

A1:手写数字体识别

一、编码过程

1.1 数据处理

1.1.1 读取文件，训练集、测试集、验证集的划分

1.1.2 小批量读取的封装

1.1.3 封装成load_data

1.2 模型设计

1.2.1 softmax分类器

1.2.2 全连接神经网络

1.3 优化方法

二、问题解答

2.1 理解基本的图像识别流程及数据驱动的方法（训练、预测等阶段）

2.2 理解训练集/验证集/测试集的数据划分，如何使用验证数据调整模型的超参数

2.3 实现一个softmax分类器

2.4 实现一个全连接神经网络分类器

2.5 理解不同的分类器之间的区别，以及使用不同的更新方法优化神经网络

附加题

2.6 尝试使用不同的损失函数和正则化方法，观察并分析其对实验结果的影响 (+5 points)

2.6.1 交叉熵损失函数

2.6.2 均方误差损失函数

2.6.2 比较

2.6.3 dropout正则化

2.6.4 比较

2.7 尝试使用不同的优化算法，观察并分析其对训练过程和实验结果的影响

班级：112

学号：2021213490

姓名：王拓

一、编码过程

1.1数据处理

数据处理要做到：

- 读取数据集文件
- 训练集、测试集、验证集的划分
- 封装小批量读取 `mini_batch` 的函数
- 如果有错误的还要进行预处理

1.1.1读取文件，训练集、测试集、验证集的划分

首先是使用的数据集是 `MNIST.json.gz`，同时对于数据集的划分：

```
1  # 声明数据集文件位置
2  datafile = 'mnist .json.gz'
3  print(f'加载数据从{datafile}')
```

Python |

```
4  # 加载json数据文件
5  data = json.load(gzip.open(datafile))
6  print('数据集加载完毕')
7  # 读取到的数据区分训练集，验证集，测试集
8  train_set, val_set, eval_set = data
9
10 # 观察训练集数据
11 imgs, labels = train_set[0], train_set[1]
12 print("训练数据集数量：", len(imgs))
13
14 # 观察验证集数量
15 imgs, labels = val_set[0], val_set[1]
16 print("验证数据集数量：", len(imgs))
17
18 # 观察测试集数量
19 imgs, labels = eval_set[0], eval_set[1]
20 print("测试数据集数量：", len(imgs))
21 print(len(imgs[0]))
22 print(len(labels[0]))
```

```
加载数据从mnist.json.gz
数据集加载完毕
训练数据集数量: 50000
验证数据集数量: 10000
测试数据集数量: 10000
784
```

数据集的划分为 `train_set: 50000` `val_set: 10000` `eval_set: 10000`

1.1.2 小批量读取的封装

▼ data_generator()

Python |

```
1  # 定义数据生成器
2  def data_generator():
3      if mode == 'train':
4          # 训练模式下要打乱数据
5          random.shuffle(index_list)
6          imgs_list = []
7          labels_list = []
8      for i in index_list:
9          # 处理数据
10         img = np.array(imgs[i]).astype('float32')
11         label = np.array(labels[i]).astype('float32')
12         imgs_list.append(img)
13         labels_list.append(label)
14         if len(imgs_list) == BATCHSIZE:
15             # 获得一个BATCHSIZE的数据
16             yield np.array(imgs_list), np.array(labels_list)
17             # 数据清空, 加载下一批
18             imgs_list = []
19             labels_list = []
20
21         # 对于小于BATCHSIZE的
22         if len(imgs_list) > 0:
23             yield np.array(imgs_list), np.array(labels_list)
```

以上定义了一个返回 `BATCHSIZE` 大小的函数, 同时对于训练打乱数据顺序

1.1.3 封装成 `load_data`

```
1  # 数据加载
2  def load_data(mode='train', batch_size = 100):
3      """
4      加载数据函数
5      传入两个参数mode,batch_size
6      """
7      datafile = './mnist.json.gz'
8      print(f'加载数据从{datafile}')
9      # 加载json数据文件
10     data = json.load(gzip.open(datafile))
11     print(f'加载数据集完毕')
12
13     # 根据输入的参数进行数据的划分
14     train_set, val_set, eval_set = data
15     if mode == 'train':
16         imgs, labels = train_set[0], train_set[1]
17     elif mode == 'valid':
18         imgs, labels = val_set[0], val_set[1]
19     elif mode == 'eval':
20         imgs, labels = eval_set[0], eval_set[1]
21     else:
22         raise Exception("mode 参数必须为['train','valid', 'eval']")
23     print(f'加载的数据集数量{len(imgs)}')
24
25     # 获取数据集的长度
26     imgs_length = len(imgs)
27
28     # 给每一条数据编号
29     index_list = list(range(imgs_length))
30     BATCHSIZE = batch_size
31
32     # 定义数据生成器
33     def data_generator():
34         if mode == 'train':
35             # 训练模式下要打乱数据
36             random.shuffle(index_list)
37             imgs_list = []
38             labels_list = []
39             for i in index_list:
40                 # 处理数据
41                 img = np.array(imgs[i]).astype('float32')
42                 label = np.array(labels[i]).astype('float32')
43                 imgs_list.append(img)
44                 labels_list.append(label)
45                 if len(imgs_list) == BATCHSIZE:
```

```

46         # 获得一个BATCHSIZE的数据
47         yield np.array(imgs_list), np.array(labels_list)
48         # 数据清空，加载下一批
49         imgs_list = []
50         labels_list = []
51
52     # 对于小于BATCHSIZE的
53     if len(imgs_list) > 0:
54         yield np.array(imgs_list), np.array(labels_list)
55     return data_generator

```

参数：

- `mode`：网络的模式，默认为 `train`。还有 `eval`、`valid`。
- `batch_size`：批的大小。

函数功能

- 传入参数，返回一个数据生成器。

1.2 模型设计

1.2.1 softmax分类器

- 一个全连接层 `in_feature=784`，`out_feature=10`
- 一个softmax函数
- 交叉熵损失函数

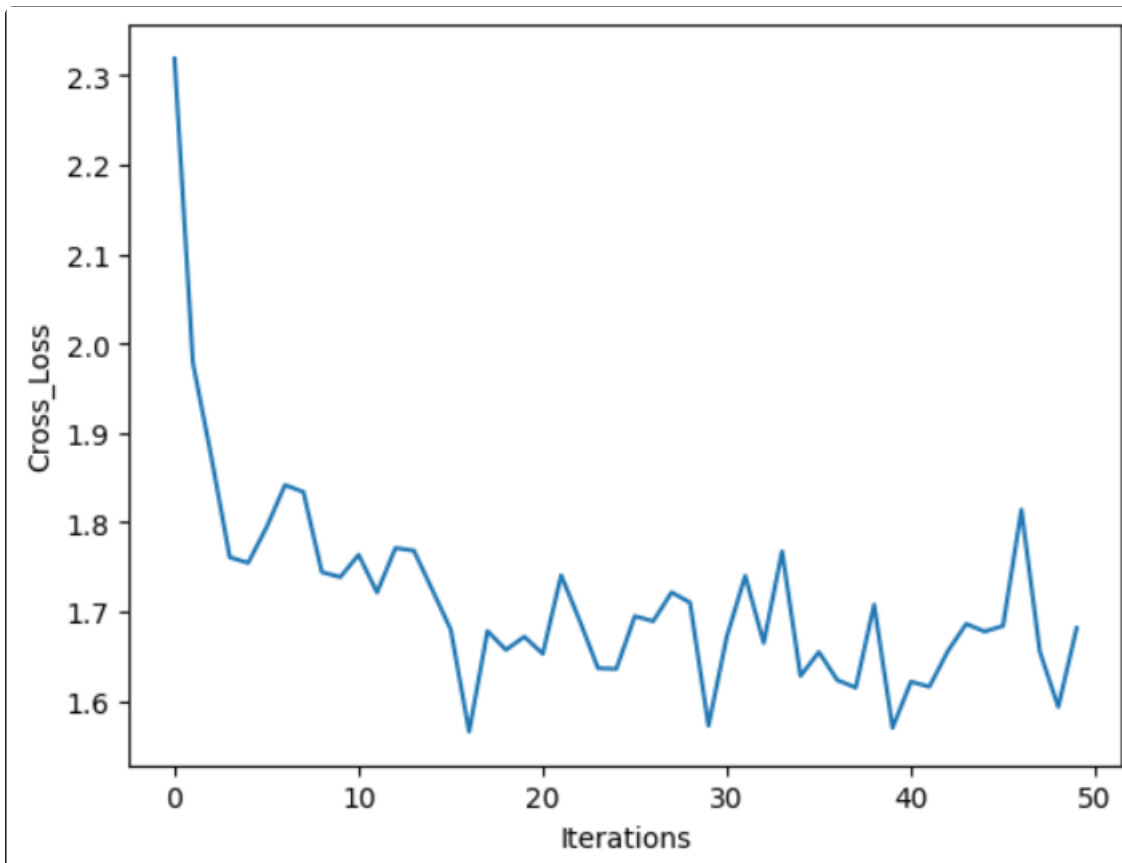
核心代码

```

1 class SOFTMAX_NET(paddle.nn.Layer):
2     def __init__(self):
3         super(SOFTMAX_NET, self).__init__()
4         self.fc1 = Linear(in_features=784, out_features=10)
5
6     def forward(self, inputs):
7         inputs = paddle.reshape(inputs, [inputs.shape[0], 784])
8         outputs1 = self.fc1(inputs)
9         outputs_final = F.softmax(outputs1)
10        return outputs_final

```

损失函数



测试集准确率

```
1 acc_of_softmax = evaluation(model_softmax, test_loader)
2 print(f'acc_of_softmax:{acc_of_softmax}')
✓ 1.9s
acc of softmax:0.8304139971733093
```

1.2.2全连接神经网络

paddle框架全连接神经网络

```
1 class MNIST(paddle.nn.Layer):
2     def __init__(self):
3         super(MNIST, self).__init__()
4         # 还是两层隐藏层的网络
5         self.fc1 = Linear(in_features=784, out_features=200)
6         self.fc2 = Linear(in_features=200, out_features=200)
7         # 输出十个维度，使用交叉熵损失函数
8         self.fc3 = Linear(in_features=200, out_features=10)
9
10    def forward(self, inputs):
11        inputs = paddle.reshape(inputs, [inputs.shape[0], 784])
12        outputs1 = self.fc1(inputs)
13        outputs1 = F.relu(outputs1)
14        outputs2 = self.fc2(outputs1)
15        outputs2 = F.relu(outputs2)
16        outputs3 = self.fc3(outputs2)
17        # 使用softmax激活
18        outputs_final = F.softmax(outputs3)
19        return outputs_final
```

全连接神经网络，我才用numpy手搓前向信号传播和误差反向传播，同时根据验证集进行参数的调优

```
1  # 全连接神经网络的搭建, 三层
2  import numpy as np
3
4  class ThreeLayerNetwork(object):
5      def __init__(self, input_num, hidden1_num, hidden2_num, output_num):
6          # 使用Xavier初始化权重
7          self.W1 = np.random.randn(input_num, hidden1_num) / np.sqrt(input
            _num)
8          self.b1 = np.zeros((1, hidden1_num))
9          self.W2 = np.random.randn(hidden1_num, hidden2_num) / np.sqrt(hid
            den1_num)
10         self.b2 = np.zeros((1, hidden2_num))
11         self.W3 = np.random.randn(hidden2_num, output_num) / np.sqrt(hidd
            en2_num)
12         self.b3 = np.zeros((1, output_num))
13
14     def forward(self, X):
15         self.z1 = np.dot(X, self.W1) + self.b1
16         self.a1 = self.sigmoid(self.z1)
17         self.z2 = np.dot(self.a1, self.W2) + self.b2
18         self.a2 = self.sigmoid(self.z2)
19         self.z3 = np.dot(self.a2, self.W3) + self.b3
20         return self.z3
21
22     def sigmoid(self, x):
23         return 1 / (1 + np.exp(-x))
24
25     def sigmoid_derivative(self, x):
26         return self.sigmoid(x) * (1-self.sigmoid(x))
27
28     def loss2(self, y_pred, y_true):
29         error = y_pred - y_true
30         num_sample = y_pred.shape[0]
31         cost = np.sum(error ** 2) / num_sample
32         return cost
33
34     def gradient(self, X, y_pred, y_true):
35         m = y_pred.shape[0]
36         delta3 = (y_pred - y_true)
37         dW3 = np.dot(self.a2.T, delta3) / m
38         db3 = np.sum(delta3, axis=0) / m
39         delta2 = np.dot(delta3, self.W3.T) * self.sigmoid_derivative(self
            .z2)
40         delta2 = np.dot(self.a1.T, delta2) / m
41         db2 = np.sum(delta2,axis=0) / m
```



```

42         delta1 = np.dot(delta2, self.W2.T) * self.sigmoid_derivative(self
43         .z1)
44         dW1 = np.dot(X.T, delta1) / m
45         db1 = np.sum(delta1, axis=0) / m
46         return dW1, db1, dW2, db2, dW3, db3
47
48     def update(self, dW1, db1, dW2, db2, dW3, db3, eta):
49         self.W1 -= eta * dW1
50         self.b1 -= eta * db1
51         self.W2 -= eta * dW2
52         self.b2 -= eta * db2
53         self.W3 -= eta * dW3
54         self.b3 -= eta * db3
55
56     def train(self, num_epochs, batch_size=10, eta = 0.01):
57         losses = []
58         train_loader = load_data('train', batch_size=batch_size)
59         for epoch_id in range(num_epochs):
60             for batch_id, data in enumerate(train_loader()):
61                 images, labels = data
62                 labels = labels.reshape(batch_size, 1)
63                 # 信号的前向传播
64                 predicts = self.forward(images)
65                 # 损失
66                 cost = self.loss2(predicts, labels)
67                 # 记录损失
68                 if batch_id % 200 == 0:
69                     losses.append(cost)
70                     print(f'epoch{epoch_id}, batch{batch_id}, loss{cost}')
71
72             # 梯度的反向传播
73             W1, b1, W2, b2, W3, b3 = self.gradient(images, predicts
74             , labels)
75             self.update(W1, b1, W2, b2, W3, b3, eta=eta)
76
77
78         return losses
79
80     def trainAndvalid(self, num_epochs, batch_size, eta):
81         """
82         进行超参数的搜索
83         """
84         train_losses = []
85         valid_losses = []
86
87         train_loader = load_data('train', batch_size=batch_size)

```

```

88         valid_loader = load_data('valid',batch_size=batch_size)
89     for epoch_id in range(num_epochs):
90         for batch_id, data in enumerate(train_loader()):
91             images, labels = data
92             labels = labels.reshape(batch_size, 1)
93             # 信号的前向传播
94             predicts = self.forward(images)
95             # 损失
96             cost = self.loss2(predicts, labels)
97             # 记录损失
98             if batch_id % 200 == 0:
99                 train_losses.append(cost)
100                 print(f'epoch{epoch_id}, batch{batch_id}, train_loss{
101 cost}')
102             # 梯度的反向传播
103             W1, b1, W2, b2, W3, b3 = self.gradient(images, predicts
104 , labels)
105             self.update(W1, b1, W2, b2, W3, b3, eta=eta)
106
107         # 每个epoch结束后进行验证集损失
108         valid_loss = 0
109         for data in valid_loader():
110             valid_imgs, valid_labels = data
111             valid_labels = valid_labels.reshape(batch_size, 1)
112             valid_predicts = self.forward(valid_imgs)
113             valid_loss += self.loss2(valid_predicts, valid_labels)
114         valid_losses.append(valid_loss)
115         print(f'epoch{epoch_id}, valid_loss{valid_loss}')
116
117     return train_losses, valid_losses
118
119     def predict(self, X):
120         z1 = np.dot(X, self.W1) + self.b1
121         a1 = self.sigmoid(z1)
122         z2 = np.dot(a1, self.W2) + self.b2
123         a2 = self.sigmoid(z2)
124         z3 = np.dot(a2, self.W3) + self.b3
125         return z3

```

训练集要优化的超参数：

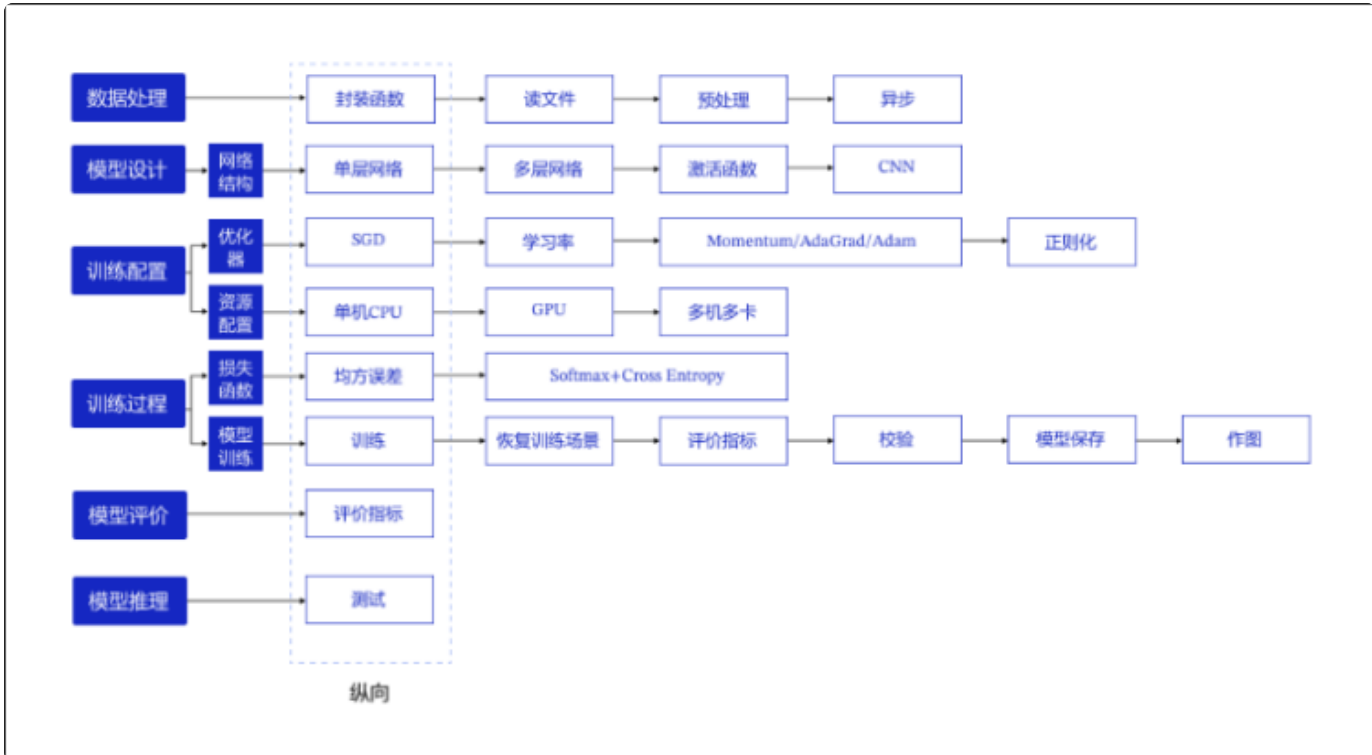
- 学习率：设置0-1，每次相隔0.1
- 节点个数：50-700，每次隔50
- epoch: 1-20,相隔1

1.3优化方法

见2.5和2.6

二、问题解答

2.1 理解基本的图像识别流程及数据驱动的方法（训练、预测等阶段）



图像识别的基本过程就是一、编码过程，包含以下几点：

- 数据处理：读取文件；进行预处理：比如归一化，旋转、通道调整等；之后封装函数，实现 `data_loader()`，批量读取。
- 模型设计：比如全连接神经网络：网络结构，损失函数，激活函数。
- 训练配置：优化器、学习率设置、正则化。还有多卡训练。
- 训练：训练对评价指标的观察、对模型进行保存。
- 模型推理：进行应用。

2.2 理解训练集/验证集/测试集的数据划分，如何使用验证数据调整模型的超参数

- 训练集：训练集是用来训练模型的参数，比如神经网络的 w ， b 以便尽可能准确地拟合数据。
- 验证集：用来调整超参数，以获得最佳的超参数配置。在训练过程中，模型通常需要根据不同的超参数设置进行多次训练。验证集不用于训练
- 测试集：模拟模型在实际应用中遇到的新数据，评估模型的泛化性能。
- 训练集用于构建模型，验证集用于调优和选择最佳超参数，而测试集用于最终评估模型性能。

验证集不是必须的，如果数据过少，可以使用k折交叉验证，从而不用验证集。

使用验证数据调整模型超参数：

- 选择一组基础的超参数：学习率，节点数量，epoch。
- 用训练集训练网络。
- 使用验证数据验证损失。
- 更新超参数。
- 重新训练
- 继续验证
- 重复以上过程，选择验证数据中表现效果较好的就超参数进行网络训练。

对于手写数字识别，学习率 0.1 ，节点数量 $150-200$ 比较合适。

代码：

网络类中封装的训练和超参数调优的函数

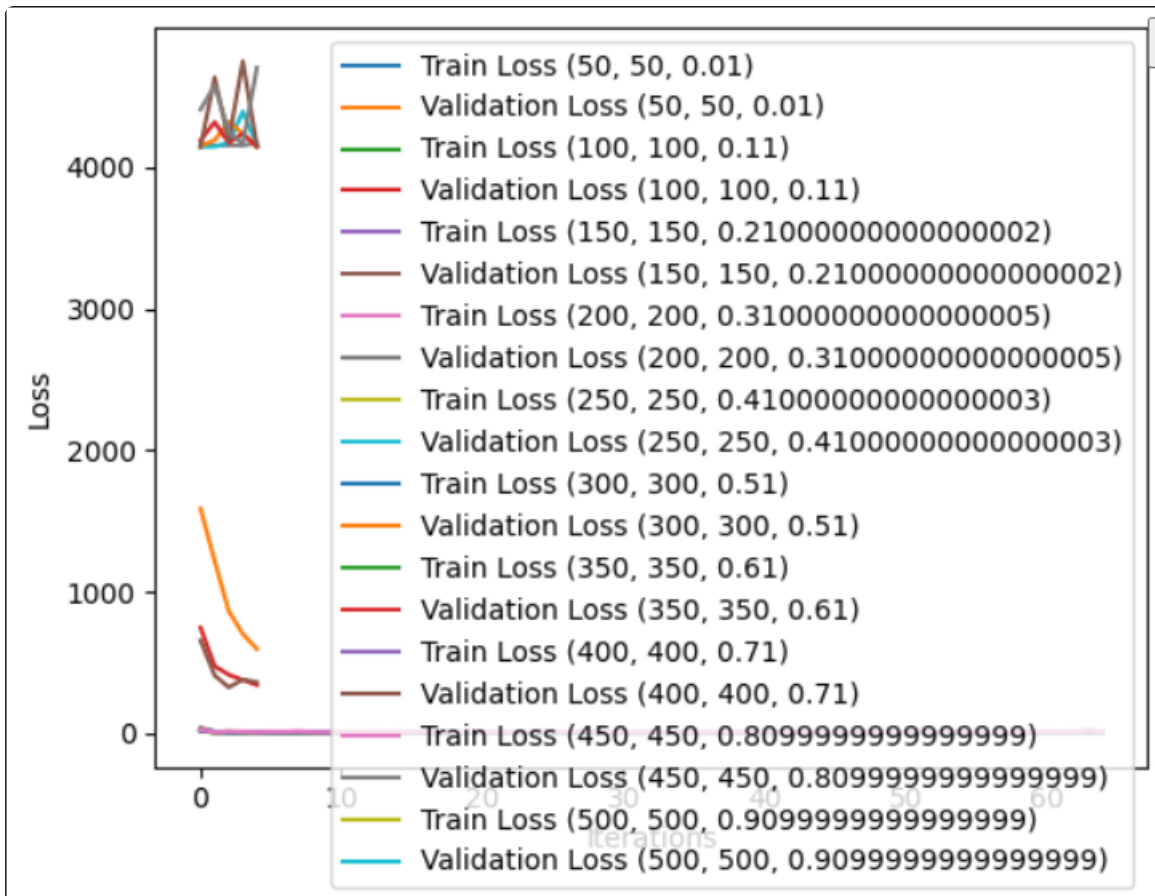
```

1  def trainAndvalid(self, num_epochs, batch_size, eta):
2      """
3      进行超参数的搜索
4      """
5      train_losses = []
6      valid_losses = []
7
8      train_loader = load_data('train', batch_size=batch_size)
9      valid_loader = load_data('valid', batch_size=batch_size)
10     for epoch_id in range(num_epochs):
11         for batch_id, data in enumerate(train_loader()):
12             images, labels = data
13             labels = labels.reshape(batch_size, 1)
14             # 信号的前向传播
15             predicts = self.forward(images)
16             # 损失
17             cost = self.loss2(predicts, labels)
18             # 记录损失
19             if batch_id % 200 == 0:
20                 train_losses.append(cost)
21                 print(f'epoch{epoch_id}, batch{batch_id}, train_loss{cost}')
22             # 梯度的反向传播
23             W1, b1, W2, b2, W3, b3 = self.gradient(images, predicts, labels)
24             self.update(W1, b1, W2, b2, W3, b3, eta=eta)
25
26         # 每个epoch结束后进行验证集损失
27         valid_loss = 0
28         for data in valid_loader():
29             valid_imgs, valid_labels = data
30             valid_labels = valid_labels.reshape(batch_size, 1)
31             valid_predicts = self.forward(valid_imgs)
32             valid_loss += self.loss2(valid_predicts, valid_labels)
33         valid_losses.append(valid_loss)
34         print(f'epoch{epoch_id}, valid_loss{valid_loss}')
35
36     return train_losses, valid_losses

```

开始训练时进行for循环实现多次训练，实现超参数调优

```
1  # 增加了超参数的优化
2  hidden1_num = 50
3  hidden2_num = 50
4  lr = 0.01
5  import matplotlib.pyplot as plt
6
7  for i in range(10):
8      # 创建网络
9      net = ThreeLayerNetwork(input_num=784, hidden1_num=hidden1_num, hidden
10                               2_num=hidden2_num, output_num=1)
11
12      # 启动训练
13      train_losses, valid_losses = net.trainAndvalid(num_epochs=5, batch_size=20, eta=lr)
14
15      # 绘制训练损失
16      plot_x_train = np.arange(len(train_losses))
17      plot_y_train = np.array(train_losses)
18      plt.plot(plot_x_train, plot_y_train, label=f'Train Loss ({hidden1_num}
19              , {hidden2_num}, {lr})')
20
21      # 绘制验证损失
22      plot_x_valid = np.arange(len(valid_losses))
23      plot_y_valid = np.array(valid_losses)
24      plt.plot(plot_x_valid, plot_y_valid, label=f'Validation Loss ({hidden1
25              _num}, {hidden2_num}, {lr})')
26
27      hidden1_num += 50
28      hidden2_num += 50
29      lr += 0.1
30  plt.xlabel('Iterations')
31  plt.ylabel('Loss')
32  plt.legend()
33  plt.show()
```



超参数组合示意图

2.3 实现一个softmax分类器

见1.2.1

2.4 实现一个全连接神经网络分类器

见1.2.2

2.5 理解不同的分类器之间的区别，以及使用不同的更新方法优化神经网络

1.结构差异：

- `softmax`：是全连接层和 `softmax` 激活函数
- 全连接：多个隐藏层和输出层。

2.输出差异：

- `softmax`：输出的是一个概率分布。
- 全连接：MLP的输出通常不是概率分布，而是一个连续的数值向量，可以用于回归任务或者二元分类任务。

3.损失函数：

- `softmax`：交叉熵损失函数
- 全连接：取决于是回归还是分类任务。

附加题

2.6 尝试使用不同的损失函数和正则化方法，观察并分析其对实验结果的影响 (+5 points)

模型参数：

- 两个隐藏层，200个节点。
- `Relu` 激活函数交叉熵损失

训练配置：

- 学习率：0.1
- epoch：10

2.6.1 交叉熵损失函数

▼ 核心代码

Python |

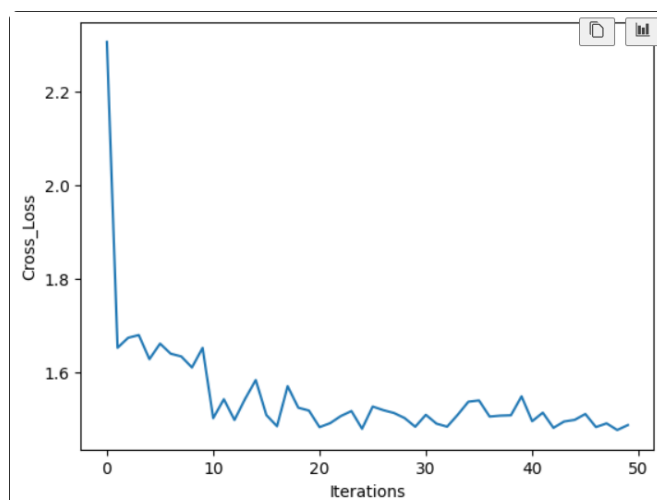
```
1      # 使用softmax激活
2      outputs_final = F.softmax(outputs3)
3      return outputs_final
4
5      # 采用交叉熵损失函数
6      loss = F.cross_entropy(predicts, labels)
7      avg_loss = paddle.mean(loss)
8
```


网络结构

```
Layer (type)      Input Shape      Output Shape      Param #
=====
Linear-13         [[1, 784]]       [1, 200]          157,000
Linear-14         [[1, 200]]       [1, 200]          40,200
Linear-15         [[1, 200]]       [1, 10]           2,010
=====
Total params: 199,210
Trainable params: 199,210
Non-trainable params: 0
=====
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.76
Estimated Total Size (MB): 0.77
=====

{'total_params': 199210, 'trainable_params': 199210}
```

交叉熵损失



测试集准确性

```
1 acc_of_Cross = evaluation(model_cross, test_loader)
2 print(f'acc_of_Cross:{acc_of_Cross}')
```

✓ 0.9s

Python

acc_of_Cross:0.9639729261398315

2.6.2 均方误差损失函数

```

1      # 输出一个维度，使用均方差作为损失
2      self.fc3 = Linear(in_features=200, out_features=1)
3
4      #计算损失，取一个批次样本损失的平均值
5      loss = F.square_error_cost(predicts, labels)

```

网络结构

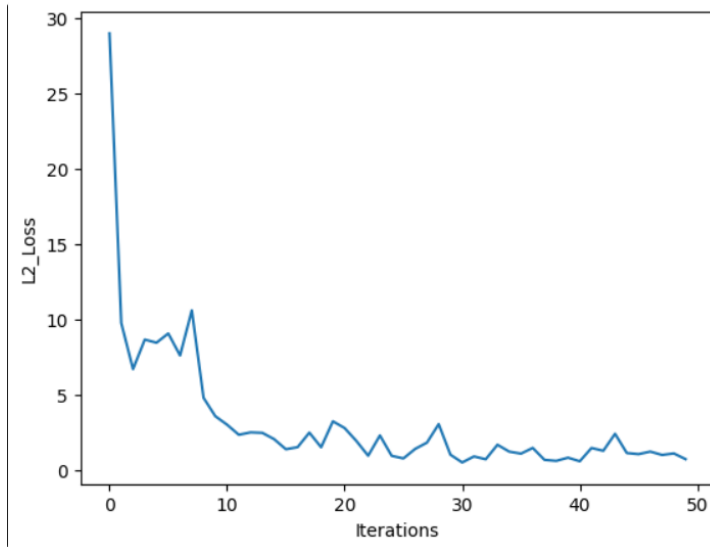
```

Layer (type)      Input Shape      Output Shape      Param #
=====
Linear-19         [[1, 784]]       [1, 200]          157,000
Linear-20         [[1, 200]]       [1, 200]          40,200
Linear-21         [[1, 200]]       [1, 1]            201
=====
Total params: 197,401
Trainable params: 197,401
Non-trainable params: 0
=====
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.75
Estimated Total Size (MB): 0.76
=====

{'total_params': 197401, 'trainable_params': 197401}

```

均方损失



测试集准确度

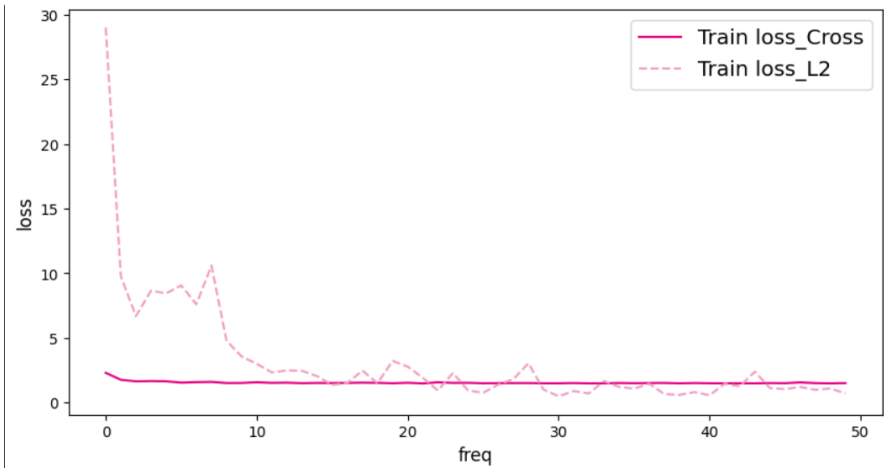
```

1  acc_of_L2 = evaluation(model_L2, test_loader)
2  print(f'acc_of_L2:{acc_of_L2}')
[33]  ✓ 0.9s
...  acc_of_L2:0.09783041477203369

```

2.6.2 比较

训练集损失



- 由于不同的损失量纲，看数值是不行的，但是看**收敛速度**，交叉熵损失函数更快。
- 对于准确率，由于交叉熵使用softmax回归，使得结果更准确。
- 交叉熵损失更适用于**分类问题**，均方误差更适用于**回归问题**

2.6.3 dropout正则化

核心代码

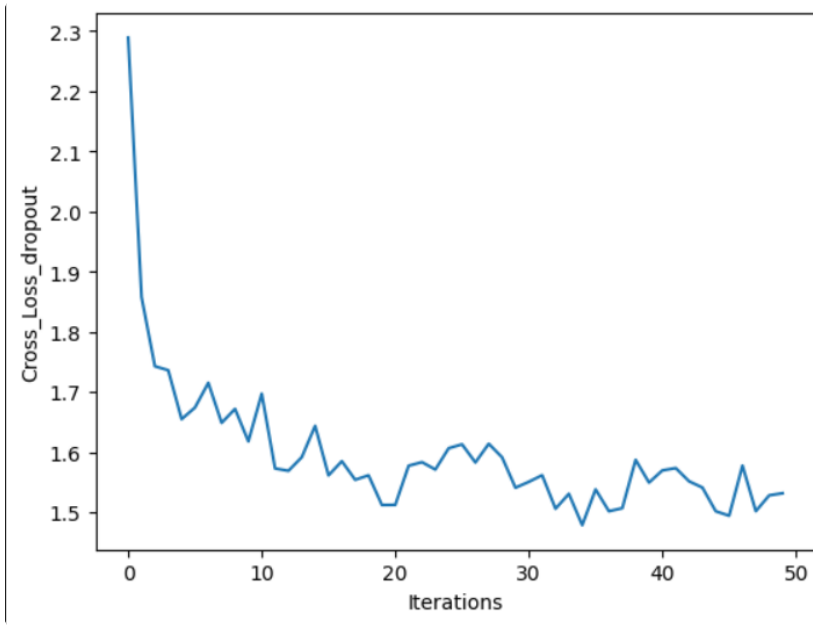
Python

```
1 self.dropout = paddle.nn.Dropout(p=dropout_prob)
2 outputs1 = self.dropout(outputs1)
3
```

网络结构

Layer (type)	Input Shape	Output Shape	Param #
Linear-25	[[1, 784]]	[1, 200]	157,000
Dropout-1	[[1, 200]]	[1, 200]	0
Linear-26	[[1, 200]]	[1, 200]	40,200
Linear-27	[[1, 200]]	[1, 10]	2,010
Total params: 199,210			
Trainable params: 199,210			
Non-trainable params: 0			
Input size (MB): 0.00			
Forward/backward pass size (MB): 0.00			
Params size (MB): 0.76			
Estimated Total Size (MB): 0.77			
{ 'total_params': 199210, 'trainable_params': 199210 }			

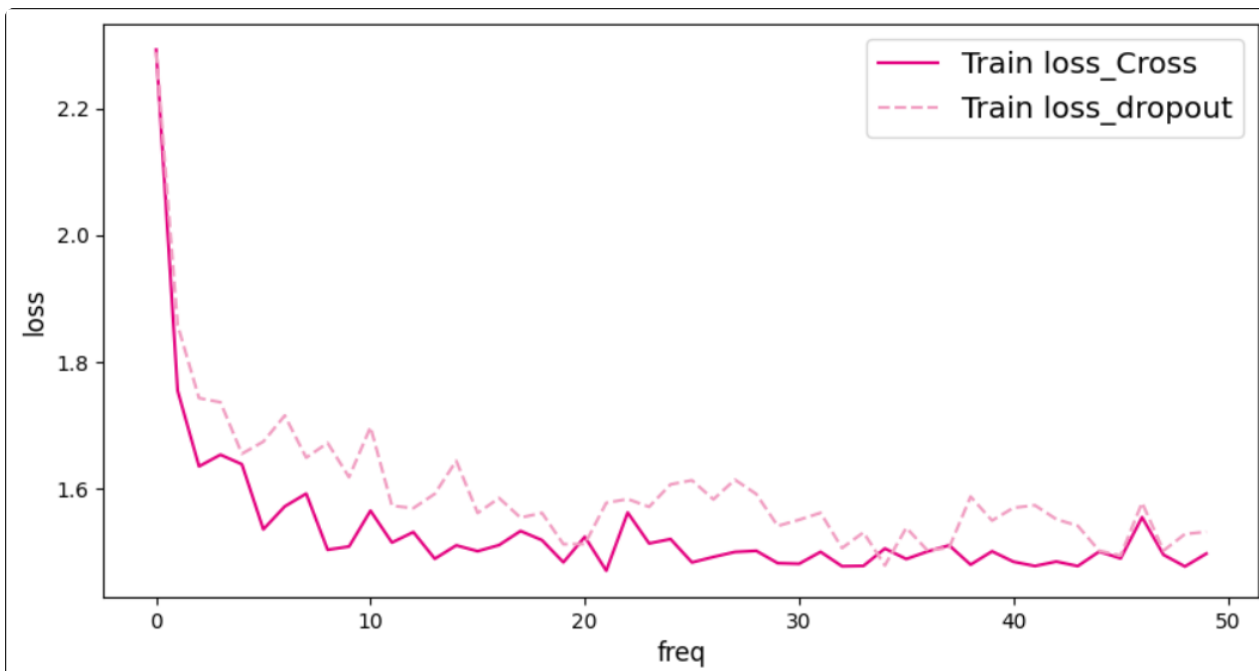
误差



测试集准确率

```
1 acc_of_dropout = evaluation(model_dropout, test_loader)
2 print(f'acc_of_dropout:{acc_of_dropout}')
✓ 0.9s Python
acc_of_dropout:0.9501393437385559
```

2.6.4 比较



可以看出训练速度变快，损失变小。

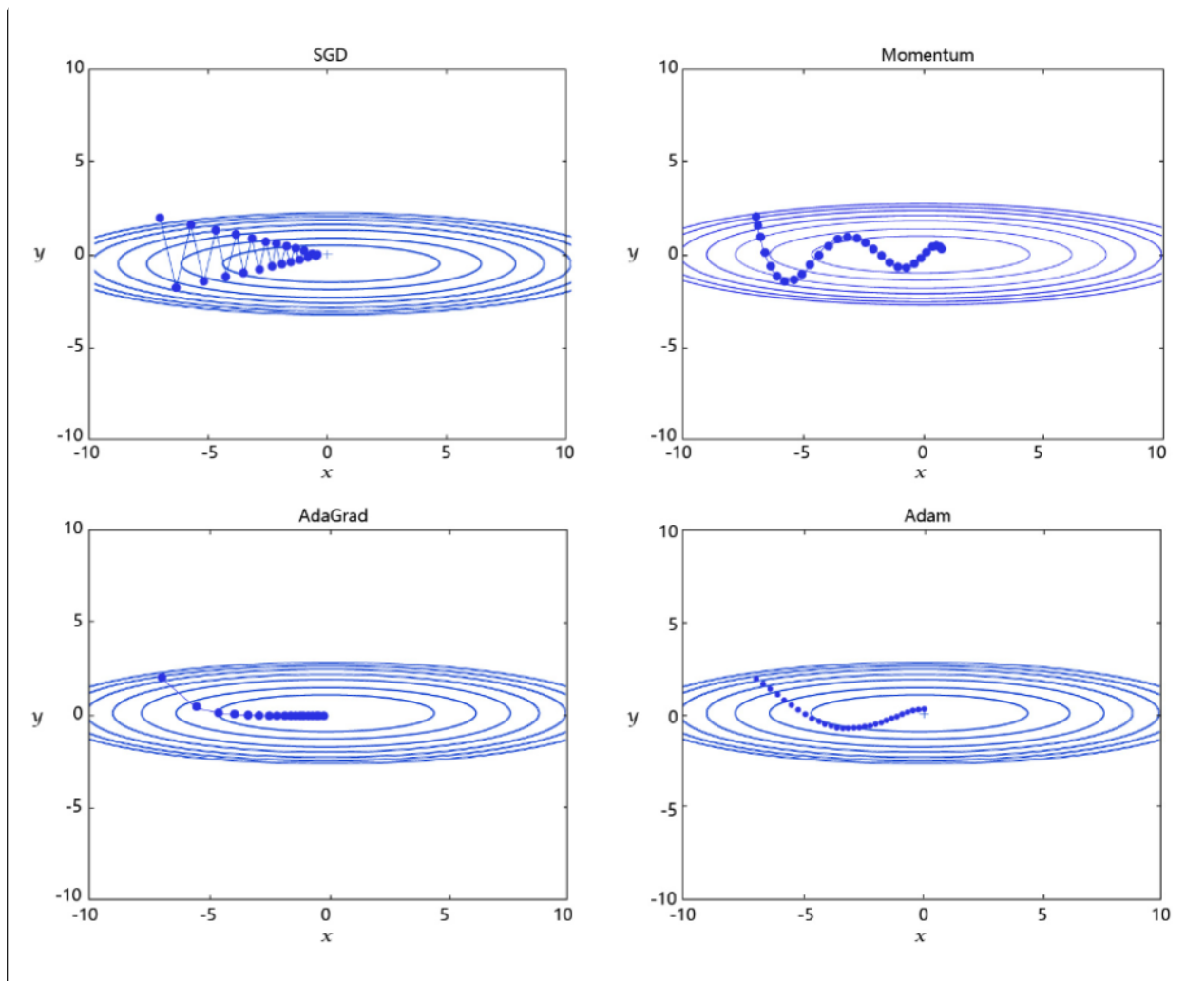
增加dropout后，有以下优点：

- 加快训练速度：因为它降低了每个样本对模型参数的贡献，使得每个批次的计算更加高效。

- 减少过拟合：不过在这个实验上看不出来。
- 提高泛化能力。

2.7 尝试使用不同的优化算法，观察并分析其对训练过程和实验结果的影响

四种学习率优化方法的示意图



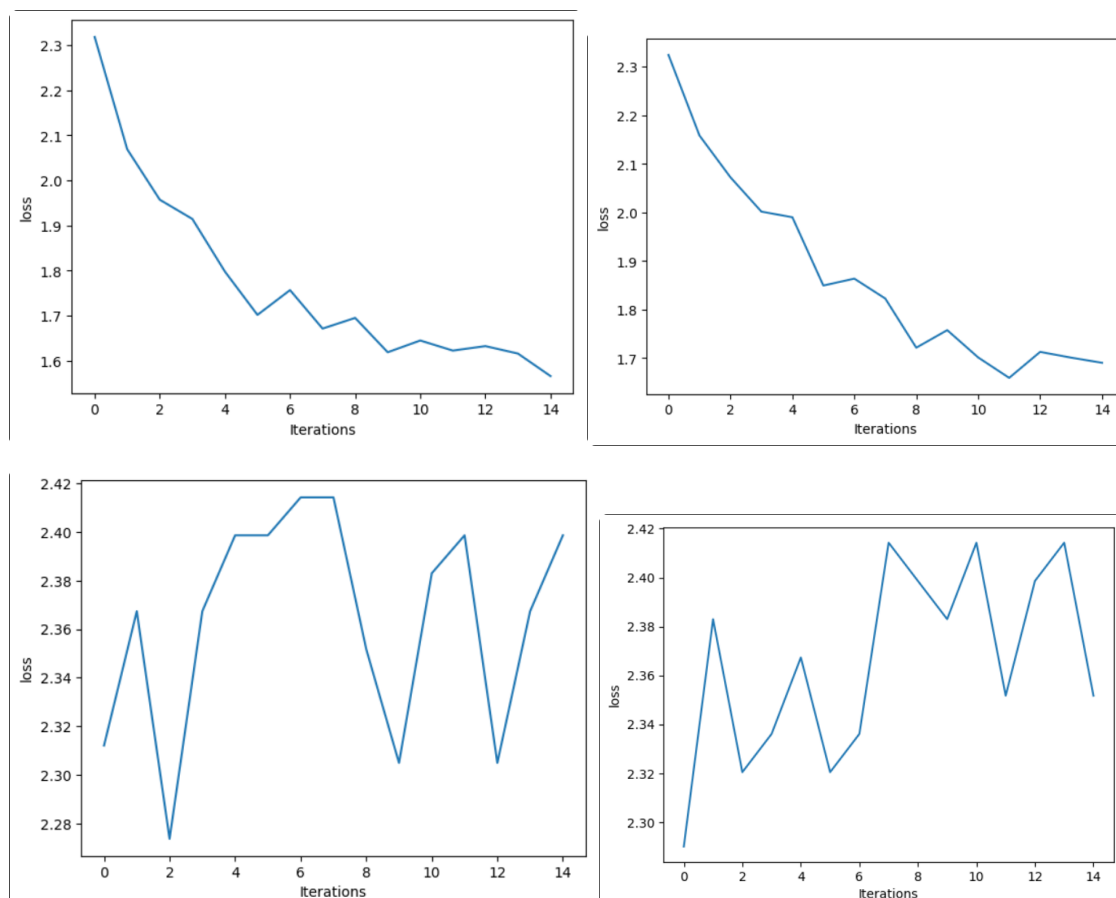
核心代码

```

1  opt = paddle.optimizer.SGD(learning_rate=0.01, parameters=model.parameters())
2  # opt = paddle.optimizer.Momentum(learning_rate=0.01, momentum=0.9, parameters=model.parameters())
3  # opt = paddle.optimizer.Adagrad(learning_rate=0.01, parameters=model.parameters())
4  # opt = paddle.optimizer.Adam(learning_rate=0.01, parameters=model.parameters())

```

损失



对应四种优化算法

- SGD：随机梯度下降算法，每次训练少量数据，抽样偏差导致的参数收敛过程中震荡。
- Momentum：引入物理“动量”的概念，累积速度，减少震荡，使参数更新的方向更稳定。
- AdaGrad：根据不同参数距离最优解的远近，动态调整学习率。学习率逐渐下降，依据各参数变化大小调整学习率。
- Adam：由于动量和自适应学习率两个优化思路是正交的，因此可以将两个思路结合起来，这是当前广泛应用的算法。

比较

- 由于我的初始学习率太小，AdaGrad 和 Adam 学习率下降的快，导致可能参数改变的小，加上EPOCH只有3，没法很好的训练。