2., 4., 2., 2., 2.],
        [2., 2., 4., 2., 2.],
        [2., 2., 2., 4., 2.]])*//
```

5、优化模型参数

（1）加载数据和建模型

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)

train_dataloader = DataLoader(training_data, batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)

class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork()
```

(2) 超参数

```
#超参数是可调整的参数，可以让你控制模型的优化过程。不同的超参数值会影响模型训练和收敛速度
learning_rate = 1e-3#每个批次/迭代时更新模型参数的幅度。较小的值会导致学习速度较慢，而较大的值可能导致训练过程中出现不可预测的行为
batch_size = 64#在更新参数之前通过网络传播的数据样本数量
epochs = 5#数据集被遍历的次数
```

(3) 优化循环

```
def train_loop(dataloader, model, loss_fn, optimizer):#数据加载器、模型、损失函数、优化器
    size = len(dataloader.dataset)#数据集总大小
    model.train()#将模型调为训练模式
    for batch, (X, y) in enumerate(dataloader):
    #使用 enumerate 对数据加载器进行迭代，得到每个批次的索引 batch 以及数据 X 和标签 y。

        pred = model(X)
        loss = loss_fn(pred, y)#计算预测结果 pred 与真实标签 y 之间的损失值

        loss.backward()
        optimizer.step()#使用计算出的梯度来更新模型参数
        optimizer.zero_grad()#重置梯度以防止累加

        if batch % 100 == 0:
            loss, current = loss.item(), batch * batch_size + len(X)#len返回批次中的样本量
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")


def test_loop(dataloader, model, loss_fn):

    model.eval()#将模型设置为评估模式
    size = len(dataloader.dataset)
    num_batches = len(dataloader)#获取数据集加载器的批次数量
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
```

```
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:
{test_loss:>8f} \n")
```

（4）优化器

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

6、保存和加载模型

```
import torch
import torchvision.models as models

#保存
model = models.vgg16(weights='IMAGENET1K_V1')
#这是 PyTorch 提供的一个函数，用于创建 VGG16 模型。VGG16 是一种深度卷积神经网络，常
用于图像分类任务
#使用在 ImageNet 数据集上预训练的权重（版本 IMAGENET1K_V1）
torch.save(model.state_dict(), 'model_weights.pth')
#model.state_dict(): 这是一个包含模型所有参数（权重和偏置）的字典
#'model_weights.pth': 这是保存模型权重的文件名，格式为 .pth，通常用于存储
PyTorch 模型的权重


#加载
model = models.vgg16()
model.load_state_dict(torch.load('model_weights.pth', weights_only=True))
#weights_only=True 被认为是在加载权重时的最佳实践
model.eval()
#在推理之前一定要调用 model.eval() 方法，以将 dropout 和批归一化层设置为评估模式。
如果不这样做，将会导致推理结果不一致


#直接保存模型
torch.save(model, 'model.pth')
model = torch.load('model.pth', weights_only=False)#加载模型时为false
```

7、基于cnn的mnist分类

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)

train_dataloader = DataLoader(training_data, batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)

class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv_stack = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.flatten = nn.Flatten()
        self.linear_stack = nn.Sequential(
            nn.Linear(64 * 7 * 7, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.conv_stack(x)
        x = self.flatten(x)
        logits = self.linear_stack(x)
        return logits
```

```
model = NeuralNetwork()

def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)

    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * batch_size + len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")


def test_loop(dataloader, model, loss_fn):
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) ==
y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:
{test_loss:>8f} \n")

learning_rate = 1e-3
batch_size = 64

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 5

for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
```

```
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

1、分析nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)

二维卷积层，1是输入图像通道数（例如，灰度图像有1个通道，RGB图像有3个通道）；32是输出通道数，卷积层生成的特征图数量。每个输出通道对应一个卷积核；kernel_size是卷积核大小，卷积核是一个 3×3 的矩阵，用于扫描输入图像；stride步幅：卷积核在输入图像上滑动的步长；padding填充：在输入图像的边缘填充一圈像素，以保持输出的空间尺寸。

（1）卷积运算：卷积核（3×3矩阵）在输入图像上滑动。每次滑动时，卷积核的值与输入图像对应位置的值相乘，然后求和，得到一个输出值。

（2）填充：填充1意味着在输入图像的四周加一圈0值像素。这样做是为了在卷积后保持输出图像的空间大小与输入相同。

（3）输出计算：

设输入图像大小为 H, W，则经过卷积后的输出大小为：

$$\text{Output Height} = \frac{H + 2 \times \text{padding} - \text{kernel\_size}}{\text{stride}} + 1$$

$$\text{Output Width} = \frac{W + 2 \times \text{padding} - \text{kernel\_size}}{\text{stride}} + 1$$

例如，对于输入大小 $28, 28$，使用3×3卷积核、步幅1、填充1，输出大小为：

$$\text{Output Height} = \frac{28 + 2 \times 1 - 3}{1} + 1 = 28$$

$$\text{Output Width} = \frac{28 + 2 \times 1 - 3}{1} + 1 = 28$$

（4）多个卷积核：该层有32个卷积核，每个核生成一个输出通道。最终输出大小为32,28,2832,28,28，即32个特征图。

（5）作用：

- 提取输入图像中的局部特征。
- 不同的卷积核可以学习到不同的特征，如边缘、纹理等。
- 通过增加输出通道数，可以捕捉更多的特征信息。

2、分析nn.MaxPool2d(kernel_size=2, stride=2)

二维最大池化层，kernel_size是池化核大小，stride步幅：池化窗口在输入特征图上滑动的步长。

（1）池化操作：最大池化在输入特征图上应用一个 2×2 的窗口。在每个窗口内，选择最大值作为输出。通过这种方式，池化层有效地缩小了特征图的尺寸。

（2）输出计算：

设输入特征图大小为 H, W，经过池化后的输出大小为：

$$\text{Output Height} = \frac{H - \text{kernel\_size}}{\text{stride}} + 1$$
$$\text{Output Width} = \frac{W - \text{kernel\_size}}{\text{stride}} + 1$$

例如，对于输入大小 $28, 28$，使用2×2池化核、步幅2，输出大小为：

$$\text{Output Height} = \frac{28-2}{2} + 1 = 14$$
$$\text{Output Width} = \frac{28-2}{2} + 1 = 14$$

（3）池化的作用：

- **降维**: 通过减少特征图的空间尺寸，降低计算复杂度。
- **特征提取**: 保留重要特征（最大值），忽略不重要的细节。
- **增强不变性**: 对位置和小幅变形具有更强的鲁棒性。

8、基于rnn的AG_news分类

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchtext.datasets import AG_NEWS
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator

# 加载数据集
train_iter, test_iter = AG_NEWS(root='.data', split=('train', 'test'))
#train_iter通常是一个迭代器或生成器，提供顺序访问数据的方式，不能通过索引直接访问
#training_data提供对数据的索引访问（如training_data[i]），可以直接获取单个数据样本及其标签，通常需要与DataLoader结合使用来进行批处#理
# 创建词汇表
tokenizer = get_tokenizer('basic_english')
#返回一个分词函数，用于将文本字符串分解为单词或标记。这里使用的是基础英语分词器
vocab = build_vocab_from_iterator(map(lambda x: tokenizer(x[1]),
train_iter), specials=["<unk>"])
#map: 是一个内置函数，用于将指定的函数应用到可迭代对象的每个元素上
#lambda x: tokenizer(x[1]): 是一个匿名函数，用于从train_iter的每个元素中提取文本部分（假设是第二个元素x[1]），并对其进行分词
#整体作用: 对train_iter中的每个数据样本，提取文本部分并进行分词，生成一个标记迭代器。这个迭代器可以用于进一步处理，比如构建词汇表
#添加特殊标记<unk>用于未知词
vocab.set_default_index(vocab["<unk>"])
#将所有未在词汇表中找到的单词映射到<unk>的索引。这确保了在处理未知单词时不会引发错误
# 数据加载器
def collate_batch(batch):
```

```
    label_list, text_list, lengths = [], [], []
    for (_label, _text) in batch:
        label_list.append(_label - 1)
        #标签减去1，以便从0开始索引（通常深度学习模型要求标签从0开始）
        processed_text = torch.tensor(vocab(tokenizer(_text)),
dtype=torch.int64)
        text_list.append(processed_text)
        lengths.append(processed_text.size(0))
    return torch.tensor(label_list), nn.utils.rnn.pad_sequence(text_list,
batch_first=True), torch.tensor(lengths)
    #nn.utils.rnn.pad_sequence(text_list, batch_first=True): 对文本张量进行
填充，使其具有相同长度，batch_first=True表示批次      #维度在第一个维度
train_dataloader = DataLoader(list(train_iter), batch_size=64,
shuffle=True, collate_fn=collate_batch)
test_dataloader = DataLoader(list(test_iter), batch_size=64,
collate_fn=collate_batch)
#shuffle=True: 在每个epoch开始时打乱数据顺序，以提高模型的泛化能力
# 定义RNN模型
class RNNModel(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, output_dim):
    #vocab_size: 词汇表的大小，用于定义嵌入层的输入维度；embed_dim: 嵌入维度，表示
词向量的大小；hidden_dim: 隐藏层维度，表示 RNN      #隐藏状态的大小；output_dim:
输出维度，通常对应于分类任务中的类别数量
        super(RNNModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        #有一个大小为vocab_size*embed_dim的嵌入矩阵，把词汇表的索引转化为嵌入向量
        self.rnn = nn.RNN(embed_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x, lengths):
        x = self.embedding(x)
        packed_input = nn.utils.rnn.pack_padded_sequence(x, lengths,
batch_first=True, enforce_sorted=False)
        #根据 lengths 信息，将输入张量中的序列打包，去除填充部分。打包后的数据可以更
高效地传递给 RNN 层。
        packed_output, hidden = self.rnn(packed_input)
        logits = self.fc(hidden[-1])
        return logits

# 初始化模型、损失函数和优化器
vocab_size = len(vocab)
embed_dim = 64
hidden_dim = 128
output_dim = 4

model = RNNModel(vocab_size, embed_dim, hidden_dim, output_dim)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

```
# 训练和测试循环
def train_loop(dataloader, model, loss_fn, optimizer):
    model.train()
    for batch, (labels, texts, lengths) in enumerate(dataloader):
        pred = model(texts, lengths)
        loss = loss_fn(pred, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            print(f"loss: {loss.item():>7f}  [{batch *
len(labels):>5d}/{len(dataloader.dataset):>5d}]")

def test_loop(dataloader, model, loss_fn):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for labels, texts, lengths in dataloader:
            pred = model(texts, lengths)
            correct += (pred.argmax(1) ==
labels).type(torch.float).sum().item()
            total += labels.size(0)

    accuracy = correct / total
    print(f"Test Accuracy: {accuracy * 100:.2f}%")

# 运行训练和测试
epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

分析self.rnn = nn.RNN(embed_dim, hidden_dim, batch_first=True)

输入向量x_i：在时间步 t 的输入向量；隐藏状态h_t:在时间步 t 的隐藏状态，包含了当前时间步的输入信息以及之前时间步的记忆

$$h_t = \sigma(W_{ih}x_t + W_{hh}h_{t-1} + b_h)$$

- W_{ih}: 输入到隐藏层的权重矩阵。
- W_{hh}: 隐藏层到隐藏层的权重矩阵（循环连接）。
- b_h: 隐藏层的偏置向量。
- σ: 激活函数（通常使用tanh或ReLU）。

特点：RNN通过隐藏状态的循环连接，能够记忆和处理序列数据中的上下文信息。

三、DeepLearning

1、Regression（回归）

（1）不同类型的函数

Regression（回归）：函数产生一个特定输出（如预测明天的气温）

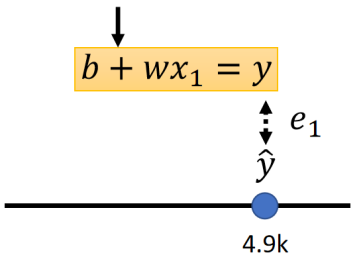
Classification（分类）：在给定的范围内，函数选出正确的类型

Structured Learning（结构化学习）：创造了一些有结构的东西（图片、文件等）

（2）基本过程

1、Function with Unknown Parameters：基于经验，构造出带有未知数的参数

2、Define Loss from Training Data：



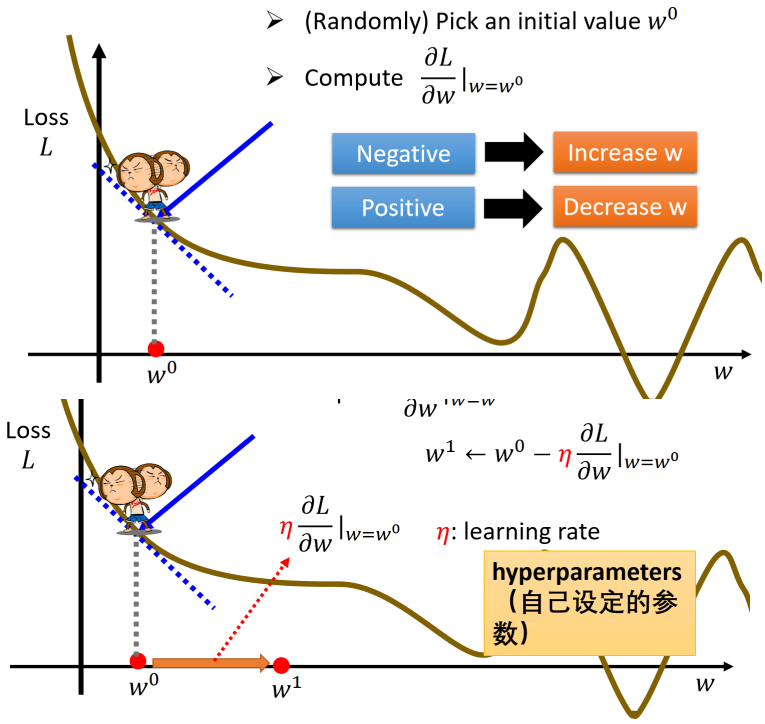
Loss: $L = \frac{1}{N}\sum_n e_n$

$e = |y - \hat{y}|$ L is mean absolute error (MAE)

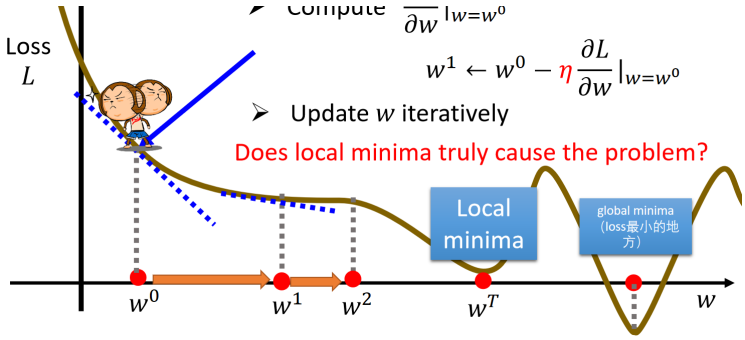
$e = (y - \hat{y})^2$ L is mean square error (MSE)

If y and $\hat{y}$ are both probability distributions ⟶ Cross-entropy

3、Optimization：Gradient Descent（梯度下降）



梯度下降可能存在的问题：



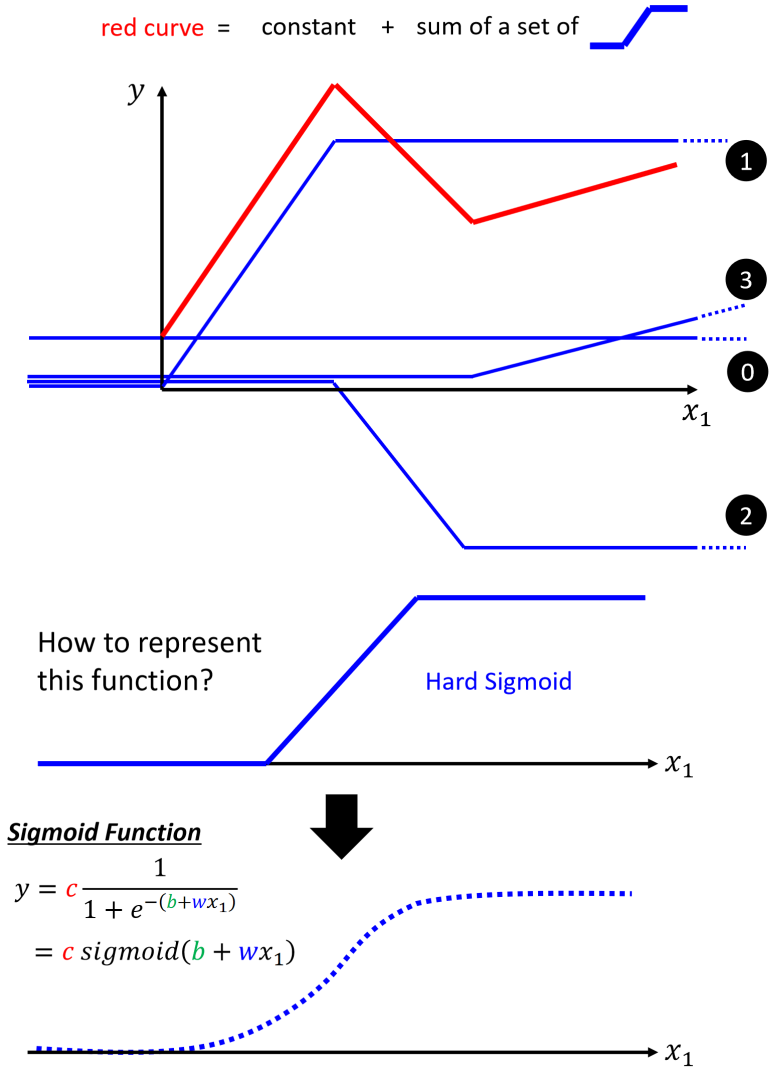
可能找到的是极小点而不是最小点

（3）Sigmoid函数

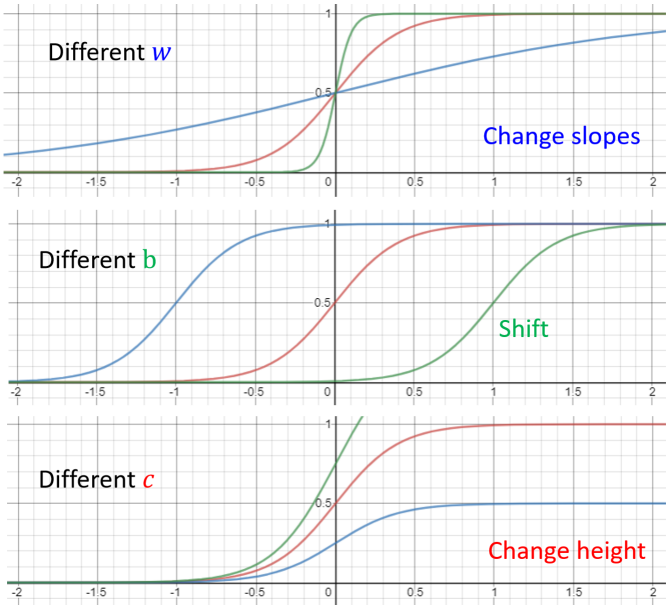
线性模型：

$$y = b + \sum_{j=1} w_j x_j$$

线性模型存在局限性（model bias），需要更好的模型



改变c,b,w的值可以得到不同的曲线



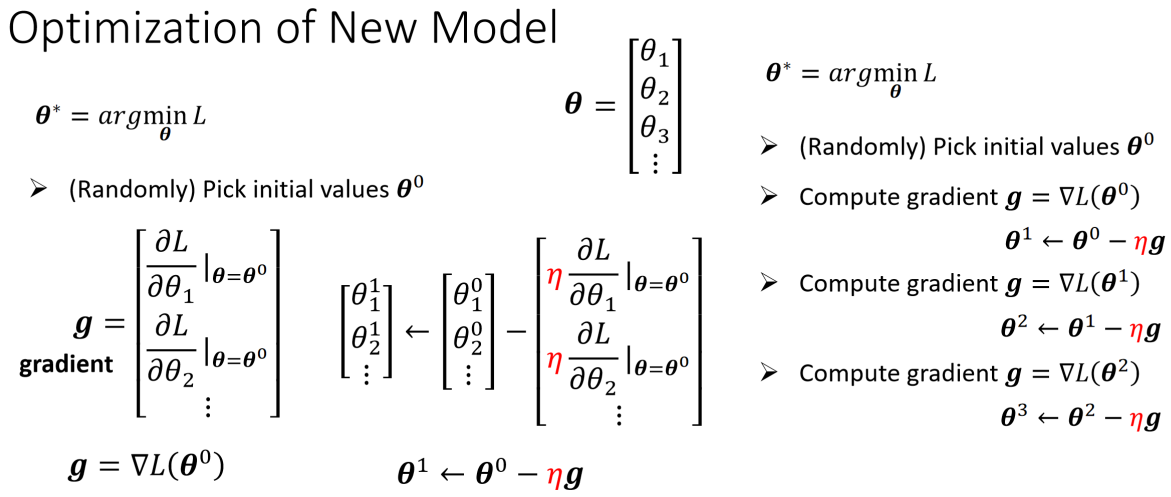
将线性模型和Sigmoid结合可以得到

$$y = b + \sum_i c_i sigmoid(b_i + \sum_j w_{ij} x_j)$$

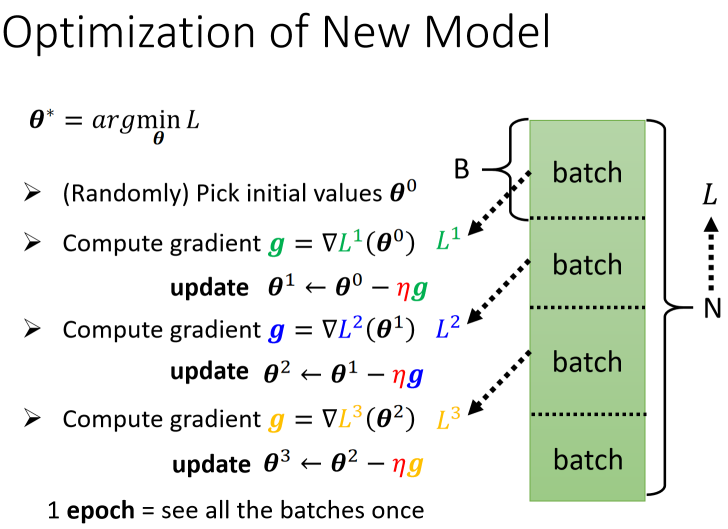
化为矩阵形式为

$$y = b + c^T \sigma(b + Wx)$$

将所有不确定的参数组合为θ向量，然后进行优化



在实践中，我们使用每个batch来计算损失函数并更新参数

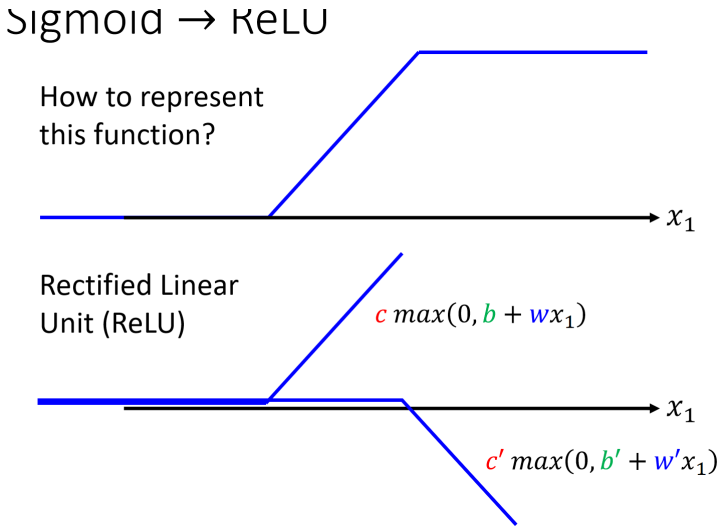


(4) RELU

可以将上面的函数式用RELU替换

$$y = b + \sum_{2i} c_i max(0, b_i + \sum_j w_{ij} x_j)$$

也同样可以构造出相同的函数曲线



神经网络层数越多，对训练集的loss就越小，那为什么不一直增加层数呢?

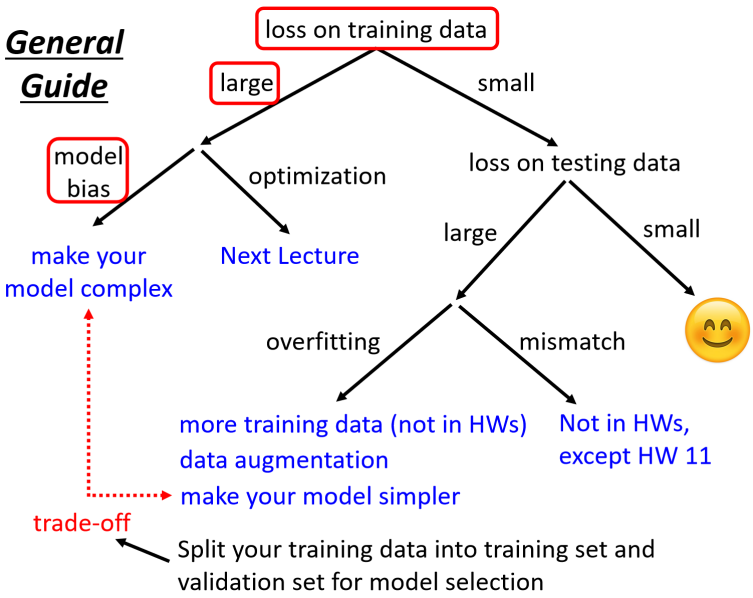
	1 layer	2 layer	3 layer	4 layer
2017 – 2020	0.28k	0.18k	0.14k	0.10k
2021	0.43k	0.39k	0.38k	0.44k

Better on training data, worse on unseen data

Overfitting

可以看到，随着层数的增加，对测试集的loss反而增加，出现了过拟合的情况。

2、模型优化未达到预期的解决方案



(1) model bias

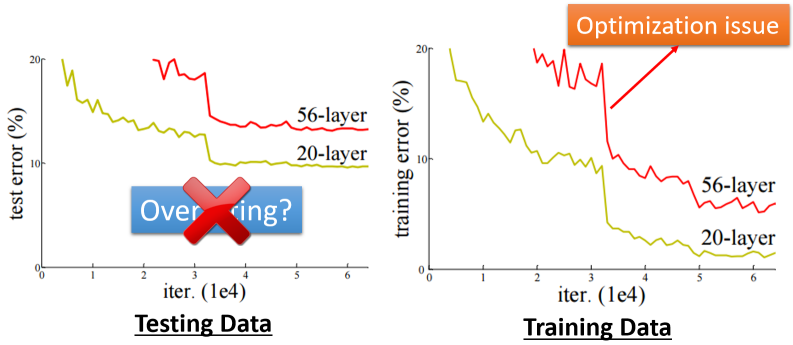
如果模型太过简单，就会导致现在的函数无法描述当前的问题，即使反复地调整参数。例如，一个线性函数无法描述一个二次函数。

解决方法：（1）more features：如果当前的预测结果只和一部分的数据集有关（例如当天的播放量只与前一天的播放量有关），那么可以引入更多的数据（加入前七天的播放量）。

（2）Deep Learning (more neurons, layers)：可以加入更多更复杂的函数（层）

(2) Optimization Issue

首先，先学会怎么判断模型是model bias还是optimization issue：如果模型一比模型二更复杂，但是loss反而更大，那就是optimization issue的问题。



观察上图，56-layer的模型比20-layer的模型层数多，但loss更大，就是optimization的问题。

(3) overfitting

在训练集上loss很小，在测试集上loss很大，这种情况就是overfitting。

An extreme example

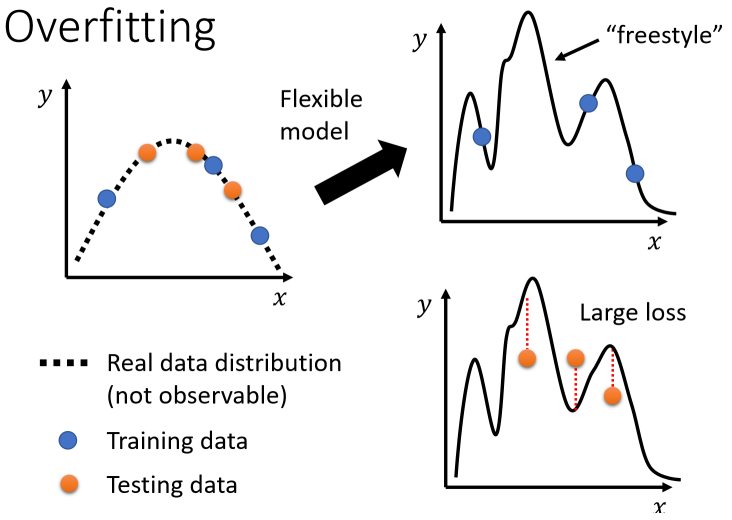
Training data: $\{(x^1, \hat{y}^1), (x^2, \hat{y}^2), ..., (x^N, \hat{y}^N)\}$

$$f(x) = \begin{cases} \hat{y}^i & \exists x^i = x \\ random & otherwise \end{cases}$$

Less than useless ...

例如上图的函数，如果x属于训练集，那么函数值就是训练集的结果，否则就是随机数。这个函数在训练集的loss为0，但在测试集的loss很大，就是典型的overfitting的问题。

overfitting的原因是模型灵活性太高（参数多，网络复杂）

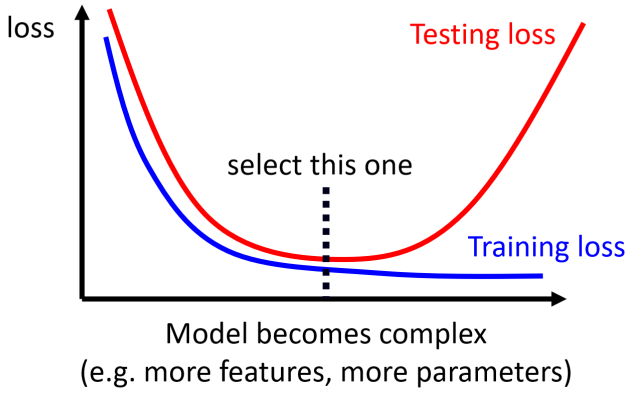


解决方法：（1）更多的训练数据

（2）Data augmentation（假设要处理图像，就把图像放大。镜像等增加数据集）

（3）constrained model（给模型增加限制）：①Less parameters, sharing parameters ②Less features ③Early stopping ④Regularization ⑤Dropout

但是限制不能过度，否则会回到model bias



尽量找到testing loss最小的值

（4）mismatch

训练集和测试集大为不同

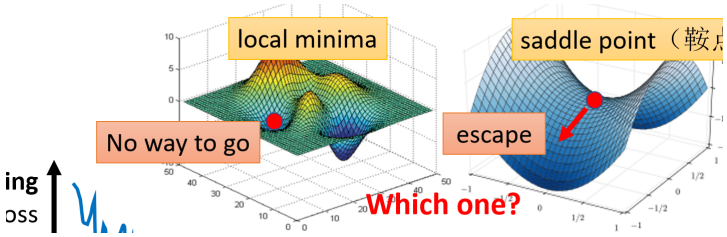
3、梯度下降到达瓶颈的情况

（1）两种情况

当梯度趋近于0时，到达critical point。此时要分辨是处在local minima还是saddle point。

local minima:局部最小值，周围都是比它大的值。

saddle point:鞍点，周围既有比它大的点，也有比它小的点。



（2）分辨方法

首先根据泰勒级数展开，我们可以得到在θ=θ'附近的$L(\theta)$的估计值：

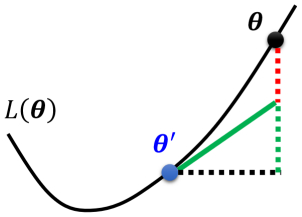
$$L(\theta) \approx L(\theta') + (\theta - \theta')^T \mathbf{g} + \frac{1}{2}(\theta - \theta')^T \mathbf{H}(\theta - \theta')$$

Gradient g is a ***vector***

$$g = \nabla L(\theta') \quad g_i = \frac{\partial L(\theta')}{\partial \theta_i}$$

Hessian H is a ***matrix***

$$H_{ij} = \frac{\partial^2}{\partial \theta_i \partial \theta_j} L(\theta')$$



在critical point的时候，g=0，所以$L(\theta)$就剩两项

Hessian

At critical point:

$$L(\theta) \approx L(\theta') + \frac{1}{2}(\theta - \theta')^T H (\theta - \theta') \qquad v^T H v$$

For all v

$v^T H v > 0$ ➡ Around θ': $L(\theta) > L(\theta')$ ➡ **Local minima**

= H is positive definite = All eigen values are positive. ⬆

For all v

$v^T H v < 0$ ➡ Around θ': $L(\theta) < L(\theta')$ ➡ **Local maxima**

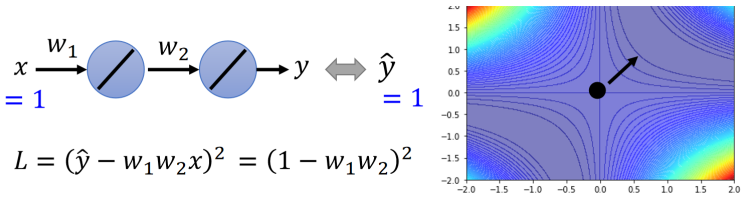
= H is negative definite = All eigen values are negative. ⬆

Sometimes $v^T H v > 0$, sometimes $v^T H v < 0$ ➡ **Saddle point**

Some eigen values are positive, and some are negative. ⬆

因此，如果H是正定矩阵，处于local minima;H是负定矩阵，处于local maxima;如果都不是，处于saddle point。

如果处于saddle point，求黑塞矩阵H的特征向量可以告诉我们优化的方向。例子如下：



$$L = (\hat{y} - w_1 w_2 x)^2 = (1 - w_1 w_2)^2$$

$$\frac{\partial L}{\partial w_1} = 2(1 - w_1 w_2)(-w_2)$$

$$\frac{\partial L}{\partial w_2} = 2(1 - w_1 w_2)(-w_1)$$

Critical point: $w_1 = 0, w_2 = 0$

$$H = \begin{bmatrix} 0 & -2 \\ -2 & 0 \end{bmatrix} \quad \lambda_1 = 2, \lambda_2 = -2$$

Saddle point

$\lambda_2 = -2$ Has eigenvector $u = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

Update the parameter along the direction of u

You can escape the saddle point and decrease the loss.

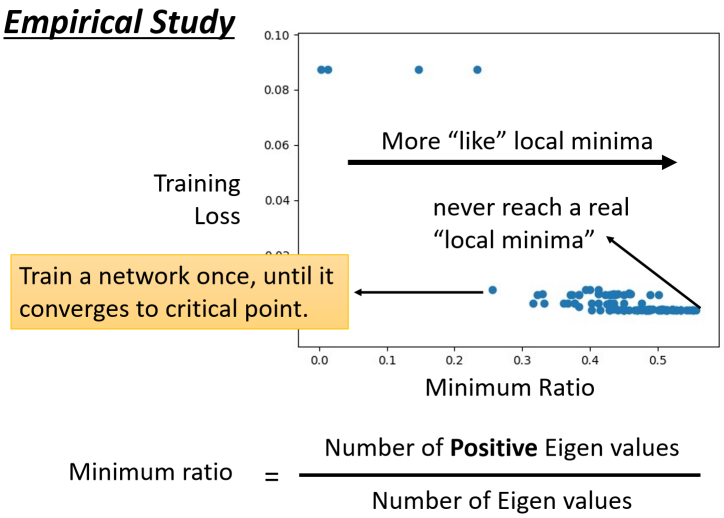
(this method is seldom used in practice)

现在有一个简单函数

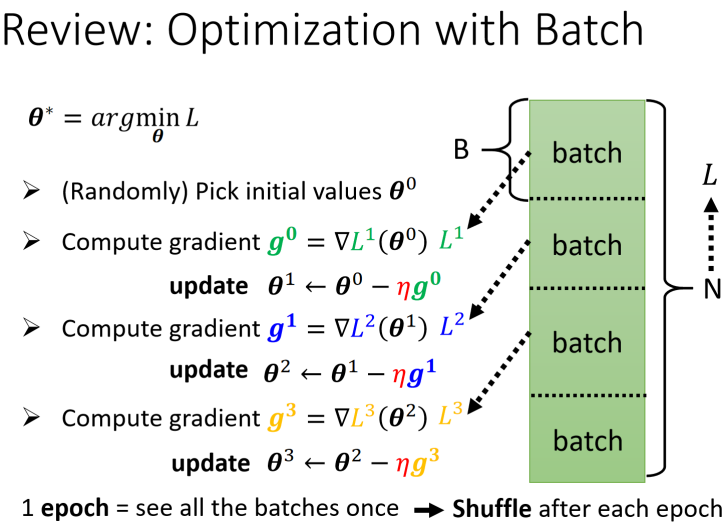
$$y = w_1 w_2 x$$

只有一个训练集（1,1），我们求出梯度为0的点（0,0），计算出黑塞矩阵的特征向量（1,1），正好是优化的方向。

在实际中，local minima通常很少，更多的是saddle point，原因是某个维度的local minima可以转化成更高的维度（增加参数）的saddle point。在下图的研究中，梯度趋于0时，minimum ratio（为1代表local minima）最高仅为0.5多一点。



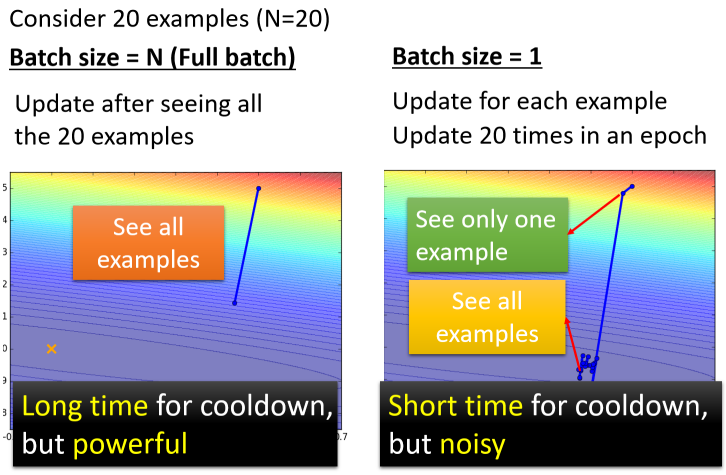
(3) batch



batch就是将训练集分为小块，每次计算每块的梯度来更新θ,在一次epoch之后，重新打乱训练集数据的顺序，然后再划分batch。

small batch vs large batch：

（1）

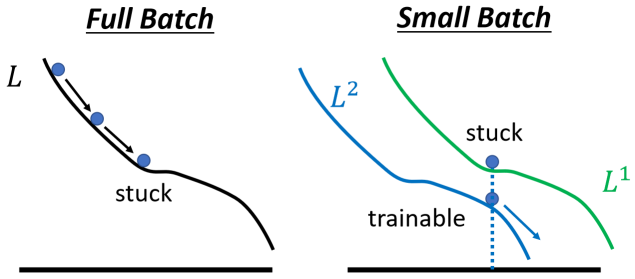


batchsize越大，一次update需要更长的时间，但是更新的走势趋于稳定；batchsize越小，一次update的时间短，更新有很多噪声。

（2）

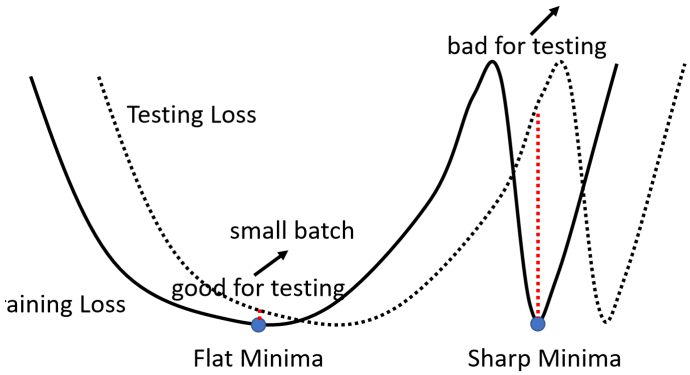
但是，batchsize越大，更新的时间真的越长吗？实际上，由于是在GPU上运行，多个任务可以同时进行，在batchsize不是太大（超过GPU处理单元几个数量级）的时候，batchsize越大，总体时间越短。

（3）那batchsize小的时候什么优点也没有吗？实际上，noisy的更新曲线更有利于训练集上loss的减小，原因是：



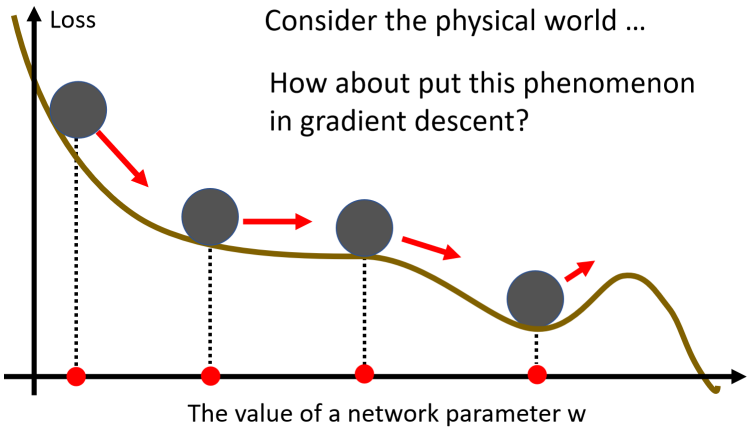
由上图，在只有一个batch的情况下，在梯度为0的点update就停止了。而在有多个small batch的情况下，如果每个batch的数据略有差异，那么update可能还可以继续进行。

（4）经过研究，small batch对测试集loss的减小也是有利的。



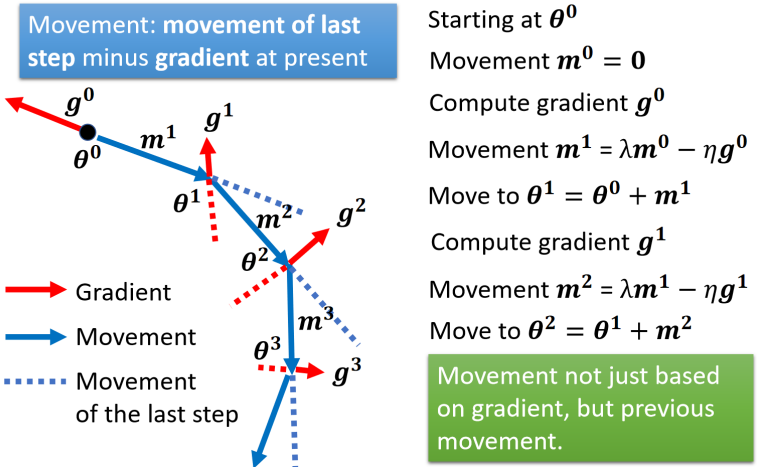
如上图，测试集和训练集可能存在一些误差，这时候flat minima相比于sharp minima更有利于测试集，因为sharp minima的两端太过陡峭，一点偏差就会使loss急剧增长。而small batch更趋向于找到flat minima。因此，small match对测试集更好。

(4) Momentum



考虑上图的场景，一个小球从高处落下，即使在local minima也没有停止运动，因为在local minima时小球仍有动能。类比小球，我们并不想让我们寻找loss最低点的路径被梯度限制死了，我们想让寻找loss的过程再朝运动方向延伸一点，这样没准能找到更好的minima。

因此，我们更新的过程由两部分来决定，第一部分是梯度的反方向，第二部分是前一步更新的方向。



由上图，θ前进的方向在g的反方向和m之间。