参考资料（从上至下，越上越好）

https://blog.csdn.net/bit452/category_10569531.html

https://blog.csdn.net/weixin_43847596/category_11440961.html

https://blog.csdn.net/m0_56676945/category_12272697.html

线性模型

visdom可视化工具包，可以看训练的图表

课程代码

```python
import numpy as np
import matplotlib.pyplot as plt
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]
def forward(x): return x * w
def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) * (y_pred - y)
w_list = []
mse_list = []
for w in np.arange(0.0, 4.1, 0.1):
    print('w=', w)
    l_sum = 0
    for x_val, y_val in zip(x_data, y_data):
        y_pred_val = forward(x_val)
        loss_val = loss(x_val, y_val)
        l_sum += loss_val
        print('\t', x_val, y_val, y_pred_val, loss_val)
    print('MSE=', l_sum / 3)
    w_list.append(w)
    mse_list.append(l_sum / 3)
plt.plot(w_list, mse_list)
plt.ylabel('Loss')
plt.xlabel('w')
plt.show()
```

作业

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]
def forward(x,b):
    return x * w+b
def loss(x, y,b):
    y_pred = forward(x,b)
    return (y_pred - y) * (y_pred - y)
w_list = []
b_list=[]
mse_list = []
for w in np.arange(0.0, 4.1, 0.1):
    for b in np.arange(0.0,2.0,0.5):
        print('w=', w)
        print('b=',b)
        l_sum = 0
        for x_val, y_val in zip(x_data, y_data):
            y_pred_val = forward(x_val,b)
            loss_val = loss(x_val, y_val,b)
            l_sum += loss_val
            print('\t', x_val, y_val, y_pred_val, loss_val)
        print('MSE=', l_sum / 3)
        w_list.append(w)
        b_list.append(b)
        mse_list.append(l_sum / 3)


fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# 传入w_list、b_list和mse_list作为x、y、z坐标数据来绘制三维表面
ax.plot_trisurf(w_list, b_list, mse_list, cmap='viridis')

# 设置坐标轴标签
ax.set_xlabel('W')
ax.set_ylabel('B')
ax.set_zlabel('MSE')

# 显示图形
plt.show()
```

- `plot_surface`: `plot_surface` 函数期望输入的数据是均匀网格的形式，通常需要提供一个二维的 `X` 和 `Y` 网格以及对应的高度数据 `Z`。这样的输入适合表示规则的网格数据。
- `plot_trisurf`: `plot_trisurf` 函数更适合不规则的数据，它使用三角形来组合数据点，因此可以接受不规则的 `X`、`Y` 和 `Z` 数据。

梯度下降算法

代码实现

```
import matplotlib.pyplot as plt

# prepare the training set
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

# initial guess of weight
w = 1.0

# define the model linear model y = w*x
def forward(x):
    return x*w

# define the cost function MSE
```

```
def cost(xs, ys):
    cost = 0
    for x, y in zip(xs, ys):
        y_pred = forward(x)
        cost += (y_pred - y) ** 2
    return cost / len(xs)

# define the gradient function gd
'''
如何找到梯度下降的方向：用目标函数对权重w求导数可找到梯度的变化方向
'''
def gradient(xs, ys):
    grad = 0
    for x, y in zip(xs, ys):
        grad += 2*x*(x*w - y)
    return grad / len(xs)
epoch_list = []
cost_list = []

print("Predict (before training)", 4, forward(4))
for epoch in range(100):
    # 计算损失值
    cost_val = cost(x_data, y_data)
    # 计算梯度变化值
    grad_val = gradient(x_data, y_data)
    w -= 0.01 * grad_val
    print('Epoch:', epoch, "w=", "%.2f" %w, "loss=", "%.2f" %cost_val)
    epoch_list.append(epoch)
    cost_list.append(cost_val)


print("Predict (after training)", 4, forward(4))
plt.plot(epoch_list, cost_list)
plt.ylabel('cost')
plt.xlabel('epoch')
plt.show()
```

SGD随机梯度下降代码实现

```
import matplotlib.pyplot as plt
# prepare the training set
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

# initial guess of weight
w = 1.0
```

```python
# define the model linear model y = w*x
def forward(x):
    return x * w

# define the cost function MSE
def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) ** 2

# define the gradient function gd
'''
如何找到梯度下降的方向：用目标函数对权重w求导数可找到梯度的变化方向
'''
def gradient(x, y):
    return 2 * x * (x * w - y)

epoch_list = []
loss_list = []

print("Predict (before training)", 4, forward(4))

# 对每一个样本的梯度进行更新
for epoch in range(100):
    for x, y in zip(x_data, y_data):
        loss_val = loss(x, y)
        grad_val = gradient(x, y)
        w = w - 0.01 * grad_val
        print("Epoch: ", epoch, "w: ", "%.2lf"%w, "loss: ", "%.2lf"%loss_val)
    epoch_list.append(epoch)
    loss_list.append(loss_val)

print("Predict (after training)", 4, forward(4))
plt.plot(epoch_list, loss_list)
plt.ylabel('loss')
plt.xlabel('epoch')
plt.show()
```

拿一个进行更新

找到一个更优的点

但时间很长

所以要用batch(取一部分)

反向传播Back Propagation

定义函数其实就是在画图

```
import matplotlib.pyplot as plt
import torch

# PyTorch实现反向传播Backward
# 1.计算损失 2.Backward 3.梯度下降继续更新
x_data = [1.0,2.0,3.0]
y_data = [2.0,4.0,6.0]

w = torch.Tensor([1.0])      # 初始化权重
w.requires_grad = True       # 表明w需要计算梯度

# define the linear model
def forward(x):
    return x * w             # 这里会构建计算图 将x转化成Tensor

# 损失函数的求解，构建计算图，并不是乘法或者乘方运算
def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) ** 2

# 打印学习之前的值，.item()表示输出张量的值
print("Predict (before training)", 4, forward(4).item())

learning_rate = 0.01
epoch_list = []
loss_list = []

# 训练过程
for epoch in range(100):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)       # forward前馈
        l.backward()         # 成员函数backward()向后传播 自动求出所有需要的梯度
        print('\tgrad:', x, y, w.grad.item())    # 将梯度存到w之中,随后释放计算图 item()可以直
接将梯度变成标量
        w.data = w.data - learning_rate * w.grad.data    # w的grad也是张量，计算应该取data
不去建立计算图
        w.grad.data.zero_()                      # 释放data
        epoch_list.append(epoch)
        loss_list.append(l.item())

    print("progress:", epoch, l.item())

print("predict (after training)", 4, forward(4).item())

# 绘制可视化
plt.plot(epoch_list, loss_list)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

实现线性回归Linear Regression with PyTorch

重点不在求导数，重点在构图

```
import torch

# mini-batch需要的数据是Tensor
x_data = torch.Tensor([[1.0], [2.0], [3.0]])
y_data = torch.Tensor([[2.0], [4.0], [6.0]])

# Design Model  重点目标在于构造计算图
"""
所有模型都要继承自Model
最少实现两个成员方法
    构造函数 初始化: __init__()
    前馈: forward()
Model自动实现backward
可以在Functions中构建自己的计算块
"""
class LinearModel(torch.nn.Module):
    def __init__(self):
        super(LinearModel, self).__init__()
        self.linear = torch.nn.Linear(1, 1)

    def forward(self, x):
        y_pred = self.linear(x)                 # linear成为了可调用的对象 直接计算forward
        return y_pred

model = LinearModel()               # 创建类的实例

# 3.Construct Loss(MSE (y_pred - y)**2 ) and Optimizer
# 构造计算图就需要集成Model模块
criterion = torch.nn.MSELoss(size_average=False)    #需要的参数是y_pred和y
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)

# 4.Training Cycle
for epoch in range(100):
    y_pred = model(x_data)          # Forward
    loss = criterion(y_pred, y_data)
    print(epoch, loss.item())

    optimizer.zero_grad()           # 梯度清零
    loss.backward()                 # 反馈
    optimizer.step()                # 更新

#Output weight and bias
print('w = ', model.linear.weight.item())
print('b = ', model.linear.bias.item())

#Test Model
x_test = torch.Tensor([[4.0]])
y_test = model(x_test)
print('y_pred = ', y_test.data)
```

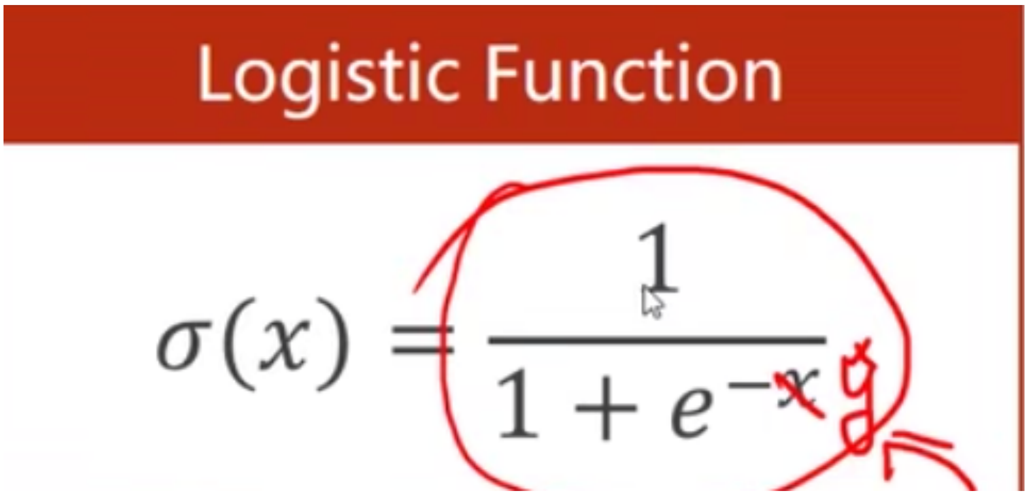
loss.backward()计算出梯度,

optimizer.step(),利用计算出的梯度进行更新

optimizer.zero_grad()把上面计算的清除,他放前面相当于每次计算回归前,都把前面计算过的清除

Logistic Regression

对于概率问题,映射到[0,1]



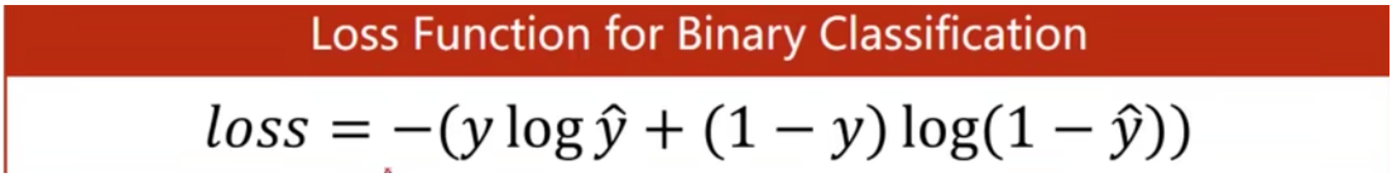
Logistic Function

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

损失函数

要计算分布之间的差异

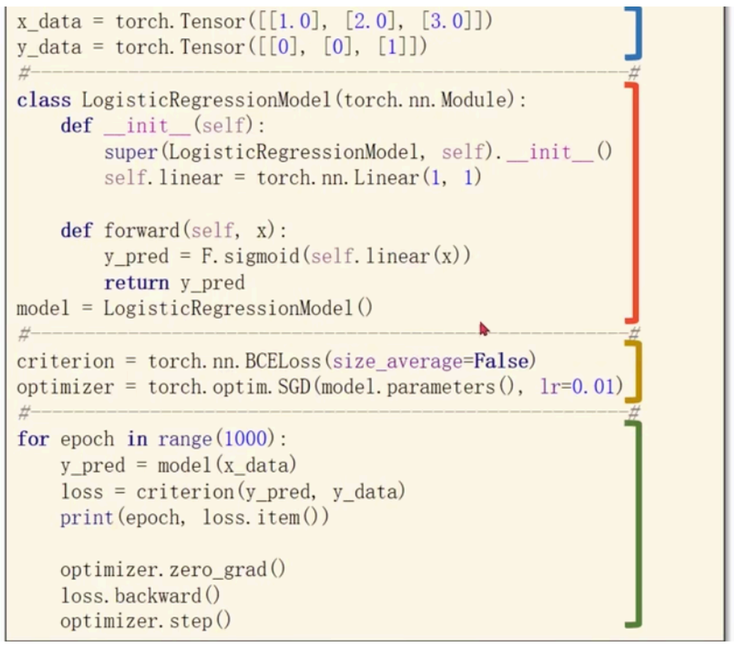
使用的是二分类交叉熵



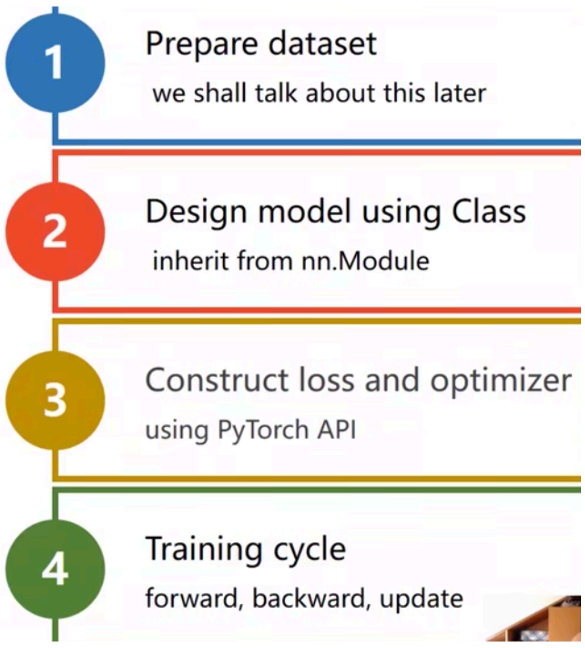
Loss Function for Binary Classification

$$loss = -(y\log\hat{y} + (1-y)\log(1-\hat{y}))$$

代码书写四步骤



```
x_data = torch.Tensor([[1.0], [2.0], [3.0]])
y_data = torch.Tensor([[0], [0], [1]])
#------------------------------------------------------------#
class LogisticRegressionModel(torch.nn.Module):
    def __init__(self):
        super(LogisticRegressionModel, self).__init__()
        self.linear = torch.nn.Linear(1, 1)

    def forward(self, x):
        y_pred = F.sigmoid(self.linear(x))
        return y_pred
model = LogisticRegressionModel()
#------------------------------------------------------------#
criterion = torch.nn.BCELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
#------------------------------------------------------------#
for epoch in range(1000):
    y_pred = model(x_data)
    loss = criterion(y_pred, y_data)
    print(epoch, loss.item())

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



```python
import torch

# 准备数据集
x_data = torch.Tensor([[1.0], [2.0], [3.0]])
y_data = torch.Tensor([[0], [0], [1]])

# 设计网络模型
class LogisticRegressionModel(torch.nn.Module):
    def __init__(self):
        super(LogisticRegressionModel, self).__init__()
        self.linear = torch.nn.Linear(1, 1)

    def forward(self, x):
        y_pred = torch.sigmoid(self.linear(x))
        return y_pred
model = LogisticRegressionModel()

# Construct Loss and optimizer
criterion = torch.nn.BCELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)

# Training cycle
for epoch in range(2000):
    y_pred = model(x_data)
    loss = criterion(y_pred, y_data)
    # print(epoch, loss.item())

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

```
x_test = torch.Tensor([1.0])
y_test = model(x_test)
print("y_pred = ", y_test.data)
import numpy as np
import matplotlib.pyplot as plt

'''
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
在指定的间隔内返回均匀间隔的数字。
返回num均匀分布的样本，在[start, stop]。
这个区间的端点可以任意的被排除在外。
'''
x = np.linspace(0,10,200)   #在0~10中，均匀取出200个点
# print(x)
x_t = torch.Tensor(x).view(200,1)  #将200个点变成（200,1）的张量
# print(x_t)
y_t = model(x_t)  #得到y_pred_t是个张量
# print(y_t)
y = y_t.data.numpy() #将y_pred_t张量转化为矩阵形式
# print(y)
plt.plot(x,y)
plt.plot([0,10],[0.5,0.5],c='r')
plt.xlabel('Hours')
plt.ylabel('Probability of Pass')
plt.grid()
plt.show()
```

Multiple Dimension Input

合并运算

$$\begin{bmatrix} \hat{y}^{(1)} \\ \vdots \\ \hat{y}^{(N)} \end{bmatrix} = \begin{bmatrix} \sigma(z^{(1)}) \\ \vdots \\ \sigma(z^{(N)}) \end{bmatrix} = \sigma\left(\begin{bmatrix} z^{(1)} \\ \vdots \\ z^{(N)} \end{bmatrix}\right)$$

Sigmoid function is in an element-wise fashion.

$$z^{(1)} = \begin{bmatrix} x_1^{(1)} & \cdots & x_8^{(1)} \end{bmatrix} \begin{bmatrix} \omega_1 \\ \vdots \\ \omega_8 \end{bmatrix} + b$$

$$\vdots$$

$$z^{(N)} = \begin{bmatrix} x_1^{(N)} & \cdots & x_8^{(N)} \end{bmatrix} \begin{bmatrix} \omega_1 \\ \vdots \\ \omega_8 \end{bmatrix} + b$$

$$\Rightarrow \begin{bmatrix} z^{(1)} \\ \vdots \\ z^{(N)} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & \cdots & x_8^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \cdots & x_8^{(N)} \end{bmatrix} \begin{bmatrix} \omega_1 \\ \vdots \\ \omega_8 \end{bmatrix} + \begin{bmatrix} b \\ \vdots \\ b \end{bmatrix}$$

转换成向量化计算可以利用设备的并行计算能力

```
self.linear = torch.nn.Linear(1, 1)
```

这行代码的意思输入一维，输出也是一维

```
import torch
import numpy as np
import matplotlib.pyplot as plt

xy = np.loadtxt('diabetes.csv.gz', delimiter=',', dtype=np.float32)
x_data = torch.from_numpy(xy[ : ,  :-1])    # 第一个:的前后参数缺省，表示读取所有行 第二个:指从开始列到倒数第二列
y_data = torch.from_numpy(xy[ : , [-1]])    # [-1]使向量转换为矩阵  from_numpy可以生成tensor
# test = np.loadtxt('test.csv.gz', delimiter='', dtype=np.float32)


# print(y_data)

# Define Model
class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear1 = torch.nn.Linear(8, 6)    # 表示8维->6维 空间变换
        self.linear2 = torch.nn.Linear(6, 4)
        self.linear3 = torch.nn.Linear(4, 1)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.linear1(x))
        x = self.sigmoid(self.linear2(x))
        x = self.sigmoid(self.linear3(x))
        return x

model = Model()

# Construct Loss and Optimizer
criterion = torch.nn.BCELoss(reduction = 'mean')
optimizer = torch.optim.SGD(model.parameters(), lr = 0.1)

epoch_list = []
loss_list = []
# Training Cycle
for epoch in range(100):
    # Forward
    y_pred = model(x_data)
    loss = criterion(y_pred, y_data)
```

```
        epoch_list.append(epoch)
        loss_list.append(loss.item())
        print(epoch, loss.item())
        # Backward
        optimizer.zero_grad()
        loss.backward()
        # Update
        optimizer.step()

plt.plot(epoch_list, loss_list)
plt.ylabel('loss_list')
plt.xlabel('epoch_list')
plt.show()
```

在深度学习中，使用多个神经网络层而不是一个单一的线性层（如`torch.nn.Linear(8, 1)`）通常有几个原因：

1. 表示能力：深度神经网络由多个层组成，每个层都有一组权重和偏差参数，这使得网络能够学习复杂的非线性函数。通过增加层数，网络可以更好地适应数据，捕获数据中的不同抽象层次的特征。
2. 特征提取：每个隐藏层可以被看作是数据的特征提取器。通过将数据传递给多个隐藏层，网络可以逐渐构建更高级别的特征表示，使得网络更容易从输入数据中提取有用的信息。
3. 非线性建模：深度神经网络中的每一层通常包括非线性激活函数（如ReLU、Sigmoid等），这允许网络学习非线性关系。如果您只使用一个线性层，网络将仅能够学习线性关系，无法捕获数据中的更复杂模式。
4. 减少参数：通过使用多个较小的线性层，您可以减少网络的参数量。这有助于降低过拟合的风险，使网络更容易训练，特别是当数据集相对较小时。

综上所述，多层神经网络的使用有助于提高网络的性能、捕获更复杂的模式、减少参数量以及实现非线性建模，这些都是在深度学习任务中非常重要的因素。因此，通常情况下，深度学习模型使用多个层而不是单一的线性层。

Dataset and DataLoader

代码总结

使用mini-batch，得用两层循环

该代码手动实现了继承Dataset,并重写了以下两个方法

```
1__len__ 2__getitem__
```

解决报错_pickle.PicklingError

```
_pickle.PicklingError: Can't pickle <class '__main__.DiabetesDataset'>: attribute
lookup DiabetesDataset on __main__ failed
```

这个错误是由于在使用多进程数据加载器（`DataLoader`）时出现的问题，具体原因是无法序列化（pickle）`DiabetesDataset` 类，因为它没有被放在独立的模块中。在多进程环境中，需要将数据加载器的数据和函数传递给不同的进程，但无法将定义在交互式环境中的类传递给子进程。

为了解决这个问题，你可以将 `DiabetesDataset` 类移到单独的Python文件（例如，一个名为`dataset.py`的文件）中，然后在你的主代码中导入它。这样，数据加载器将能够正确序列化`DiabetesDataset`类。

首先，将`DiabetesDataset`类移至一个单独的Python文件（比如`dataset.py`）：

DiabetesDataset.py

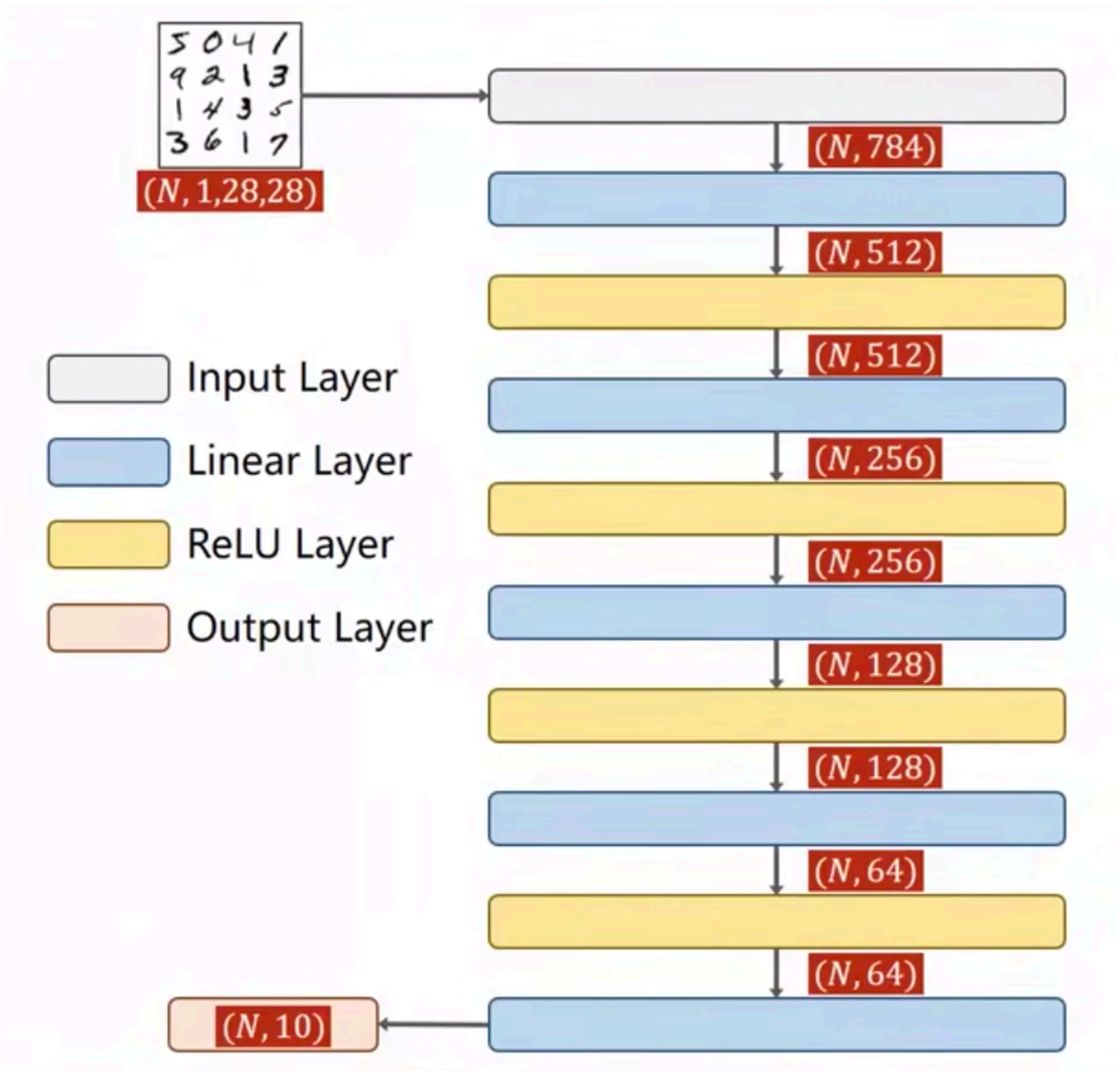
```python
import torch
import numpy as np
from torch.utils.data import Dataset
class DiabetesDataset(Dataset):
    def __init__(self, filepath):
        xy = np.loadtxt(filepath, delimiter=',', dtype=np.float32)
        self.len = xy.shape[0]              # 取出有多少行
        self.x_data = torch.from_numpy(xy[:, :-1])
        self.y_data = torch.from_numpy(xy[:, [-1]])

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

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

dataloader.py

```python
import torch
import numpy as np
from torch.utils.data import Dataset, DataLoader
from DiabetesDataset import DiabetesDataset



dataset = DiabetesDataset('diabetes.csv.gz')
train_loader = DataLoader(dataset=dataset, batch_size=32, shuffle=True, num_workers=2)

class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear1 = torch.nn.Linear(8, 6)
        self.linear2 = torch.nn.Linear(6, 4)
        self.linear3 = torch.nn.Linear(4, 1)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.linear1(x))
        x = self.sigmoid(self.linear2(x))
        x = self.sigmoid(self.linear3(x))
        return x

model = Model()

criterion = torch.nn.BCELoss(reduction='mean')
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

if __name__ == '__main__':
    for epoch in range(100):
        for i, data in enumerate(train_loader, 0):
            inputs, labels = data
            y_pred = model(inputs)
            loss = criterion(y_pred, labels)
            print(epoch, i, loss.item())
            optimizer.zero_grad()
            loss.backward()

            optimizer.step()
```

Softmax Classifier

解决多分类问题



转换是为了运算更高效

```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,),
(0.3081,))]) # 归一化,均值和方差
```

```
transforms.ToTensor()
```

上面一行是为了转换成带通道的为了运算更高效

```
transforms.Normalize((0.1307,), (0.3081,))
```

上面一行是为了归一化，这两个数据不用管，老师给的

为什么使用torch.nn.CrossEntropyLoss() 最后一次不进行非线性变换

在深度学习中，使用`torch.nn.CrossEntropyLoss()`通常用于多类别分类问题，它的设计已经包含了Softmax非线性变换。这意味着最后一层不需要显式的非线性激活函数，因为CrossEntropyLoss已经处理了这一点。

```python
import torch
from torchvision import transforms
from torchvision import datasets
from torch.utils.data import DataLoader
import torch.nn.functional as F
import torch.optim as optim

# prepare dataset

batch_size = 64
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), 
(0.3081,))])  # 归一化,均值和方差

train_dataset = datasets.MNIST(root='../dataset/mnist/', train=True, download=True, 
transform=transform)
train_loader = DataLoader(train_dataset, shuffle=True, batch_size=batch_size)
test_dataset = datasets.MNIST(root='../dataset/mnist/', train=False, download=True, 
transform=transform)
test_loader = DataLoader(test_dataset, shuffle=False, batch_size=batch_size)


# design model using class


class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.l1 = torch.nn.Linear(784, 512)
        self.l2 = torch.nn.Linear(512, 256)
        self.l3 = torch.nn.Linear(256, 128)
        self.l4 = torch.nn.Linear(128, 64)
        self.l5 = torch.nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(-1, 784)  # -1其实就是自动获取mini_batch
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = F.relu(self.l3(x))
        x = F.relu(self.l4(x))
        return self.l5(x)  # 最后一层不做激活，不进行非线性变换
```

```python
model = Net()

# construct loss and optimizer
criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)


# training cycle forward, backward, update


def train(epoch):
    running_loss = 0.0
    for batch_idx, data in enumerate(train_loader, 0):
        # 获得一个批次的数据和标签
        inputs, target = data
        optimizer.zero_grad()
        # 获得模型预测结果(64, 10)
        outputs = model(inputs)
        # 交叉熵代价函数outputs(64,10),target（64）
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if batch_idx % 300 == 299:
            print('[%d, %5d] loss: %.3f' % (epoch + 1, batch_idx + 1, running_loss / 
300))
            running_loss = 0.0


def test():
    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            outputs = model(images)
            _, predicted = torch.max(outputs.data, dim=1)  # dim = 1 列是第0个维度，行是第
1个维度
            total += labels.size(0)
            correct += (predicted == labels).sum().item()  # 张量之间的比较运算
    print('accuracy on test set: %d %% ' % (100 * correct / total))


if __name__ == '__main__':
    for epoch in range(10):
        train(epoch)
        test()
```

Basic CNN

知识点

通过卷积层，目的是保留空间结构

 下采样目的减少数据量

 特征提取器

全连接网络 进行分类

Q&A

卷积神经网络中，构建模型要用到ReLU层，ReLU层的作用是什么

ReLU（Rectified Linear Unit）是一种常用的激活函数，它在卷积神经网络（CNN）中被广泛使用。ReLU的作用是引入非线性性，以帮助模型更好地捕捉和表示复杂的数据模式和特征。以下是ReLU层的主要作用：

1. 非线性激活：ReLU是一个非线性函数，它在输入大于零时返回输入值本身，而在输入小于或等于零时返回零。这个非线性性使神经网络能够学习复杂的非线性关系，从而更好地拟合和表示数据中的模式。
2. 梯度传播：相对于一些其他激活函数，如Sigmoid或Tanh，ReLU的梯度计算更加简单，有助于避免梯度消失问题。这使得训练更深的神经网络更为容易，因为梯度可以更好地传播回网络的较早层。
3. 稀疏激活：当输入小于零时，ReLU的输出为零，这意味着神经元对于特定输入不会激活。这种稀疏激活的特性可以使网络变得更加稀疏，从而减少计算和存储的需求，有助于训练更高效的模型。
4. 加速收敛：由于ReLU的非线性特性和梯度传播的改善，它通常能够加速神经网络的收敛，即模型可以更快地学习和拟合数据。

总之，ReLU层的作用是引入非线性性，帮助神经网络学习非线性关系，加速收敛，改善梯度传播，以及在稀疏性方面提供一些优势。这些特性使ReLU成为卷积神经网络中常见的激活函数。

代码

```
import torch
from torchvision import transforms
from torchvision import datasets
from torch.utils.data import DataLoader
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt

# prepare dataset

batch_size = 64
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,),
(0.3081,))])

train_dataset = datasets.MNIST(root='../dataset/mnist/', train=True, download=True,
transform=transform)
train_loader = DataLoader(train_dataset, shuffle=True, batch_size=batch_size)
```

```python
test_dataset = datasets.MNIST(root='../dataset/mnist/', train=False, download=True,
transform=transform)
test_loader = DataLoader(test_dataset, shuffle=False, batch_size=batch_size)

# design model using class


class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = torch.nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = torch.nn.Conv2d(10, 20, kernel_size=5)
        self.pooling = torch.nn.MaxPool2d(2)
        self.fc = torch.nn.Linear(320, 10)


    def forward(self, x):
        # flatten data from (n,1,28,28) to (n, 784)

        batch_size = x.size(0)
        x = F.relu(self.pooling(self.conv1(x)))
        x = F.relu(self.pooling(self.conv2(x)))
        x = x.view(batch_size, -1) # -1 此处自动算出的是320
        # print("x.shape",x.shape)
        x = self.fc(x)

        return x


model = Net()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# construct loss and optimizer
criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

# training cycle forward, backward, update


def train(epoch):
    running_loss = 0.0
    for batch_idx, data in enumerate(train_loader, 0):
        inputs, target = data
        inputs, target = inputs.to(device), target.to(device)
        optimizer.zero_grad()

        outputs = model(inputs)
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if batch_idx % 300 == 299:
            print('[%d, %5d] loss: %.3f' % (epoch+1, batch_idx+1, running_loss/300))
            running_loss = 0.0


def test():
    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, dim=1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print('accuracy on test set: %d %% ' % (100*correct/total))
    return correct/total


if __name__ == '__main__':
    epoch_list = []
    acc_list = []

    for epoch in range(10):
        train(epoch)
        acc = test()
        epoch_list.append(epoch)
        acc_list.append(acc)

    plt.plot(epoch_list,acc_list)
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.show()
```

Advanced CNN

知识点

利用1*1的卷积能简化计算

很多层

但是很多层容易出现梯度消失问题

解决梯度消失

跳连接解决这个问题

建议

1.花书

2.阅读pytorch文档

3.复现代码（读代码）

代码实现

```python
import torch
import torch.nn as nn
from torchvision import transforms
from torchvision import datasets
from torch.utils.data import DataLoader
import torch.nn.functional as F
import torch.optim as optim

# prepare dataset

batch_size = 64
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])  # 归一化,均值和方差

train_dataset = datasets.MNIST(root='../dataset/mnist/', train=True, download=True, transform=transform)
train_loader = DataLoader(train_dataset, shuffle=True, batch_size=batch_size)
test_dataset = datasets.MNIST(root='../dataset/mnist/', train=False, download=True, transform=transform)
test_loader = DataLoader(test_dataset, shuffle=False, batch_size=batch_size)


# design model using class
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.channels = channels
        self.conv1 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)
    def forward(self, x):
        y = F.relu(self.conv1(x))
        y = self.conv2(y)
        return F.relu(x + y)


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=5)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5)  # 88 = 24x3 + 16

        self.rblock1 = ResidualBlock(16)
        self.rblock2 = ResidualBlock(32)

        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(512, 10)  # 暂时不知道1408咋能自动出来的

    def forward(self, x):
        in_size = x.size(0)

        x = self.mp(F.relu(self.conv1(x)))
        x = self.rblock1(x)
        x = self.mp(F.relu(self.conv2(x)))
        x = self.rblock2(x)

        x = x.view(in_size, -1)
        x = self.fc(x)
        return x


model = Net()

# construct loss and optimizer
criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)


# training cycle forward, backward, update


def train(epoch):
    running_loss = 0.0
    for batch_idx, data in enumerate(train_loader, 0):
        inputs, target = data
        optimizer.zero_grad()

        outputs = model(inputs)
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if batch_idx % 300 == 299:
            print('[%d, %5d] loss: %.3f' % (epoch + 1, batch_idx + 1, running_loss /
300))
            running_loss = 0.0


def test():
    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            outputs = model(images)
            _, predicted = torch.max(outputs.data, dim=1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print('accuracy on test set: %d %% ' % (100 * correct / total))


if __name__ == '__main__':
    for epoch in range(10):
        train(epoch)
        test()
```

basic RNN

代码疑问

```
 def forward(self, input):
        hidden = torch.zeros(self.num_layers, self.batch_size, self.hidden_size)
        out, _ = self.rnn(input, hidden)  # out: tensor of shape (seq_len, batch,
hidden_size)
        return out.view(-1, self.hidden_size)  # 将输出的三维张量转换为二维张量,
(seqLen×batchSize,hiddenSize)
```

在深度学习中，将输出转换为二维张量的目的通常是为了将其传递给后续层或损失函数，以便与问题的要求相匹配。在这里，将输出从三维张量 `(seq_len, batch_size, hidden_size)` 转换为二维张量 `(seq_len * batch_size, hidden_size)` 有以下原因：

1. 适配损失函数：在训练深度学习模型时，通常需要计算损失函数，损失函数通常期望输入是二维的。例如，在文本分类任务中，通常使用交叉熵损失函数，它的输入是模型的预测输出和实际标签，而这些都需要具有相同的二维形状。将输出从三维转换为二维是为了与损失函数的输入形状相匹配。
2. 批次处理：深度学习模型通常使用批次（batch）进行训练，而不是一个样本一个样本地进行。将输出转换为二维张量可以使模型同时处理整个批次的数据，从而提高训练效率。
3. 数据整体性：某些任务需要将序列数据整体视为一个连续序列，而不考虑批次中的单个序列。将输出转换为二维张量使得可以在整个序列上执行操作，而不会受到批次的限制。

在你的代码中，`out.view(-1, self.hidden_size)` 的目的是将 RNN 模型的输出从 `(seq_len, batch_size, hidden_size)` 转换为 `(seq_len * batch_size, hidden_size)`，以便在后续步骤中更方便地计算损失、进行反向传播和进行梯度更新。这种形状变换是深度学习模型训练中的常见操作，以满足任务的要求。

advanced RNN