Project1 说明文档

姓名: 林奕铖

学号: 20302010008

Part1. 反向传播算法

代码基本结构

文件目录结构

```
│
# 网络结构
├─ data_manager.py # 训练与测试数据管理
├─ network.py # 主要网络结构
# 回归任务
├─ regression.py # 回归任务(包括数据生成、训练、测试、拟合可视化等)
├─ regression # 存储回归任务的数据及模型
│  ├─ data10000.npy
│  └─ my_sin_model.pkl
# 分类任务
├─ classification.py # 分类任务(包括数据预处理、训练、测试等)
├─ classification # 存储分类任务的数据及模型
│  ├─ data_bmp
│  ├─ dataX.npy
│  ├─ dataY.npy
│  └─ my_model.pkl
└─
```

代码基本结构

- **`network.py`**

  定义网络基本结构

  - **函数定义**

    为了实现灵活调整激活函数、损失函数的目的，在文件开头定义了一组函数及其导函数。以激活函数为例，其参数都是一个 `numpy` 矩阵，返回的也是一个 `numpy` 矩阵。

    这样做的好处在于，在定义网络结构的激活函数时，只需传入Callable对象即可。因此可以方便地更换激活函数、或在不同层使用不同的激活函数。

    ```python
    def ReLU(X):
        return np.maximum(0, X)

    def ReLU_derived(X):
        return np.where(X <= 0, 0, 1)

    def identity(X):
        return X

    def identity_derived(X):
        return np.ones(X.shape)
    ```

  - **线性层定义**

    线性层类用于进行线性变换和激活函数变换。该类的实例化对象可以作为神经网络的一层，用于构建一个完整的神经网络。

    该类包含以下方法：

    1. `__init__`方法：初始化线性层的参数，包括输入大小、输出大小、激活函数、激活函数的导数等。同时，随机初始化该层的权重矩阵`W`和偏置向量`b`。

2. `forward`方法：前向传播过程中，将输入数据进行线性变换和激活函数变换，得到该层的输出，并返回该输出。
3. `test_forward`方法：用于测试前向传播过程中的线性变换和激活函数变换，不会记录该层的输入、输出等状态。
4. `backward`方法：反向传播过程中，根据损失函数对该层输出的导数，计算该层权重矩阵和偏置向量的梯度，并返回该层输入的导数，以便进行下一层的反向传播。
5. `update`方法：根据梯度下降的优化算法，更新该层的权重矩阵和偏置向量。

▼ **神经网络定义**

神经网络类作为一个完整的神经网络，用于进行训练和预测。它由多个线性层类组合而成。

该类包含以下方法：

1. `__init__`方法：初始化神经网络的参数，包括每层的神经元数量、激活函数和激活函数的导数。在该方法中，通过调用`LinearLayer`类的构造函数，创建了多个线性层，并将它们组合在一起，构成了一个完整的神经网络。
2. `forward`方法：前向传播过程中，将输入数据依次传递给每一层，并将每一层的输出作为下一层的输入，最终得到神经网络的输出，并返回该输出。
3. `backward`方法：反向传播过程中，根据损失函数对神经网络输出的导数，依次计算每一层的输入的导数，并将其传递给上一层，以便进行下一层的反向传播。
4. `update`方法：根据梯度下降的优化算法，更新神经网络的参数，包括每一层的权重矩阵和偏置向量。
5. `test_forward`方法：用于测试神经网络的前向传播过程，只记录输出但不改变神经网络本身的参数。

▼ **`regression.py`**

回归任务的训练与测试

▼ **训练函数**

1. 创建神经网络：通过调用`Network`类的构造函数，创建一个多层神经网络，并指定每层的神经元数量、激活函数和激活函数的导数。
2. 进行训练：通过循环迭代训练数据集，依次计算每个样本的输出，并根据损失函数的导数计算每一层的梯度，并更新神经网络的参数。
3. 保存模型：将训练好的神经网络模型保存到文件中，以便后续的预测使用。

```
# create the network
my_network = Network(
  neurons_per_layer=neurons,
  activation_functions=activation_functions,
  activation_derived_functions=activation_derived_functions,
)

# training
for epoch in range(epochs):
  X_train, Y_train = data_manager.train_data()
  for i in range(data_manager.train_size):
      x = X_train[:, i:i+1]
      y = Y_train[:, i:i+1]
      y_pred = my_network.forward(x)

      dLoss_dy = loss_function_derived(y, y_pred)
      my_network.backward(dLoss_dy)
      my_network.update(learning_rate)

      if i == 0:
          print(f"epoch {epoch}, loss {loss_function(y, y_pred)}")

# save the model
with open("regression/my_sin_model.pkl", "wb") as f:
  pickle.dump(my_network, f)
```

▼ **测试函数**

1. 加载模型：通过调用Python标准库`pickle`的`load`函数，从文件中加载训练好的神经网络模型。

2. 进行测试：通过调用神经网络对象的`test_forward`方法，对测试数据集进行预测，并计算预测结果与真实标签之间的均方误差 (MSE) 和平均误差。
3. 绘制图像：将测试数据集的真实标签和预测结果绘制在同一张图上，以便进行可视化比较。

```
# load the model
my_network = None
with open("regression/my_sin_model.pkl", "rb") as f:
    my_network = pickle.load(f)

# testing
X_test, Y_test = data_manager.test_data()
Y_pred = my_network.test_forward(X_test)
print(f"MSE = {MSE(Y_test, Y_pred)}")
print(f"Avg error = {np.average(np.abs(Y_test - Y_pred))}")

# plotting
plt.scatter(X_test, Y_test, color='blue', s=5, label='actual')
plt.scatter(X_test, Y_pred, color='red', s=5, label='predicted')
plt.legend()
plt.show()
```

▼ **classification.py**

分类任务的训练与测试

▼ **训练函数**

训练函数与回归任务的训练函数一致。

▼ **测试函数**

1. 加载模型：通过调用Python标准库`pickle`的`load`函数，从文件中加载训练好的神经网络模型。
2. 进行测试：通过调用神经网络对象的`test_forward`方法，对测试数据集进行预测，并计算预测结果与真实标签之间的准确率。

```
# load the model
my_network = None
with open("classification/my_model.pkl", "rb") as f:
  my_network = pickle.load(f)

# testing
X_test, Y_true = data_manager.test_data()
Y_pred = my_network.test_forward(X_test)
true = np.argmax(Y_true, axis=0)
pred = np.argmax(Y_pred, axis=0)
correct = np.sum(true == pred)
total = len(pred)
print(f"correct: {correct}, total: {total}, accuracy: {correct / total * 100: .2f}%")
```

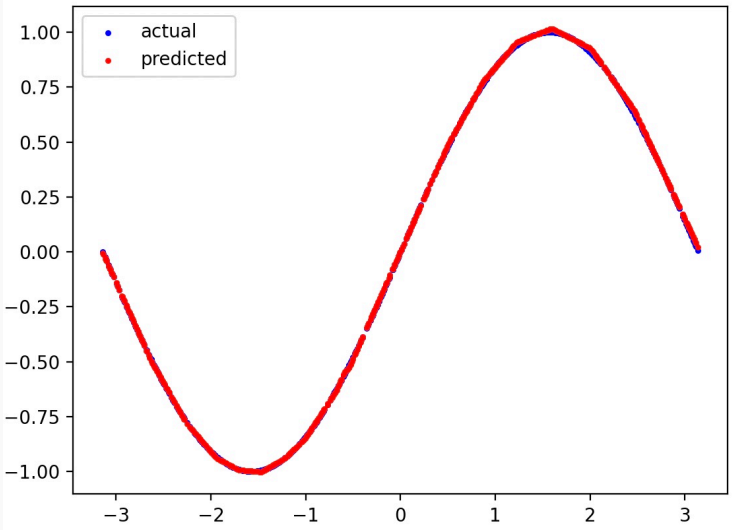
实验结果与参数比较

实验结果

回归任务

MSE = **0.0283**

平均误差为 **0.00370**



分类任务

correct: **711**, total: **744**, accuracy: **95.56%**

不同网络结构、网络参数的实验比较

对于 **回归任务**

我采用了仅一层隐藏层(50个神经元)的网络结构，激活函数为 `ReLU` 函数，学习率为 `0.005`， `epochs` 为 `1000`。

该网络已经能够较好地解决拟合问题。

对于 **分类任务**

```
learning_rate = 0.05
epochs = 5000
neurons = [784, 256, 64, 12]
activation_functions = [None, ReLU, ReLU, softmax]
activation_derived_functions = [None, ReLU_derived, ReLU_derived, identity_derived]
loss_function = cross_entropy
loss_function_derived = cross_entropy_and_softmax_derived
```

对于28*28的输入图片，展开成一维向量后长度为784。为了确保充分学习样本的特征，线性层的神经元数量至少需要上百的规模。经过尝试和研究，我认为上述网络规模比较合适。一方面，增加规模很难提高模型在验证集上的准确率，并且会显著降低训练效率；另一方面，减小模型规模可能导致模型无法充分学习图像特征。

对于权重参数 `W` 和偏置参数 `b` 的初始化：最初我将 `W` 设置为0到1的均匀随机数， `b` 初始设置为0，发现训练效果不佳，大致只能到达75%左右的准确率。

使用以下方法后，准确率大幅度提升：

1. `W` 初始化为 `[-1,1]` 的随机数 / `sqrt(连接的神经元个数)`
2. `b` 初始化为 最后一层 `[-0.2,+0.2]` 其他层 `[-1,0]`

在选择激活函数方面， `sigmoid` 函数涉及指数运算，反向传播求误差梯度时计算量较大。 `ReLU` 函数，只需与0进行比较即可完成激活函数的计算与反向传播的求导，计算量显著降低，可以提高训练效率。

对反向传播算法的理解

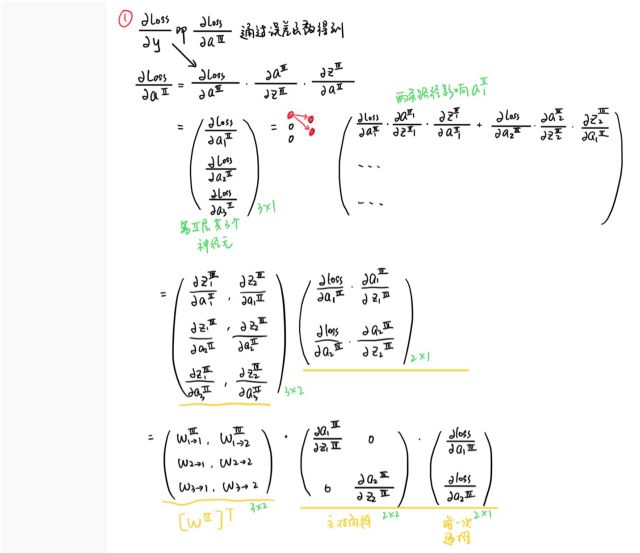
反向传播算法的基本思想是通过链式法则，将网络的损失函数对于每个参数的导数计算出来，然后根据导数更新参数，从而使得网络的输出更好地逼近目标值。

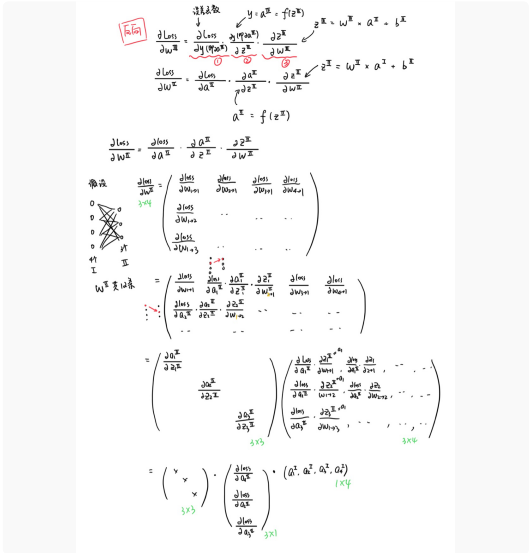
包含以下几个步骤：

1. 前向传播：将训练数据输入网络，通过各个层的线性变换和激活函数计算得到网络的输出。
2. 计算损失：将网络输出与真实标签进行比较，计算损失函数的值，用于衡量网络的输出与真实值之间的差距。
3. 反向传播：根据损失函数对于网络输出的导数，通过链式法则逐层计算每个参数的导数。从输出层开始，将导数乘以激活函数的导数，然后将结果传递到上一层，依次计算每层的参数的导数。
4. 参数更新：根据参数的导数和学习率等参数，使用梯度下降的优化算法更新网络中的参数，使得损失函数的值逐渐减小。
5. 重复迭代：通过不断重复前向传播、反向传播和参数更新的过程，逐渐优化网络的参数，使得网络输出与真实标签更加接近，达到最优的模型。

在Python程序中，使用`for`循环进行训练效率较低，关键在于如何将上述过程转换为矩阵计算，并使用`numpy`库提高训练效率。

推导过程如下：





Part2. 卷积神经网络

代码基本结构

文件目录结构

```
│
├─ data_manager.py # 训练与测试数据管理
├─ network.py # CNN网络结构
├─ train.py # 训练函数
├─ test.py # 测试函数
├─ classification # 存储数据及模型
│  ├─ data
│  │  ├─ testing.pkl
│  │  ├─ training.pkl
│  │  └─ validation.pkl
│  ├─ data_bmp
│  └─ model.pth
└─
```

代码基本结构

- **network.py**
  CNN网络结构

▼ **LeNet-5 卷积神经网络定义**

1. `__init__`方法：该类继承自`torch.nn.Module`类，利用`pytorch`库的相关函数，构建了包括卷积层、池化层、全连接层的LeNet-5 卷积神经网络。
2. `forward`方法：将张量`x`输入网络，依次经过7层返回输出。

▼ **`train.py`**

训练函数文件

首先从`data_manager`中读取训练、验证和测试数据集。之后定义了损失函数为交叉熵损失函数、优化器为随机梯度下降(SGD)、并定义了一个学习率调度器，用于动态调整学习率。

训练过程的每个`epoch`中，再分为多个`batch`。在每个批次中，先将梯度清零、计算模型的输出、计算损失函数、进行反向传播、更新模型参数。

在每个`epoch`结束后，使用验证数据集来评估模型的准确率。首先将模型设置为评估模式，然后使用`torch.no_grad()`上下文管理器来关闭梯度计算。将验证集输入模型，最后输出当前训练轮次的损失和验证集准确率，然后使用学习率调度器来动态调整学习率。

在训练全部结束后，将训练好的模型保存到文件中。

▼ **`test.py`**

测试函数文件

首先从`data_manager`中加载测试数据集，并使用`torch.load()`函数加载训练好的模型参数。将模型设置为评估模式，并使用`torch.no_grad()`上下文管理器来关闭梯度计算。

在测试过程中，通过测试数据集的模型输出，使用`torch.max()`函数来获取每个样本的预测标签，并统计正确预测的样本数。最后输出测试集的总样本数、正确预测的样本数和准确率。

设计实验改进网络并论证

如图为`batch_size=64`，`epoch=50`时的训练数据。

可发现：

(1) 超过某一轮次后，准确率基本不变。

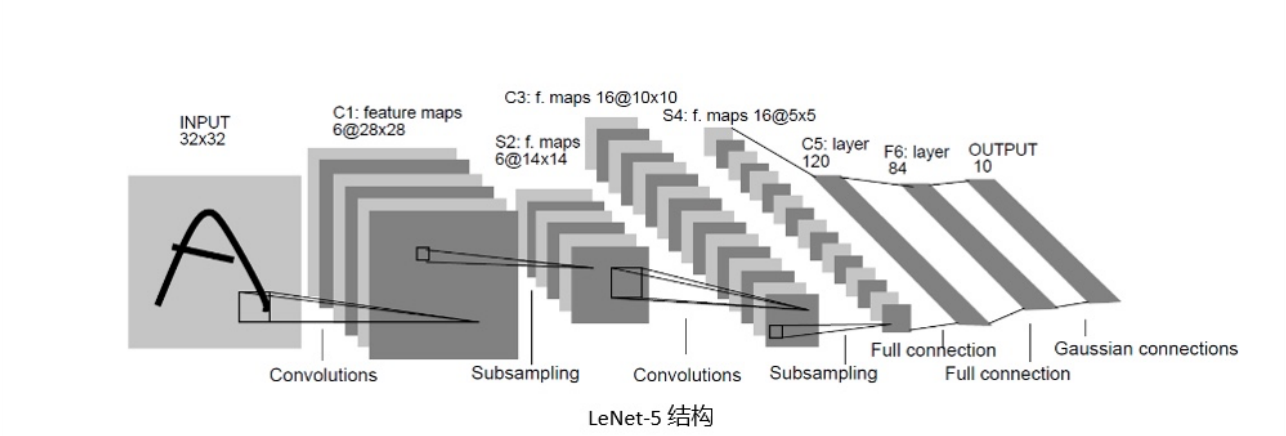
(2) 对于`batch_size=64`而言，准确率为15%，效果不佳。

```
Epoch: 38, Train Loss: 2.451 Validation Accuracy: 15.323 %
Epoch: 39, Train Loss: 2.466 Validation Accuracy: 15.323 %
Epoch: 40, Train Loss: 2.452 Validation Accuracy: 15.323 %
Epoch: 41, Train Loss: 2.458 Validation Accuracy: 15.323 %
Epoch: 42, Train Loss: 2.461 Validation Accuracy: 15.323 %
Epoch: 43, Train Loss: 2.460 Validation Accuracy: 15.323 %
Epoch: 44, Train Loss: 2.450 Validation Accuracy: 15.323 %
Epoch: 45, Train Loss: 2.459 Validation Accuracy: 15.323 %
Epoch: 46, Train Loss: 2.449 Validation Accuracy: 15.323 %
Epoch: 47, Train Loss: 2.460 Validation Accuracy: 15.323 %
Epoch: 48, Train Loss: 2.453 Validation Accuracy: 15.323 %
Epoch: 49, Train Loss: 2.460 Validation Accuracy: 15.323 %
```

减小`batch_size`的规模，发现减小为8时，准确率为`97.98%`。但此时进一步减小`batch_size`会导致训练效率大幅下降，因此最终选择`batch_size`为8。

对于`epoch`而言，在`batch_size`不变的情况下，超过某一轮次之后，准确率基本保持不变。因此`epoch`无需设置过大，对于`batch_size=8`而言，`epoch`为25较好。

对网络设计的理解



LeNet-5 结构

LeNet-5是一种经典的卷积神经网络结构，它由7层神经网络组成，包括两个卷积层、两个池化层和三个全连接层。

输入: 1@28*28
Padding: 大小2 -> 1@32*32
卷积: 卷积核6@5*5 步长1 -> 6@(32-5+1)*(32-5+1) -> 6@28*28
池化: 核大小2*2 步长2 (无重叠) -> 6@14*14

1. 卷积层，使用一个5x5的卷积核对输入图像进行卷积操作，输出特征图。同时使用ReLU作为激活函数，提取图像的特征。
2. 池化层，使用2x2的最大池化操作对特征图进行下采样，减少特征图的尺寸，保留主要特征。

输入: 6@14*14
卷积: 卷积核16@5*5 步长1 -> 16@(14-5+1)*(14-5+1) -> 16@10*10
池化: 核大小2*2 步长2 (无重叠) -> 16@5*5

3. 卷积层，使用16个5x5的卷积核对第二层的池化结果进行卷积操作，提取更高级的特征。
4. 池化层，同样使用2x2的最大池化操作对特征图进行下采样。

输入:16@5*5 输出:120

5. 全连接层，将池化后的特征图展平为一维向量，并连接到一个全连接层上，进行线性变换操作。

输入:120 输出:84

6. 全连接层，再次进行线性变换，并使用ReLU作为激活函数。

输入:84 输出:12

7. 全连接层，输出最终的分类结果。

每一层的作用：

- 卷积层通过卷积操作提取图像的特征。
- 池化层进行下采样，减少特征图的尺寸。
- 全连接层将特征图展平并进行线性变换操作。
- 激活函数用于引入非线性性，增强网络的表达能力。

因为卷积核一般较小，卷积神经网络注重提取局部的信息。而在全连接神经网络中，所有神经元都是和上一层所有神经元相连，全连接神经网络更加关注全局的特征。对于图像分类的任务，因为局部特征可以更好地描述图像的细节和特点，而全局特征则更容易受到图像整体的变化和噪声的影响。通过提取图像的局部特征，可以更准确地识别和分类图像。因此卷积神经网络更适合处理此类任务。