A1:手写数字体识别

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

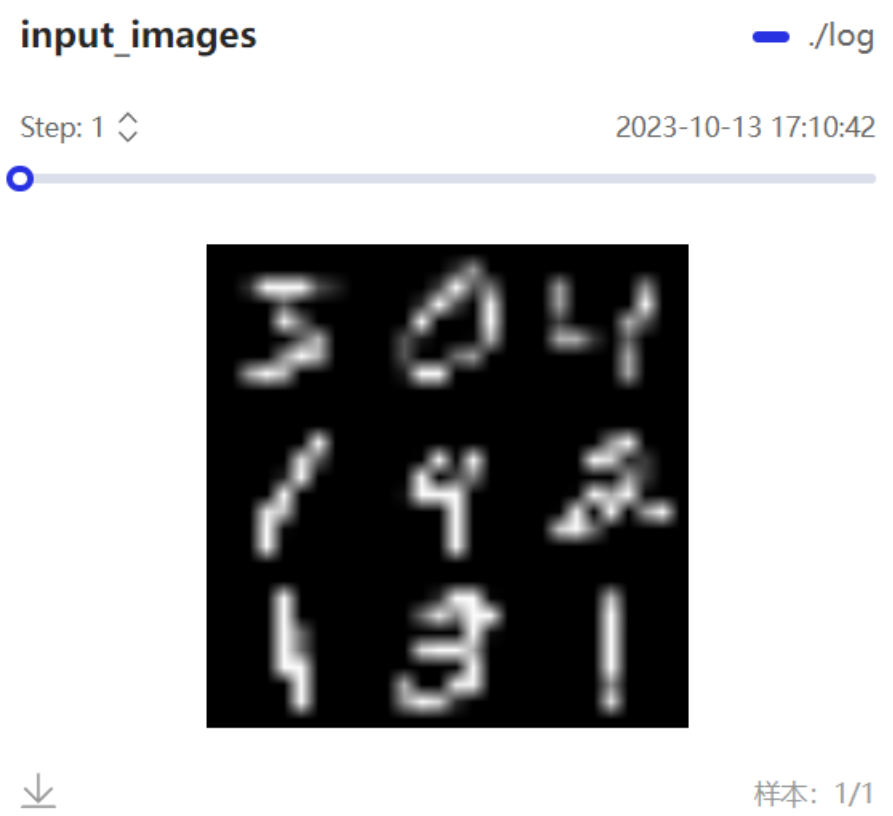
首先是使用的数据集是 `MNIST.json.gz` ，同时对于数据集的划分：

Python

```python
# 声明数据集文件位置
datafile = 'mnist .json.gz'
print(f'加载数据从{datafile}')
# 加载json数据文件
data = json.load(gzip.open(datafile))
print('数据集加载完毕')
# 读取到的数据区分训练集，验证集，测试集
train_set, val_set, eval_set = data

# 观察训练集数据
imgs, labels = train_set[0], train_set[1]
print("训练数据集数量: ", len(imgs))

# 观察验证集数量
imgs, labels = val_set[0], val_set[1]
print("验证数据集数量: ", len(imgs))

# 观察测试集数量
imgs, labels = val= eval_set[0], eval_set[1]
print("测试数据集数量: ", len(imgs))
print(len(imgs[0]))
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` `ebal_set: 10000`



1.1.2 小批量读取的封装

data_generator()

Python

```python
# 定义数据生成器
    def data_generator():
        if mode == 'train':
            # 训练模式下要打乱数据
            random.shuffle(index_list)
        imgs_list = []
        labels_list = []
        for i in index_list:
            # 处理数据
            img = np.array(imgs[i]).astype('float32')
            label = np.array(labels[i]).astype('float32')
            imgs_list.append(img)
            labels_list.append(label)
            if len(imgs_list) == BATCHSIZE:
                # 获得一个BATCHSIZE的数据
                yield np.array(imgs_list),np.array[labels_list]
                # 数据清空，加载下一批
                imgs_list = []
                labels_list = []

        # 对于小于BATCESIZE的
        if len[imgs_list]>0:
            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`

load_data Python

```python
# 数据加载
def load_data(mode='train', batch_size = 100):
    """
    加载数据函数
    传入两个参数mode,batch_size
    """
    datafile = './mnist .json.gz'
    print(f'加载数据从{datafile}')
    # 加载json数据文件
    data = json.load(gzip.open(datafile))
    print(f'加载数据集完毕')

    # 根据输入的参数进行数据的划分
    train_set, val_set, eval_set = data
    if mode == 'train':
        imgs, labels = train_set[0], train_set[1]
    elif mode == 'valid':
        imgs, labels = val_set[0], val_set[1]
    elif mode == 'eval':
        imgs, labels = eval_set[0], eval_set[1]
    else:
        raise Exception("mode 参数必须为['train','valid', 'eval']")
    print(f"加载的数据集数量{len(imgs)}")

    # 获取数据集的长度
    imgs_length = len(imgs)

    # 给每一条数据编号
    index_list = list(range(imgs_length))
    BATCHSIZE = batch_size

    # 定义数据生成器
    def data_generator():
        if mode == 'train':
            # 训练模式下要打乱数据
            random.shuffle(index_list)
        imgs_list = []
        labels_list = []
        for i in index_list:
            # 处理数据
            img = np.array(imgs[i]).astype('float32')
            label = np.array(labels[i]).astype('float32')
            imgs_list.append(img)
            labels_list.append(label)
            if len(imgs_list) == BATCHSIZE:
```

```
                # 获得一个BATCHSIZE的数据
                yield np.array(imgs_list),np.array[labels_list]
                # 数据清空，加载下一批
                imgs_list = []
                labels_list = []

        # 对于小于BATCESIZE的
        if len[imgs_list]>0:
            yield np.array(imgs_list), np.array(labels_list)
    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

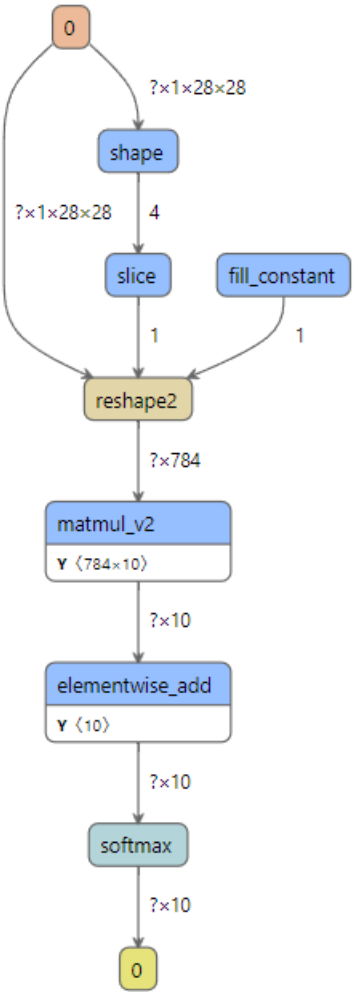
1.2.1softmax分类器

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

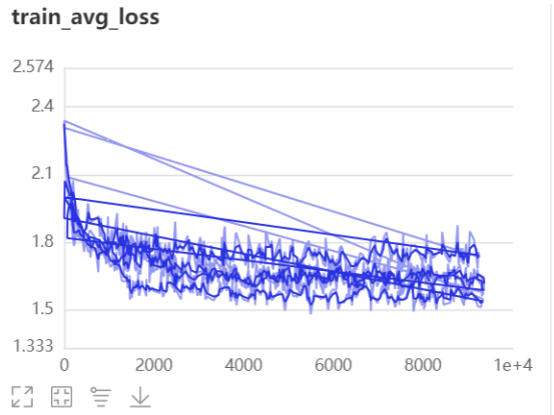
核心代码

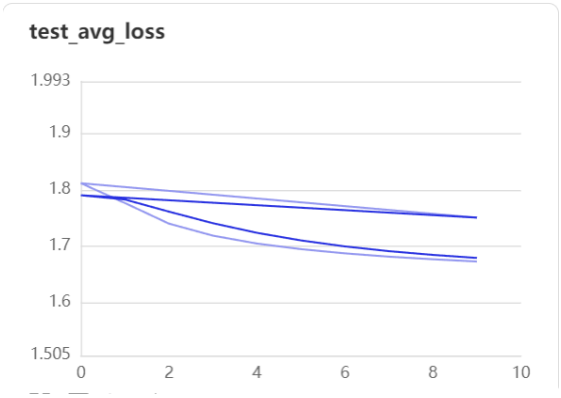
```python
class SOFTMAX_NET(paddle.nn.Layer):
    def __init__(self):
        super(SOFTMAX_NET, self).__init__()
        self.fc1 = Linear(in_features=784, out_features=10)

    def forward(self, inputs):
        inputs = paddle.reshape(inputs, [inputs.shape[0], 784])
        outputs1 = self.fc1(inputs)
        outputs_final = F.softmax(outputs1)
        return outputs_final
```

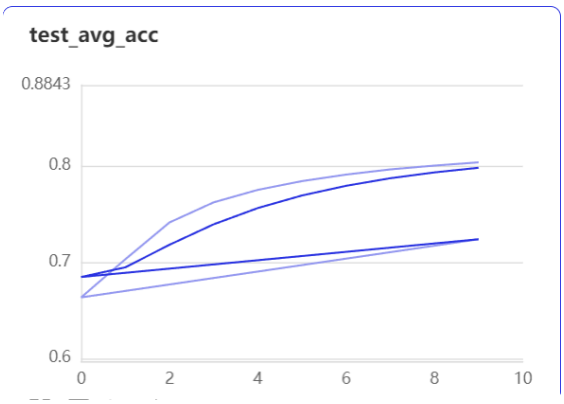
网络结构



训练集损失、测试集损失、测试集准确率







可以看到最后的准确率随着训练的增多而提高

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

```python
class MNIST(paddle.nn.Layer):
    def __init__(self):
        super(MNIST, self).__init__()
        # 还是两层隐藏层的网络
        self.fc1 = Linear(in_features=784, out_features=200)
        self.fc2 = Linear(in_features=200, out_features=200)
        # 输出十个维度，使用交叉熵损失函数
        self.fc3 = Linear(in_features=200, out_features=10)

    def forward(self, inputs):
        inputs = paddle.reshape(inputs, [inputs.shape[0], 784])
        outputs1 = self.fc1(inputs)
        outputs1 = F.relu(outputs1)
        outputs2 = self.fc2(outputs1)
        outputs2 = F.relu(outputs2)
        outputs3 = self.fc3(outputs2)
        # 使用softmax激活
        outputs_final = F.softmax(outputs3)
        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

全连接神经网络 Python

```python
# 全连接神经网络的搭建，三层
import numpy as np

class ThreeLayerNetwork(object):
    def __init__(self, input_num, hidden1_num, hidden2_num, output_num):
        # 使用Xavier初始化权重
        self.W1 = np.random.randn(input_num, hidden1_num) / np.sqrt(input_num)
        self.b1 = np.zeros((1, hidden1_num))
        self.W2 = np.random.randn(hidden1_num, hidden2_num) / np.sqrt(hidden1_num)
        self.b2 = np.zeros((1, hidden2_num))
        self.W3 = np.random.randn(hidden2_num, output_num) / np.sqrt(hidden2_num)
        self.b3 = np.zeros((1, output_num))

    def forward(self, X):
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = self.sigmoid(self.z1)
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = self.sigmoid(self.z2)
        self.z3 = np.dot(self.a2, self.W3) + self.b3
        return self.z3

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return self.sigmoid(x) * (1-self.sigmoid(x))

    def loss2(self, y_pred, y_true):
        error = y_pred - y_true
        num_sample = y_pred.shape[0]
        cost = np.sum(error ** 2)/ num_sample
        return cost

    def gradient(self, X, y_pred, y_true):
        m = y_pred.shape[0]
        delta3 = (y_pred - y_true)
        dW3 = np.dot(self.a2.T, delta3) / m
        db3 = np.sum(delta3, axis=0) / m
        delta2 = np.dot(delta3, self.W3.T) * self.sigmoid_derivative(self.z2)
        dW2 = np.dot(self.a1.T, delta2) / m
        db2 = np.sum(delta2,axis=0) / m
```

```
        delta1 = np.dot(delta2, self.W2.T) * self.sigmoid_derivative(self.z1)
        dW1 = np.dot(X.T, delta1) / m
        db1 = np.sum(delta1, axis=0) / m
        return dW1, db1, dW2, db2, dW3, db3

    def update(self, dW1, db1, dW2, db2, dW3, db3, eta):
        self.W1 -= eta * dW1
        self.b1 -= eta * db1
        self.W2 -= eta * dW2
        self.b2 -= eta * db2
        self.W3 -= eta * dW3
        self.b3 -= eta * db3

    def train(self, num_epochs, batch_size=10, eta = 0.01):
        losses = []
        train_loader = load_data('train', batch_size=batch_size)
        for epoch_id in range(num_epochs):
            for batch_id, data in enumerate(train_loader()):
                images, labels = data
                labels = labels.reshape(batch_size, 1)
                # 信号的前向传播
                predicts = self.forward(images)
                # 损失
                cost = self.loss2(predicts, labels)
                # 记录损失
                if batch_id % 200 == 0:
                    losses.append(cost)
                    print(f'epoch{epoch_id}, batch{batch_id}, loss{cost}')
                # 梯度的反向传播
                W1, b1, W2, b2, W3, b3 = self.gradient(images, predicts
                                                       ,labels)
                self.update(W1, b1, W2, b2, W3, b3, eta=eta)



        return losses

    def trainAndvalid(self, num_epochs, batch_size, eta):
        """
        进行超参数的搜索
        """
        train_losses = []
        valid_losses = []

        train_loader = load_data('train', batch_size=batch_size)
```

```
        valid_loader = load_data('valid',batch_size=batch_size)
        for epoch_id in range(num_epochs):
            for batch_id, data in enumerate(train_loader()):
                images, labels = data
                labels = labels.reshape(batch_size, 1)
                # 信号的前向传播
                predicts = self.forward(images)
                # 损失
                cost = self.loss2(predicts, labels)
                # 记录损失
                if batch_id % 200 == 0:
                    train_losses.append(cost)
                    print(f'epoch{epoch_id}, batch{batch_id}, train_loss{cost}')
                # 梯度的反向传播
                W1, b1, W2, b2, W3, b3 = self.gradient(images, predicts,labels)
                self.update(W1, b1, W2, b2, W3, b3, eta=eta)

            # 每个epoch结束后进行验证集损失
            valid_loss = 0
            for data in valid_loader():
                valid_imgs, valid_labels = data
                valid_labels = valid_labels.reshape(batch_size, 1)
                valid_predicts = self.forward(valid_imgs)
                valid_loss += self.loss2(valid_predicts, valid_labels)
            valid_losses.append(valid_loss)
            print(f'epoch{epoch_id}, valid_loss{valid_loss}')

        return train_losses, valid_losses


    def predict(self, X):
        z1 = np.dot(X, self.W1) + self.b1
        a1 = self.sigmoid(z1)
        z2 = np.dot(a1, self.W2) + self.b2
        a2 = self.sigmoid(z2)
        z3 = np.dot(a2, self.W3) + self.b3
        return z3
```

训练集要优化的超参数：

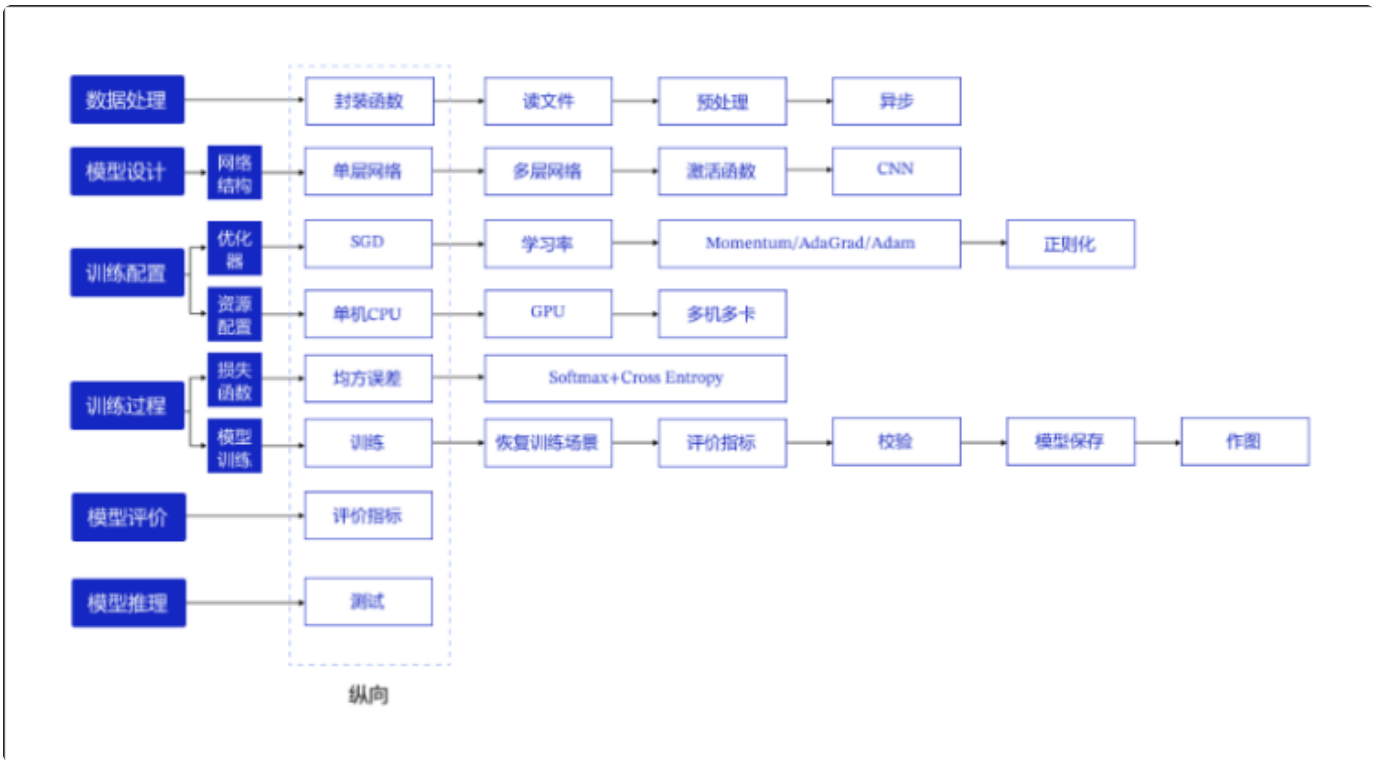
- 学习率：设置0-1，每次相隔0.1
- 节点个数：50-700，每次隔50
- epoch: 1-20,相隔1

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

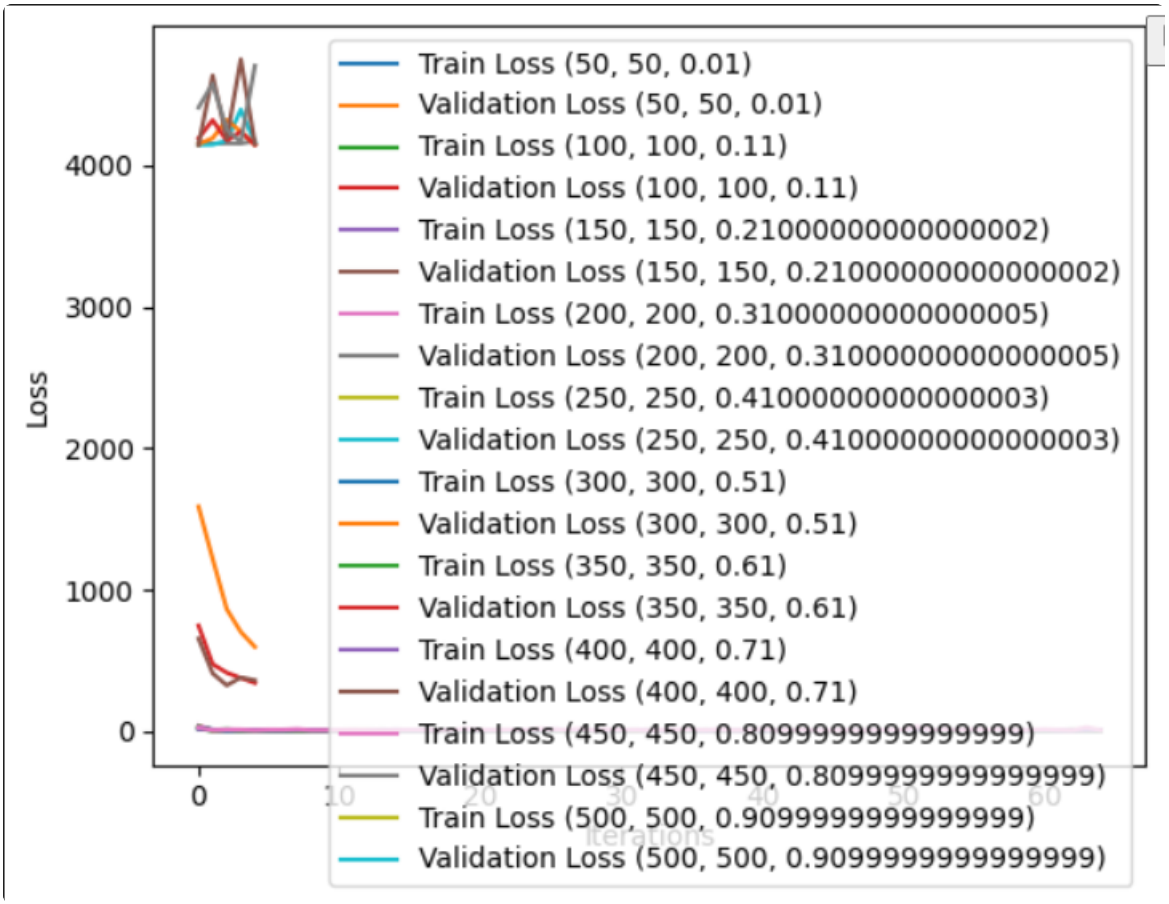
trainAndvalid Python

```python
    def trainAndvalid(self, num_epochs, batch_size, eta):
        """
        进行超参数的搜索
        """
        train_losses = []
        valid_losses = []

        train_loader = load_data('train', batch_size=batch_size)
        valid_loader = load_data('valid',batch_size=batch_size)
        for epoch_id in range(num_epochs):
            for batch_id, data in enumerate(train_loader()):
                images, labels = data
                labels = labels.reshape(batch_size, 1)
                # 信号的前向传播
                predicts = self.forward(images)
                # 损失
                cost = self.loss2(predicts, labels)
                # 记录损失
                if batch_id % 200 == 0:
                    train_losses.append(cost)
                    print(f'epoch{epoch_id}, batch{batch_id}, train_loss{cost}')
                # 梯度的反向传播
                W1, b1, W2, b2, W3, b3 = self.gradient(images, predicts
                                                       ,labels)
                self.update(W1, b1, W2, b2, W3, b3, eta=eta)

            # 每个epoch结束后进行验证集损失
            valid_loss = 0
            for data in valid_loader():
                valid_imgs, valid_labels = data
                valid_labels = valid_labels.reshape(batch_size, 1)
                valid_predicts = self.forward(valid_imgs)
                valid_loss += self.loss2(valid_predicts, valid_labels)
            valid_losses.append(valid_loss)
            print(f'epoch{epoch_id}, valid_loss{valid_loss}')

        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

训练优化超参数 Python

```python
# 增加了超参数的优化
hidden1_num = 50
hidden2_num = 50
lr = 0.01
import matplotlib.pyplot as plt

for i in range(10):
    # 创建网络
    net = ThreeLayerNetwork(input_num=784, hidden1_num=hidden1_num, hidden2_num=hidden2_num, output_num=1)

    # 启动训练
    train_losses, valid_losses = net.trainAndvalid(num_epochs=5, batch_size=20, eta=lr)

    # 绘制训练损失
    plot_x_train = np.arange(len(train_losses))
    plot_y_train = np.array(train_losses)
    plt.plot(plot_x_train, plot_y_train, label=f'Train Loss ({hidden1_num}, {hidden2_num}, {lr})')

    # 绘制验证损失
    plot_x_valid = np.arange(len(valid_losses))
    plot_y_valid = np.array(valid_losses)
    plt.plot(plot_x_valid, plot_y_valid, label=f'Validation Loss ({hidden1_num}, {hidden2_num}, {lr})')

    hidden1_num += 50
    hidden2_num += 50
    lr += 0.1
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
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

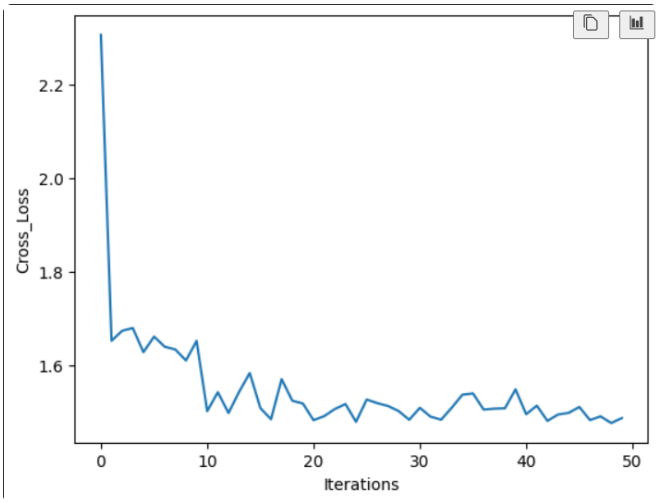
核心代码 Python

```python
        # 使用softmax激活
        outputs_final = F.softmax(outputs3)
        return outputs_final

        # 采用交叉熵损失函数
        loss = F.cross_entropy(predicts, labels)
        avg_loss = paddle.mean(loss)

```

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

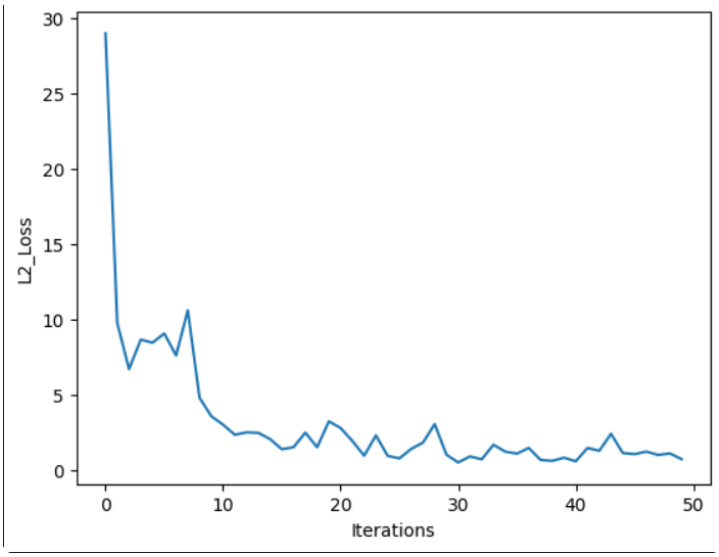
核心代码 Python

```python
        # 输出一个维度，使用均方差作为损失
        self.fc3 = Linear(in_features=200, out_features=1)

        #计算损失，取一个批次样本损失的平均值
        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

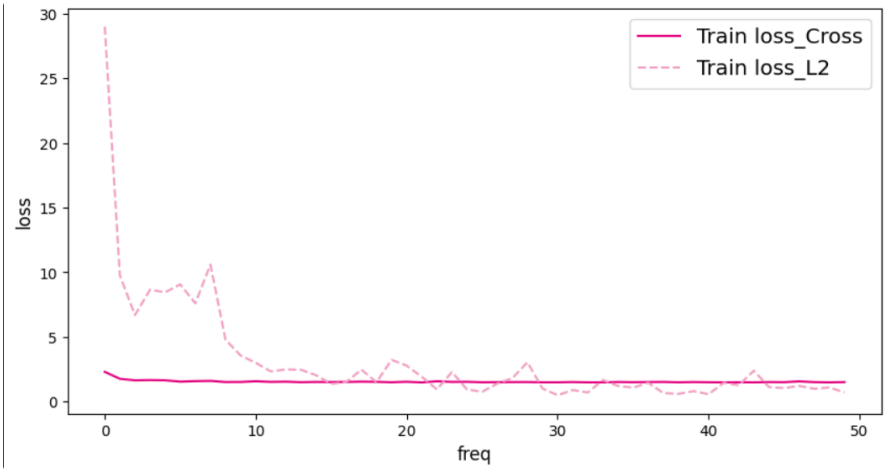
```python
acc_of_L2 = evaluation(model_L2, test_loader)
print(f'acc_of_L2:{acc_of_L2}')
```

[33] ✓ 0.9s Python

```
acc_of_L2:0.09783041477203369
```

2.6.2 比较

训练集损损失



- 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

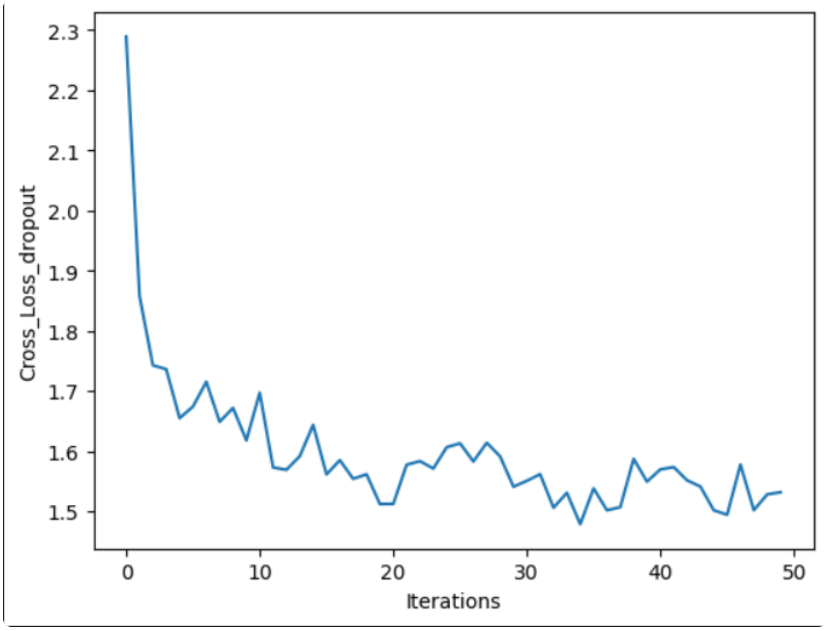
核心代码 Python

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

```

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



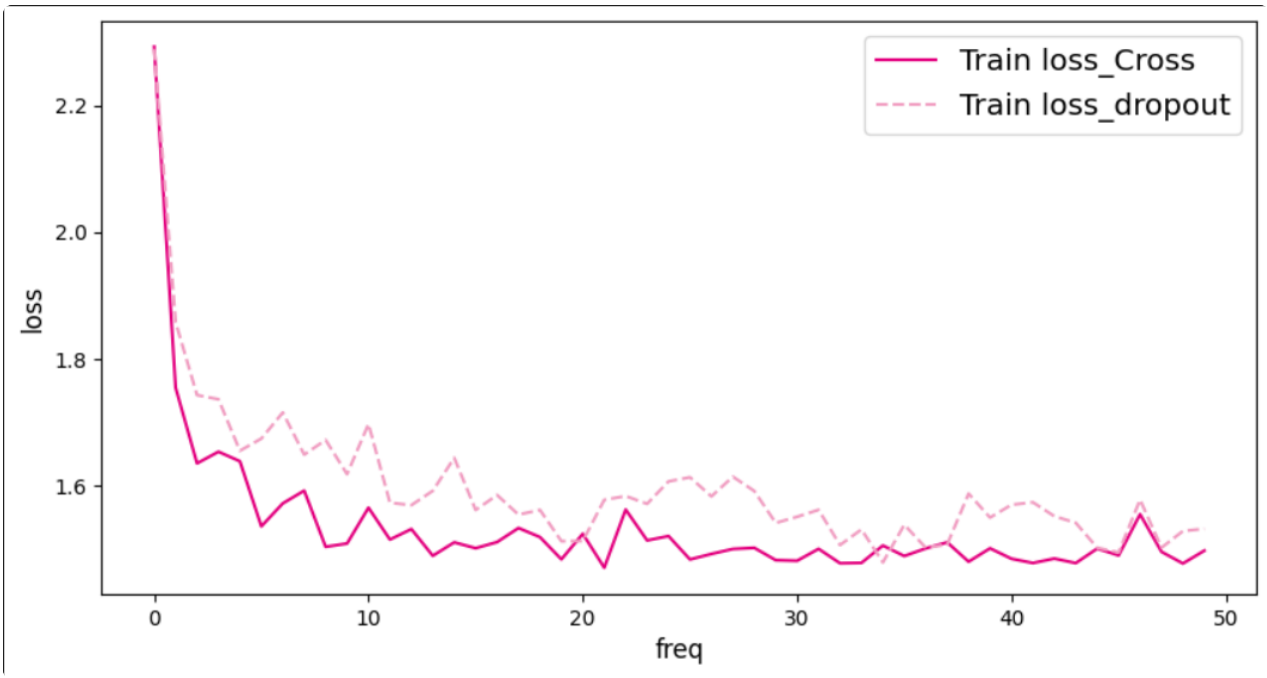
测试集准确率

```
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。

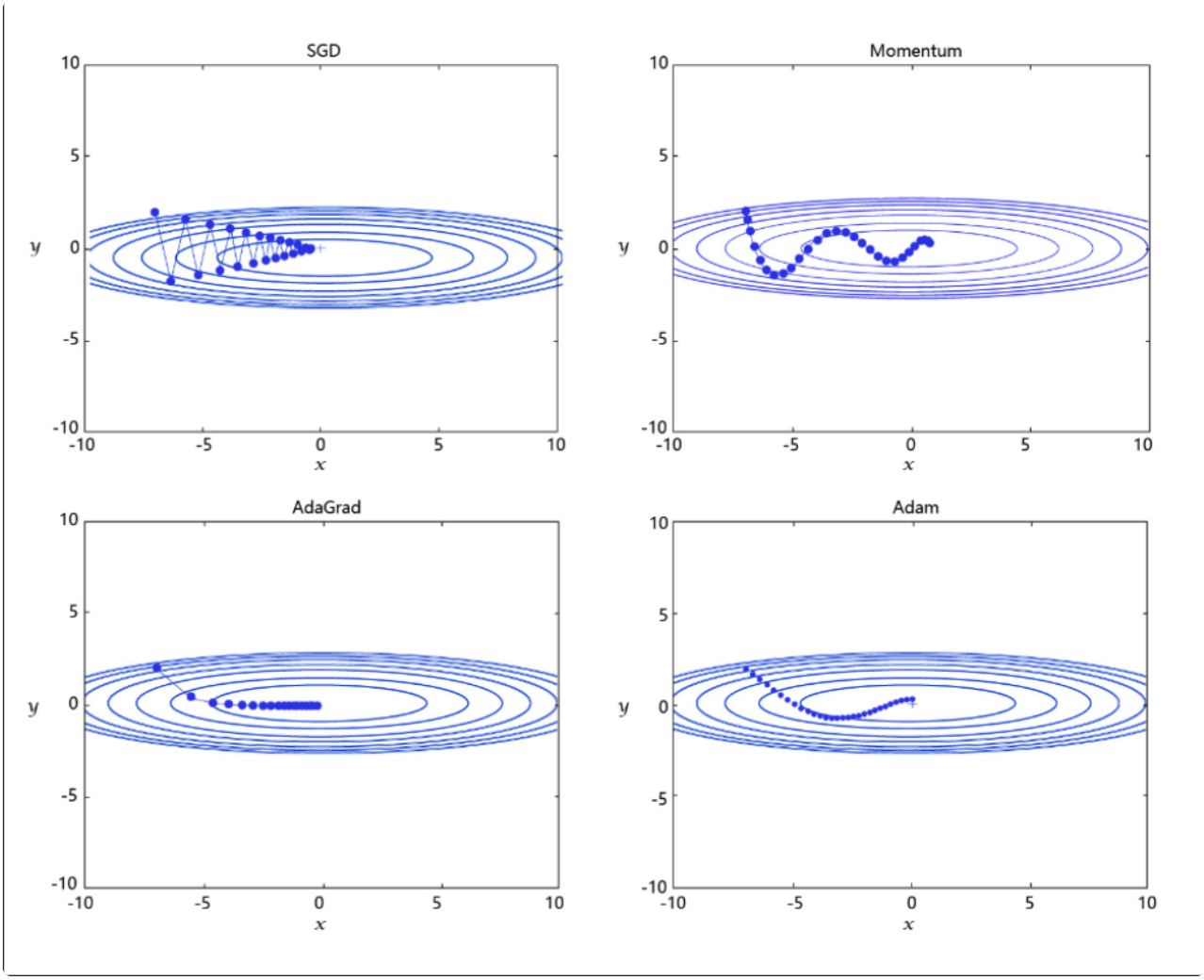
增加dropout后，有以下优点：

- 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

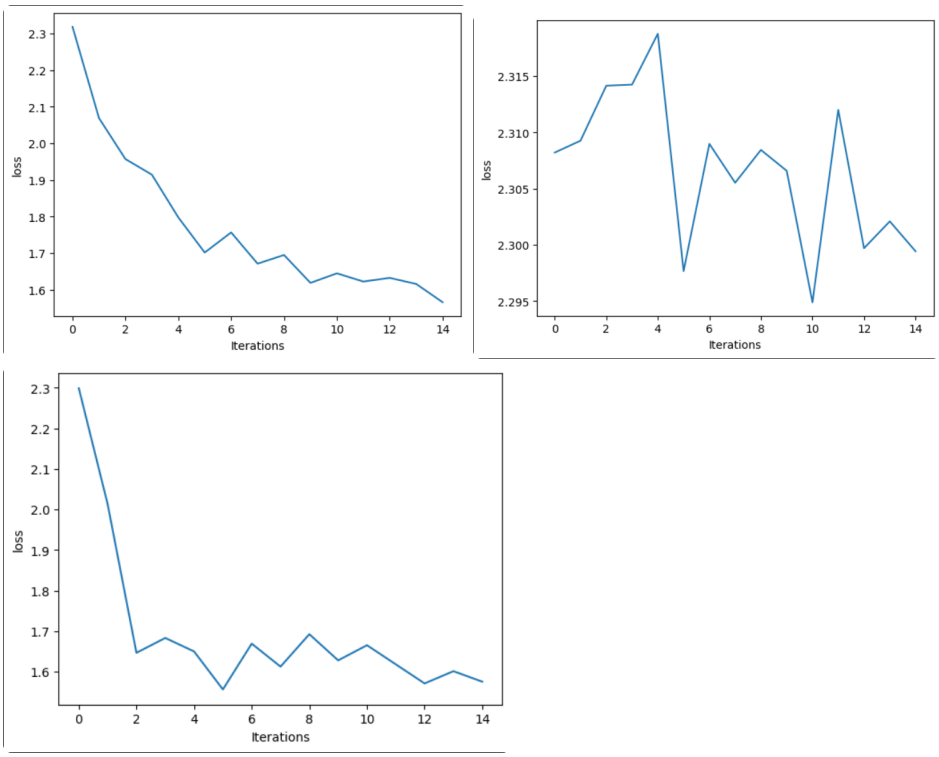
```python
    opt = paddle.optimizer.SGD(learning_rate=0.01, parameters=model.parameters())
    # opt = paddle.optimizer.Momentum(learning_rate=0.01, momentum=0.9, parameters=model.parameters())
    # opt = paddle.optimizer.Adagrad(learning_rate=0.01, parameters=model.parameters())
    # opt = paddle.optimizer.Adam(learning_rate=0.001, parameters=model.parameters())
```

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

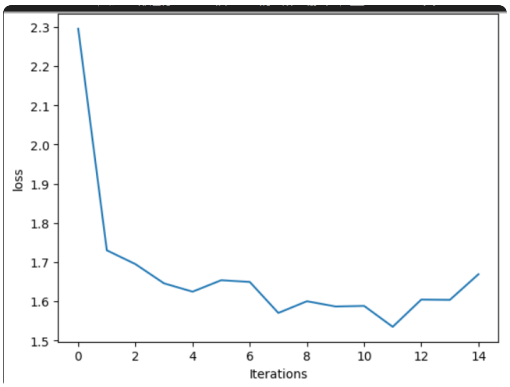
SGD



学习率分别对应 `0.01` `0.0001` `0.9`

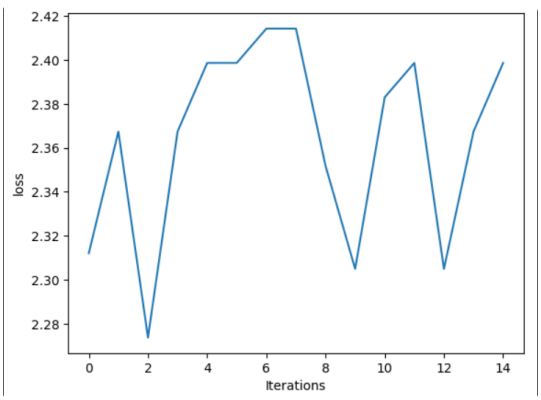
- 学习率太小：会收敛速度慢，如图二
- 学习率太大：会震荡，如图三

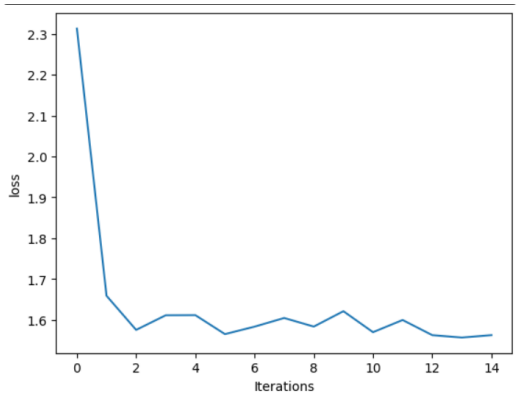
Momentum动量梯度下降法



- 由于累加梯度信息，可以冲出鞍点。

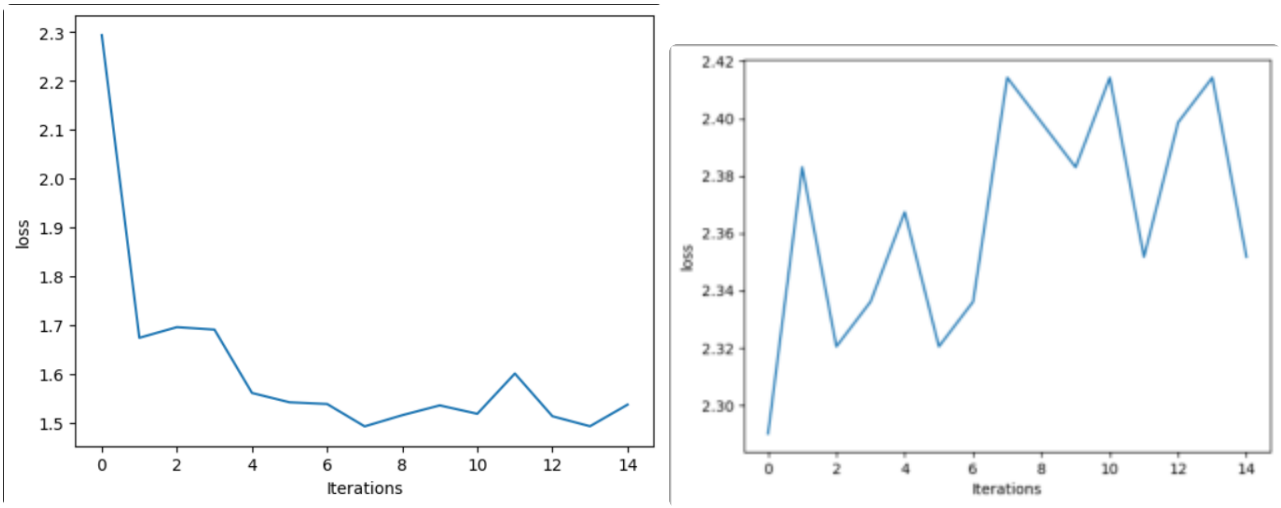
Adagrad





- 左图对应的学习率为 `0.01` ，可以看到loss没有下降
- 右图对应学习率为 `0.001` ，可以看到效果很好，说明Adagrad的步长走的更大，他的更新效果更好。

Adam



- 左图为 `Adam` 优化器学习率为0.001，右图为 `Adam` 学习率为0.01
- 和Adagrad是一样的问题，同时也说明它的更新效果好，可以用更好的学习率