机器学习基础 实验四 实验报告

学号：202122202214

姓名：马贵亮

班级：2021级软件5班

心得体会

实验目的

掌握 BP 神经网络的基本原理和基本的设计步骤 。

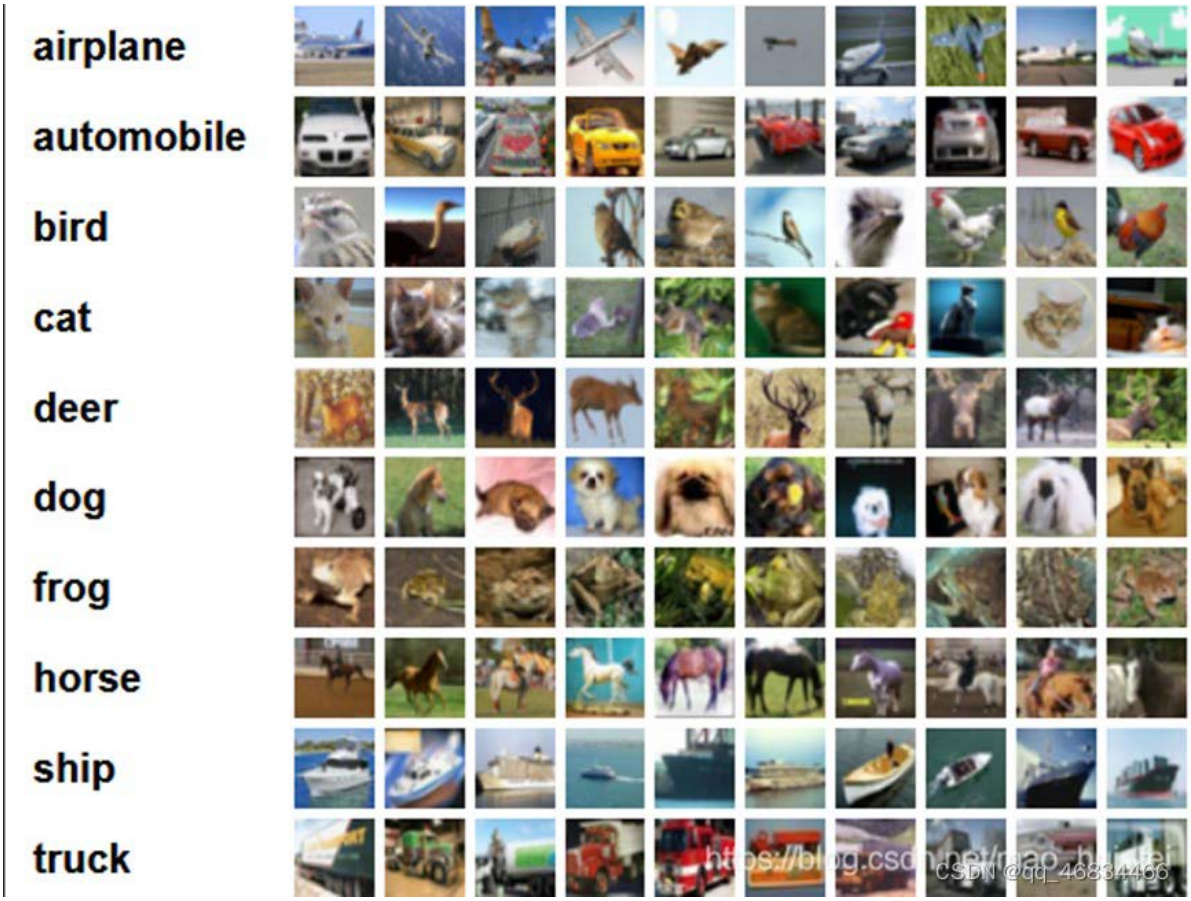
了解 BP 算法中各参数的作用和意义 。

实验内容

实验数据集

CIFAR-10数据集，数据集中包含 50000 张训练样本，10000 张测试样本，可将训练样本划分为49000 张样本的训练集和1000 张样本的验证集，测试集可只取1000 张测试样本。其中每个样本都是32×32 像素的RGB 彩色图片，具有三个通道，每个像素点包括RGB三个数值，数值范围0 ~ 255，所有照片分属10个不同的类别：飞机（airplane）、汽车（automobile）、鸟类（bird）、猫（cat）、鹿（deer）、狗（dog）、蛙类（frog）、马（horse）、船（ship）和卡车（truck）。

数据集展示如下图所示

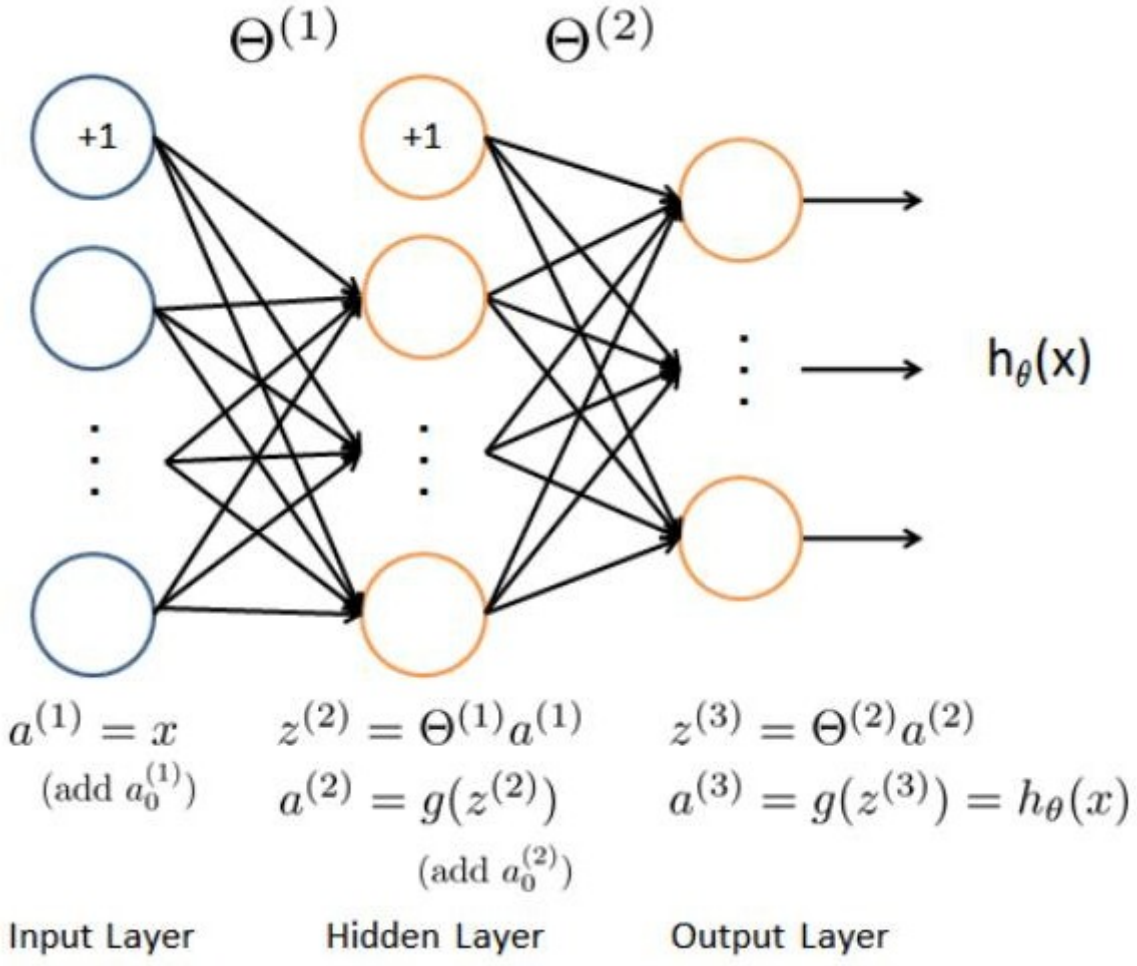


实验要求

用神经网络对给定的数据集进行分类，画出loss图，给出在测试集上的精确度；

不能使用 pytorch 等框架，也不能使用库函数，所有算法都要自己实现；

神经网络结构如下：



整个神经网络包括 3 层——输入层，隐藏层，输出层。输入层有 32*32*3个神经元，隐藏层有1024个神经元，输出层有 10 个神经元（对应 10 个类别）。 训练10个epoch。

注意事项：三层网络模型较为简单，模型准确率不需要很高，保证正确实现神经网络的搭建和训练即可。

其他提示：

1. 建议使用批处理和矩阵运算代替for循环，可以提高效率。
2. RGB图像的维度是：3（通道数）×32（长）×32（宽），可以根据自己的需求选择平均通道值还是最大最小通道值。
3. 输出层需要加入激活函数

思维发散

可以试着添加卷积层，修改隐藏层神经元数，层数，学习率，正则化权重等参数，探究参数对实验结果的影响。（尝试使用pytorch或tensorflow，将结果对比截图放入实验报告）

实验过程

1. 数据载入与处理

根据所给出的cifar-10-batches-py的数据集，数据集中包含 50000 张训练样本，10000 张测试样本，可将训练样本划分为49000 张样本的训练集和1000 张样本的验证集，再测试剩余的10000张测试样本。

通过阅读数据集中给出的readme.html文件，可以采用相同的方式将所有的数据集导入，获取其对应的样本数据和标签数据。

```python
def load_cifar10_batch(file_path):
    """ Load a single batch of CIFAR-10 data. """
    with open(file_path, 'rb') as file:
        datadict = pickle.load(file, encoding="latin1")  # 读取全部内容
        data = np.reshape(datadict['data'], (10000, 3072))
        labels = np.array(datadict['labels'])
        return data, labels


def load_all_batches(data_dir, batches):
    """ Load all CIFAR-10 data batches. """
    all_images = []
    all_labels = []
    for batch_name in batches:
        file_path = os.path.join(data_dir, batch_name)
        images, labels = load_cifar10_batch(file_path)
        all_images.append(images)
        all_labels.append(labels)
    return np.concatenate(all_images), np.concatenate(all_labels)

data_dir = '../data/cifar-10-batches-py'
batch_files = ['data_batch_1', 'data_batch_2', 'data_batch_3',
'data_batch_4', 'data_batch_5']
images, labels = load_all_batches(data_dir, batch_files)

test_files = ['test_batch']
test_images, test_labels = load_all_batches(data_dir, test_files)
```

通过readme.html中的文件给出的方法即可轻松读取该数据集的所有内容，这样就有了`train_images`、`train_labels`、`test_images`、`test_labels`两组互相对应的文件。

读取后每一个图片都被展平为一个3072为的向量，整体的训练数据集为一个50000*3072的矩阵，其中每一个特征对应其对应颜色通道上一点的像素值。由于rgb三色通道的像素值取值为0-255，因此我们需要对图片数据进行归一化处理，将其归一化到0-1的范围，便于后续神经网的计算和收束。

```python
images = images.astype(np.float32) / 255.0
test_images = test_images.astype(np.float32) / 255.0
```

而每一个`label`都是一个0-9的数字，表示对应的类别，显然这样是很难用于神经网络学习的，因为我们本质是有10种标签而不是一个连续的十个数值，因此需要对`label`进行独热编码（one-hot encoding）。

那么对于每一个`label`按照其具体数值，将1填充到对应的位置。例如一个标签为5的`label`，转换后成为下述表格的形式。

```python
def to_one_hot(labels, num_classes):
    one_hot_labels = np.zeros((labels.size, num_classes))
    one_hot_labels[np.arange(labels.size), labels] = 1
    return one_hot_labels
```

class0	class1	class2	class3	class4	class5	class6	class7	class8	class9
0	0	0	0	0	1	0	0	0	0

当然，此处对于训练数据的图片数据的处理还有很多，比如可以采用主成分分析法(PCA)方法来对3072维向量进行降维，又或者是在PCA过程种对数据进行白化，减少数据之间的相关性，但是由于不采用包来实现这两种数据处理的方法比较繁琐，因此在此不进行处理。

2. BP神经网络设计

BP神经网络，即反向传播（Backpropagation）神经网络，是一种多层前馈神经网络，其训练算法是通过误差反向传播来调整网络权重，以最小化输出误差。

2.1 符号定义

$\mathbf{X}$: 输入矩阵，维度为$n \times t$ 。n 为数据个数，t 为特征向量维数。

$\mathbf{y}^m$: 第m层神经元的输出矩阵。维度为$1 \times size(m)$

y_i^m: 第m层第i个神经元的输出值。

$\mathbf{w}^m$: 第$m-1$ 层神经元到第m层神经元的权重。维度为 $size(m-1) \times size(m)$

w_{ji}^m: 第$m-1$ 层第j个神经元到第m层第i个神经元的权重

E^m: 第m层神经元的线性组合。$1 \times size(m)$

ϵ_i^m: 第m层第i个神经元的线性组合。即$\epsilon_i^m = \sum_{j=1}^{size(m-1)} \omega_{ji}^m y_j^{m-1} + b_i^m$

b^m: 第m层神经元的偏置。维度为$1 \times size(m)$

D: 神经元层数。标量

$\mathbf{l}_i$: 输出层第i个神经元所造成的。维度为$1 \times size(D)$

L: 总损失。标量

Act_m : 第m层神经元的激活函数

$Loss$: 损失函数

2.2 运算定义

$\mathbf{y}^m = Act_m(\mathbf{y}^{m-1} \cdot \mathbf{w}^m + b^m)$

$L = \sum_{i=1}^{size(D)} l_i = \sum_{i=1}^{size(D)} Loss(y_i^D)$

2.3 反向传播推理：

ω_{kj}^m: 第$m-1$ 层第k个神经元到第m层第j个神经元的权重

$$\frac{\partial L}{\partial w_{kj}^m} = \frac{\partial L}{\partial \epsilon_j^m} \frac{\partial \epsilon_j^m}{\partial w_{kj}^m} = \frac{\partial L}{\partial \epsilon_j^m} y_k^{m-1}$$

考虑第$m-1$ 层所有神经元到第m层第j个神经元的权重

$$\frac{\partial L}{\partial \mathbf{w}_{\cdot j}^m} = \frac{\partial L}{\partial \epsilon_j^m} \frac{\partial \epsilon_j^m}{\partial \mathbf{w}_{\cdot j}^m} = \frac{\partial L}{\partial \epsilon_j^m} (\mathbf{y}^{m-1})^T$$

考虑矩阵

$$d\mathbf{w}^m = \frac{\partial L}{\partial \mathbf{w}^m} = \frac{\partial L}{\partial \mathrm{E}^m} \frac{\partial \mathrm{E}}{\partial \mathbf{w}^m} = (\mathbf{y}^{m-1})^T \frac{\partial L}{\partial \mathrm{E}^m}$$

再来考虑 $\frac{\partial L}{\partial \mathrm{E^m}}$，先考虑其中任意一个值

$$\frac{\partial L}{\partial \epsilon_j^m} = \sum_{i=1}^{size(m+1)} \frac{\partial L}{\partial \epsilon_i^{m+1}} \frac{\partial \epsilon_i^{m+1}}{\partial y_j^m} \frac{\partial y_j^m}{\partial \epsilon_j^m}$$

$$= \frac{\partial y_j^m}{\partial \epsilon_j^m} \sum_{i=1}^{size(m+1)} \frac{\partial L}{\partial \epsilon_i^{m+1}} w_{ji}^{m+1}$$

$$= Act_m^{'}(\epsilon_j^m)(\frac{\partial L}{\partial \mathrm{E}^{m+1}} \cdot (\mathbf{w}_{j\cdot}^{m+1})^T)$$

考虑矩阵

$$\frac{\partial L}{\partial \mathrm{E}^m} = Act_m^{'}(\mathrm{E}^m) \odot [\frac{\partial L}{\partial \mathrm{E}^{m+1}} \cdot (\mathbf{w}^{m+1})^T]$$

当$m = D$时

$$\frac{\partial L}{\partial \epsilon_i^D} = \frac{\partial L}{\partial y_i^D} \frac{\partial y_i^D}{\partial \epsilon_i^D} = Loss^{'}(y_i^D) \cdot Act_D^{'}(\epsilon_i^D)$$

考虑矩阵

$$\frac{\partial L}{\partial \mathrm{E}^D} = Loss^{'}(\mathbf{y}^D) \odot Act_D^{'}(\mathrm{E}^D)$$

考虑b^m

$$db^m = \frac{\partial L}{\partial \mathrm{E}^m}$$

如果综合，即b^m为标量

$$db^m = \sum_{i=1}^{size(m)} \frac{\partial L}{\partial \epsilon_i^m}$$

2.4 正向传播

$$\mathbf{y}^0 = \mathbf{X}$$

$$\mathbf{y}^m = Act_m(\mathrm{E}^m) = Act_m(\mathbf{y}^{m-1} \cdot \mathbf{w}^m + b^m)$$

记录所有 E^m 和 $\mathbf{y}^m$

2.5 反向传播

step1：计算

$$\frac{\partial L}{\partial \mathrm{E}^D} = Loss^{'}(\mathbf{y}^D) \odot Act_D^{'}(\mathrm{E}^D)$$

step2：自最深层向前遍历依次计算 `for m from D to 1`

$$d\mathbf{w}^m = \frac{\partial L}{\partial \mathbf{w}^m} = \frac{\partial L}{\partial \mathrm{E}^m} \frac{\partial \mathrm{E}^m}{\partial \mathbf{w}^m} = (\mathbf{y}^{m-1})^T \frac{\partial L}{\partial \mathrm{E}^m}$$

$$db^m = \frac{\partial L}{\partial \mathrm{E}^m} 或 db^m = \sum_{i=1}^{size(m)} \frac{\partial L}{\partial \epsilon_i^m}$$

$$\mathbf{w}^m = \mathbf{w}^m - \alpha d\mathbf{w}^m //不考虑L1、L2回归$$

$$b^m = b^m - \alpha db^m$$

$$\frac{\partial L}{\partial \mathrm{E}^{m-1}} = Act_m^{'}(\mathrm{E}^{m-1}) \odot [\frac{\partial L}{\partial \mathrm{E}^m} \cdot (\mathbf{w}^m)^T]$$

2.6 实验神经网络设计

本次实验首先采用最简单的三层全连接bp神经网络策略，第一层3072个神经元，第二层1024个神经元，第三层为10个神经元。

在激活函数方面，设计隐藏层的激活函数为 Relu，输出层的激活函数为 softmax

在计算损失函数上采用交叉熵来进行计算，主要是可以减少一部分有关导数的计算

3. BP神经网络实现

依据上述所描述的反向传播神经网络，我们进行简单的代码复现如下：

```
class NeuralNetwork:
    def __init__(self, layers, activations, loss='mse'):
        if len(activations) != len(layers) - 1:
            raise ValueError("Number of activations must be equal to number of layers - 1")

        self.layers = layers
        self.activations_info = activations
        self.loss = loss
        self.weights = []
        self.biases = []
        self.activation_funcs = []
        self.activation_derivs = []
        self.d_weights = [None] * (len(self.layers) - 1)  # Initializing gradients of weights
        self.d_biases = [None] * (len(self.layers) - 1)  # Initializing gradients of biases
        self.loss_list = []

        for i in range(len(layers) - 1):
            input_dim = self.layers[i]
            output_dim = self.layers[i + 1]

            weight = np.random.randn(input_dim, output_dim) * 0.01

            self.weights.append(weight)
            self.biases.append(np.zeros((1, output_dim)))

            activation, activation_derivative = self._get_activation(activations[i])
            self.activation_funcs.append(activation)
            self.activation_derivs.append(activation_derivative)

        self.loss_func, self.loss_derivative = self._get_loss_function(loss, activations[-1])

    def _get_activation(self, activation_name):
        activations = {
            'sigmoid': (sigmoid, sigmoid_derivative),
            'relu': (relu, relu_derivative),
```

```
            'tanh': (tanh, tanh_derivative),
            'softmax': (softmax, lambda x: 1)  # Dummy derivative for softmax at output
        }
        return activations.get(activation_name, (None, None))

    def _get_loss_function(self, loss_name, activation_name):
        if activation_name == 'softmax':
            loss_functions = {
                'mse': (mse, mse_derivative),
                'cross_entropy': (cross_entropy, cross_entropy_der_softmax)
            }
        else:
            loss_functions = {
                'mse': (mse, mse_derivative),
                'cross_entropy': (cross_entropy, cross_entropy_derivative)
            }
        return loss_functions.get(loss_name, (None, None))

    def forward(self, x, training=True):
        self.activations = [x]
        self.linearcombination = [x]
        for w, b, activation_func in zip(self.weights, self.biases, self.activation_funcs):
            z = np.dot(self.activations[-1], w) + b
            a = activation_func(z)
            self.linearcombination.append(z)
            self.activations.append(a)
        return self.activations[-1]

    def backward(self, y_true):
        error = self.loss_derivative(y_true, self.activations[-1])
        for i in reversed(range(len(self.weights))):
            error *= self.activation_derivs[i](self.linearcombination[i + 1])
            self.d_weights[i] = np.dot(self.activations[i].T, error)
            self.d_biases[i] = np.sum(error, axis=0, keepdims=True)
            error = np.dot(error, self.weights[i].T)

    def update_weights(self, learning_rate):
        for i in range(len(self.weights)):
            self.weights[i] -= learning_rate * (self.d_weights[i])
            self.biases[i] -= learning_rate * self.d_biases[i]

    def calculate_accuracy(self, y_true, y_pred):
        # 将热编码的真实标签转换为类别索引
        y_true_labels = np.argmax(y_true, axis=1)
        # 计算准确率
        correct_predictions = np.sum(y_true_labels == y_pred)
        accuracy = 100 * correct_predictions / len(y_true_labels)
        return accuracy

    def train(self, X, y, epochs,initial_lr, task='Exp4', val_size=0.2):
        if task == 'Exp4':
            X_temp = X[:49000]
            y_temp = y[:49000]
            X_test = X[49000:]
            y_test = y[49000:]
```

```
        else:
            val_len = int((1 - val_size) * X.shape[0])
            X_temp = X[:val_len]
            y_temp = y[:val_len]
            X_test = X[val_len:]
            y_test = y[val_len:]

        for epoch in range(epochs):
            self.forward(X_temp)
            self.backward(y_temp)
            self.update_weights(learning_rate)
            current_loss = self.loss_func(y,
self.forward(X,training=False))
            self.loss_list.append(current_loss)
            print(f"Epoch {epoch + 1}, Loss: {current_loss}")
            y_pred = self.predict(X_test)

            current_accuracy = self.calculate_accuracy(y_test, y_pred)
            print(f"Epoch {epoch + 1}, Validation Accuracy:
{current_accuracy:.2f}%")

        y_pred = self.predict(X_test)
        test_accuracy = self.calculate_accuracy(y_test, y_pred)
        print(f"Validation Accuracy: {test_accuracy:.2f}%")

    def predict(self, x):
        # 进行前向传播得到预测结果
        predictions = self.forward(x,training=False)
        # print(predictions)
        return np.argmax(predictions, axis=1)
```

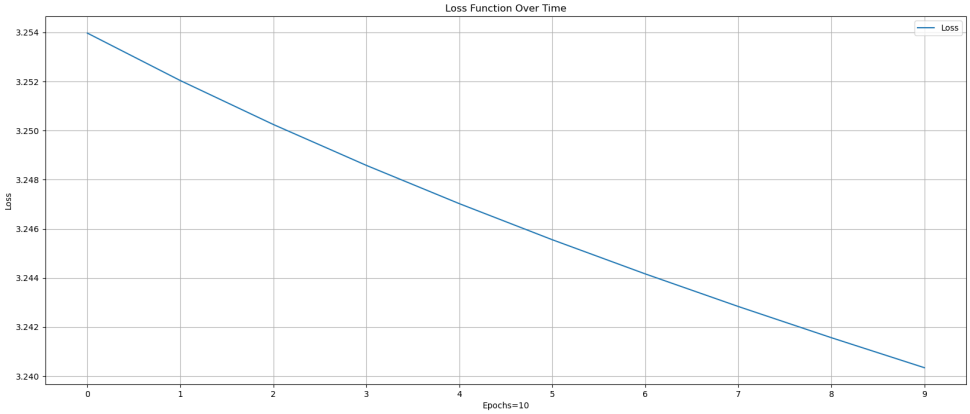
提前设计好所有的激活函数与激活函数的导数，并且设置好损失函数的计算方式。

```
def sigmoid(x):
    # print(x)
    x = np.clip(x, -20, 20)
    return 1 / (1 + np.exp(-x))


def sigmoid_derivative(x):
    sig = sigmoid(x)
    return sig * (1 - sig)


def relu(x):
    return np.maximum(0, x)


def relu_derivative(x):
    return (x > 0).astype(float)


def softmax(x):
    x_max = np.max(x, axis=1, keepdims=True)
    e_x = np.exp(x - x_max)
    sum_e_x = e_x.sum(axis=1, keepdims=True)
    return e_x / sum_e_x
```

```
25
26
27  def tanh(x):
28      return np.tanh(x)
29
30
31  def tanh_derivative(x):
32      return 1 - np.tanh(x) ** 2
33
34
35  def mse(y_true, y_pred):
36      return np.mean((y_true - y_pred) ** 2)
37
38
39  def mse_derivative(y_true, y_pred):
40      return 2 * (y_pred - y_true) / y_true.size
41
42
43  def cross_entropy(y_true, y_pred):
44      epsilon = 1e-12
45      y_pred = np.clip(y_pred, epsilon, 1. - epsilon)
46      return -np.sum(y_true * np.log(y_pred + 1e-9) + (1 - y_true + 1e-9) *
    np.log(1 - y_pred + 1e-9)) / y_true.shape[0]
47
48
49  def cross_entropy_derivative(y_true, y_pred):
50      epsilon = 1e-12
51      y_pred = np.clip(y_pred, epsilon, 1. - epsilon)
52      derivative = -y_true / y_pred + (1 - y_true) / (1 - y_pred)
53      return derivative / y_pred.shape[0]
54
55
56  def cross_entropy_der_softmax(y_true, y_pred):
57      epsilon = 1e-12
58      y_pred = np.clip(y_pred, epsilon, 1. - epsilon)
59      return (y_pred - y_true) / y_pred.shape[0]
```

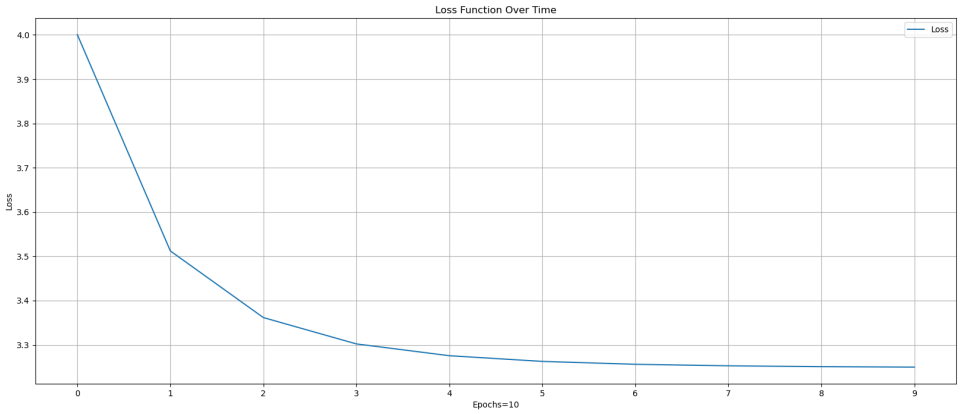
利用这样的方式，对原始的数据集进行预测，可以绘制其对应的损失变化图像和查看其准确率，其准确率是非常的低的，与自然情况下进行随机的区别不大，虽然我们的迭代次数仅有10次，但是这显示并不是非常的优秀，而且这还是采用了relu和softmax的情况下，运算速度还得到了提升。

```
exp4-hand ×
Epoch 8, Loss: 3.2428370248110268
Epoch 8, Validation Accuracy: 10.30%
Epoch 9, Loss: 3.2415666715255815
Epoch 9, Validation Accuracy: 9.90%
Epoch 10, Loss: 3.2403446802685894
Epoch 10, Validation Accuracy: 9.60%
Validation Accuracy: 9.60%
========================
Test Accuracy: 11.33%

进程已结束，退出代码为 0
```



在此设计一个各层之间均采用sigmoid函数的作为对比，相比于刚刚的设计方式，这种方式的准确率得到了提高，但是运算速度可以说是非常的缓慢：

```
exp4-hand ×
Epoch 9, Validation Accuracy: 16.00%
Epoch 10, Loss: 3.24981089525282
Epoch 10, Validation Accuracy: 15.90%
Validation Accuracy: 15.90%
========================
Test Accuracy: 15.26%

进程已结束，退出代码为 0
```



4. 小批次处理优化

小批次梯度下降法（Mini-Batch Gradient Descent）是一种在机器学习和深度学习训练中常用的优化算法，结合了批量梯度下降（Batch Gradient Descent）和随机梯度下降（Stochastic Gradient Descent）的优点。

4.1 梯度下降法简介

1. **批量梯度下降（Batch Gradient Descent, BGD）**：
    - 每次迭代使用整个训练数据集计算梯度并更新模型参数。
    - 优点：梯度计算精确，收敛稳定。
    - 缺点：当数据集很大时，计算梯度的过程非常耗时，且需要大量内存。
2. **随机梯度下降（Stochastic Gradient Descent, SGD）**：
    - 每次迭代仅使用一个训练样本计算梯度并更新模型参数。
    - 优点：更新频繁，计算快，内存需求小。
    - 缺点：梯度计算有噪声，收敛路径不稳定，可能导致震荡。

4.2 小批次梯度下降法

小批次梯度下降法结合了上述两者的优点，通过以下方式工作：

1. **小批次划分**：
    - 将训练数据集划分为多个小批次（mini-batches），每个小批次包含一定数量的样本（通常为几十到几百个样本）。
2. **梯度计算与更新**：
    - 每次迭代时，仅使用一个小批次的数据计算梯度并更新模型参数。
    - 通过这种方式，既减少了每次更新时的计算量，又降低了梯度的噪声。

4.3 具体步骤

1. **初始化模型参数。**
2. **划分数据集**：将训练数据集划分为多个小批次。
3. **迭代训练**：
    - 从数据集中依次取出一个小批次。
    - 使用该小批次的数据计算损失函数的梯度。
    - 根据梯度更新模型参数。
4. **重复步骤3，直至达到预设的迭代次数或损失函数收敛。**

4.4 实际应用

每次选取一个小批次，在实际过程中可以通过筛选下标来进行实现，但是我考虑到如果每轮每次我们选择的批次的内容均保持一致，那么在后续批次中该批次对应的若干样本的信息很有可能会被抵消，那么对于每次的预测过程中会更加收到后面批次的影响，因此为了让神经网络可以充分的学习特征，我在每轮将训练集再次进行打乱，这样每轮学习的批次内部信息不相同，以此来更好的增强学习能力。

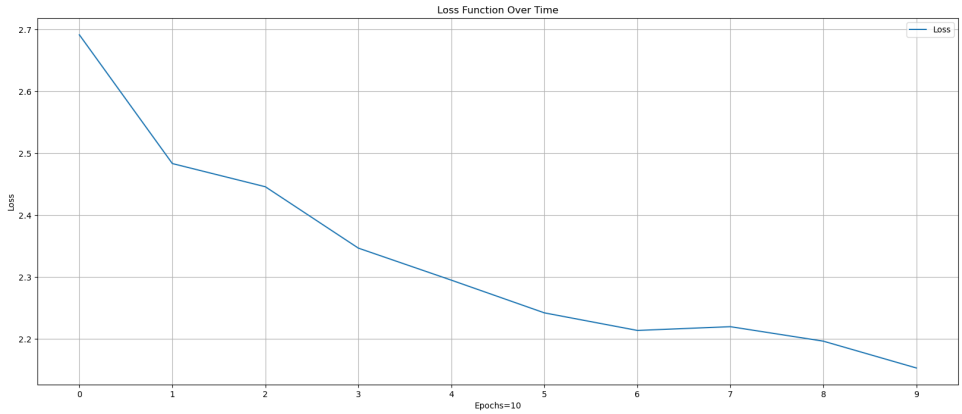
该步骤的优化仅仅只需要在`train`的训练过程种进行简单的修改即可，无需更改更多的代码，更新后的`train`函数如下：

```
def train(self, X, y, epochs, batch_size, initial_lr, task='Exp4', val_size=0.2):
    if task == 'Exp4':
        X_temp = X[:49000]
        y_temp = y[:49000]
        X_test = X[49000:]
        y_test = y[49000:]
    else:
        val_len = int((1 - val_size) * X.shape[0])
        X_temp = X[:val_len]
        y_temp = y[:val_len]
        X_test = X[val_len:]
        y_test = y[val_len:]

    for epoch in range(epochs):
        indices = np.arange(X_temp.shape[0])
        np.random.shuffle(indices)
        X_train = X_temp[indices]
        y_train = y_temp[indices]
        for i in range(0, X_train.shape[0], batch_size):
            X_batch = X_train[i:i + batch_size]
            y_batch = y_train[i:i + batch_size]
            self.forward(X_batch)
            self.backward(y_batch)
            self.update_weights(learning_rate)
            if ((i / batch_size) + 1) % 100 == 0:
                current_loss = self.loss_func(y_test, self.forward(X_test))
                print(f"Epoch {epoch + 1}, Batch {int((i / batch_size) + 1)}, Loss = {current_loss}")

        current_loss = self.loss_func(y, self.forward(X,training=False))
        self.loss_list.append(current_loss)
        print(f"Epoch {epoch + 1}, Loss: {current_loss}")

        y_pred = self.predict(X_test)
        current_accuracy = self.calculate_accuracy(y_test, y_pred)
        print(f"Epoch {epoch + 1}, Validation Accuracy: {current_accuracy:.2f}%")
    y_pred = self.predict(X_test)
    test_accuracy = self.calculate_accuracy(y_test, y_pred)
    print(f"Validation Accuracy: {test_accuracy:.2f}%")
```

进行小批次处理后我们继续采用原先的模型结构，先设置`batch_size = 64`，分别测试ReLU+softmax 的组合和 sigmoid+sigmoid的组合。

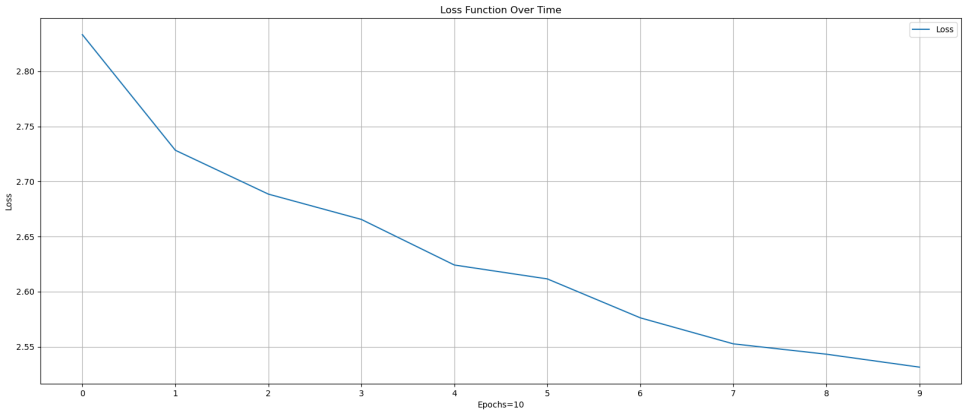
对于ReLU+softmax组合：效果比之前非常的显示，第一轮训练结束后准确率就可以达到35%左右，优化效果还是十分的显著

```
exp4-hand ×
Epoch 8, Batch 200, Loss = 2.3746954724520615
Epoch 8, Batch 300, Loss = 2.323954968816914
Epoch 8, Batch 400, Loss = 2.3254757435053897
Epoch 8, Batch 500, Loss = 2.324557554712854
Epoch 8, Batch 600, Loss = 2.3118430156285434
Epoch 8, Batch 700, Loss = 2.3006065020503703
Epoch 8, Loss: 2.2200650545273044
Epoch 8, Validation Accuracy: 45.70%
Epoch 9, Batch 100, Loss = 2.30185659950868
Epoch 9, Batch 200, Loss = 2.3026107259652426
Epoch 9, Batch 300, Loss = 2.3154700043001255
Epoch 9, Batch 400, Loss = 2.300162402529987
Epoch 9, Batch 500, Loss = 2.3031989373095008
Epoch 9, Batch 600, Loss = 2.3094061941280115
Epoch 9, Batch 700, Loss = 2.282135580586637
Epoch 9, Loss: 2.1966518514840367
Epoch 9, Validation Accuracy: 46.00%
Epoch 10, Batch 100, Loss = 2.3331997354428258
Epoch 10, Batch 200, Loss = 2.2914043998714835
Epoch 10, Batch 300, Loss = 2.2945547293895707
Epoch 10, Batch 400, Loss = 2.2958660887626676
Epoch 10, Batch 500, Loss = 2.288078371684721
Epoch 10, Batch 600, Loss = 2.3146412909572747
Epoch 10, Batch 700, Loss = 2.2590020931324384
Epoch 10, Loss: 2.153317839713151
Epoch 10, Validation Accuracy: 44.20%
Validation Accuracy: 44.20%
------------------------
Test Accuracy: 45.89%

进程已结束，退出代码为 0
```



对于sigmoid+sigmoid组合：

```
exp4-hand ×
Epoch 8, Batch 200, Loss = 2.592299349453407
Epoch 8, Batch 300, Loss = 2.584196626710591
Epoch 8, Batch 400, Loss = 2.587899463892876
Epoch 8, Batch 500, Loss = 2.6081591086362623
Epoch 8, Batch 600, Loss = 2.589497202818674
Epoch 8, Batch 700, Loss = 2.5813893341312246
Epoch 8, Loss: 2.5526214904784372
Epoch 8, Validation Accuracy: 41.40%
Epoch 9, Batch 100, Loss = 2.5846631607082116
Epoch 9, Batch 200, Loss = 2.5644849168595347
Epoch 9, Batch 300, Loss = 2.5781838766455525
Epoch 9, Batch 400, Loss = 2.566767969085119
Epoch 9, Batch 500, Loss = 2.5623893913072537
Epoch 9, Batch 600, Loss = 2.5830213587598903
Epoch 9, Batch 700, Loss = 2.57898364503429
Epoch 9, Loss: 2.5432244061825697
Epoch 9, Validation Accuracy: 40.20%
Epoch 10, Batch 100, Loss = 2.5707630371714805
Epoch 10, Batch 200, Loss = 2.550917838933001
Epoch 10, Batch 300, Loss = 2.5683785433871016
Epoch 10, Batch 400, Loss = 2.5700936649702686
Epoch 10, Batch 500, Loss = 2.5528220084695747
Epoch 10, Batch 600, Loss = 2.553282724703835
Epoch 10, Batch 700, Loss = 2.5572865455665683
Epoch 10, Loss: 2.5314867536290704
Epoch 10, Validation Accuracy: 39.80%
Validation Accuracy: 39.80%
------------------------
Test Accuracy: 41.34%

进程已结束，退出代码为 0
```



通过对比可以看出在引入小批次下降后，由于更细次数变得更多，ReLU+softmax 的组合的上升速度更快，整体效率会更好。而sigmoid函数在层层叠加之间可能出现在`exp`操作后越界的情况，因此以下所有的优化均在隐藏层为relu，输出层为softmax的基础上进行。

5. 正则化引入

正则化的目的是为了防止模型过拟合（overfitting），从而提高模型在新数据上的泛化能力（generalization）。具体来说，正则化通过在损失函数中引入额外的惩罚项来限制模型的复杂度，使得模型不仅在训练数据上表现良好，而且在测试数据上也能有较好的表现。

5.1 引入L2正则化

1. **定义正则化损失函数**：在原始损失函数基础上添加L2正则化项。即$\frac{1}{2}||\mathbf{w}||^2$
2. **计算梯度**：在反向传播过程中，除了计算标准的损失函数梯度，还需要计算正则化项对每个权重的梯度。
3. **更新权重**：在每次迭代中，按照梯度下降法更新权重时，考虑正则化项对梯度的影响。

5.2 具体实现

假设原始损失函数的梯度为$d\mathbf{w}^m = \frac{\partial L}{\partial \mathbf{w}^m}$，则加入L2正则化后的梯度为：

$$d\mathbf{w}^m = \frac{\partial L}{\partial \mathbf{w}^m} + l_2\mathbf{w}^m$$

因此，权重更新公式变为：

$$\mathbf{w} = \mathbf{w} - \eta d\mathbf{w}^m = \mathbf{w} - \eta(\frac{\partial L}{\partial \mathbf{w}^m} + l_2\mathbf{w}^m)$$

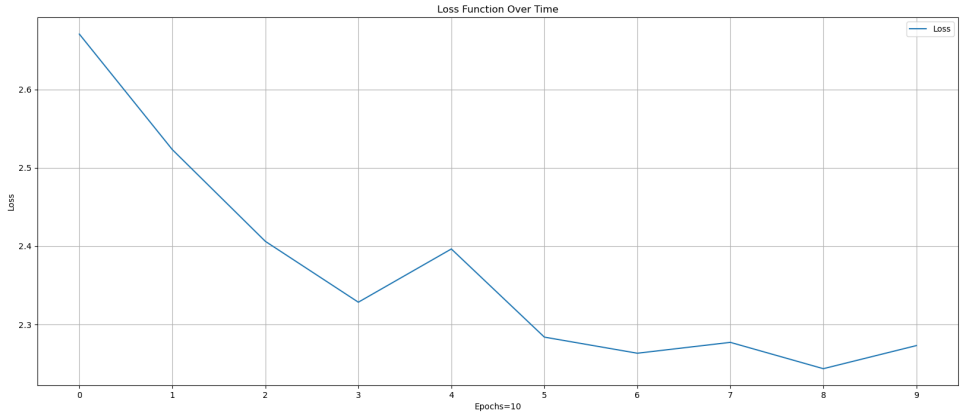
其中，η是学习率。

这一部分在代码种的实现相对容易，即在反向传播更新梯度向量时进行简单修改即可：

```
def backward(self, y_true):
    error = self.loss_derivative(y_true, self.activations[-1])
    for i in reversed(range(len(self.weights))):
        error *= self.activation_derivs[i](self.linearcombination[i + 1])
        self.d_weights[i] = np.dot(self.activations[i].T, error) +
self.l2_lambda * self.weights[i]
        self.d_biases[i] = np.sum(error, axis=0, keepdims=True)
        error = np.dot(error, self.weights[i].T)
```

5.3 实际应用

仍采用刚刚设计的基本模型，在此基础上再次运行，由于每次运行中存在随机数因素，因此保持准确率大致相同的情况下，则说明正则化略微起到了限制过拟合的作用。



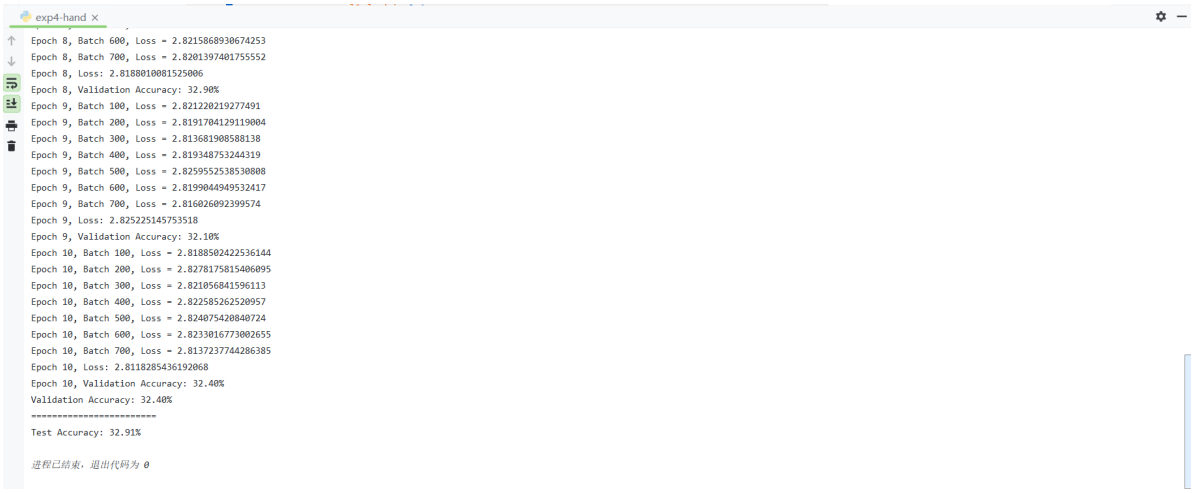
6. 尝试提高模型准确率

6.1 改变正则化权重

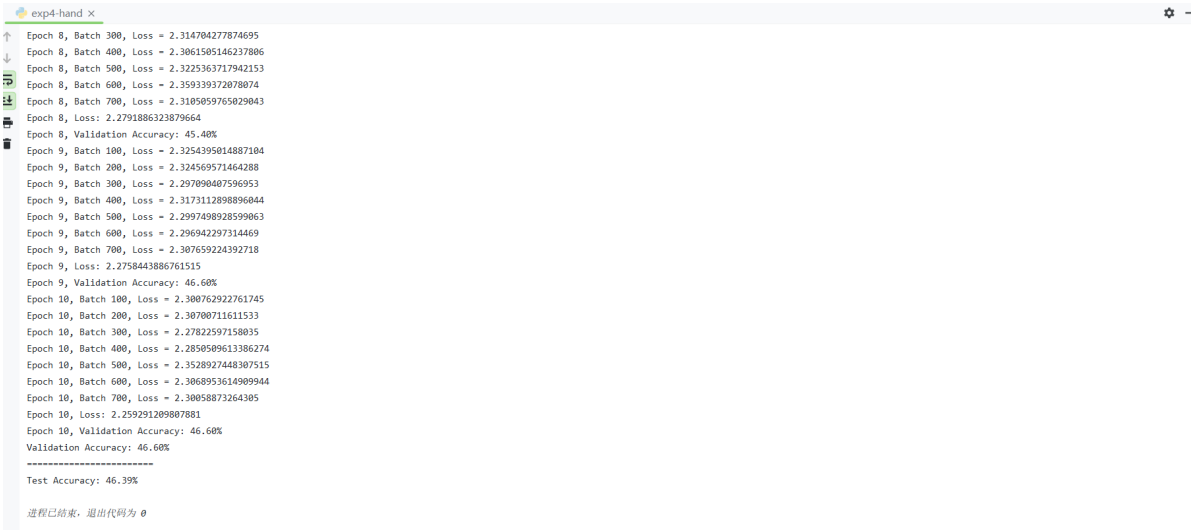
分别尝试不同的正则化系数 $l_2 = 0.1$ 和 $l_2 = 0.001$ 两种不同的正则化权重，在模型结构保持不变的情况下，即神经元个数为3072+1024+10，隐藏层激活函数采用ReLU、输出层激活函数采用softmax的基础上，`batch_size=64`。来探究和上述表示中$l_2 = 0.01$时的损失函数图像与准确率的变化。

下列为运行过程中准确率的截图：

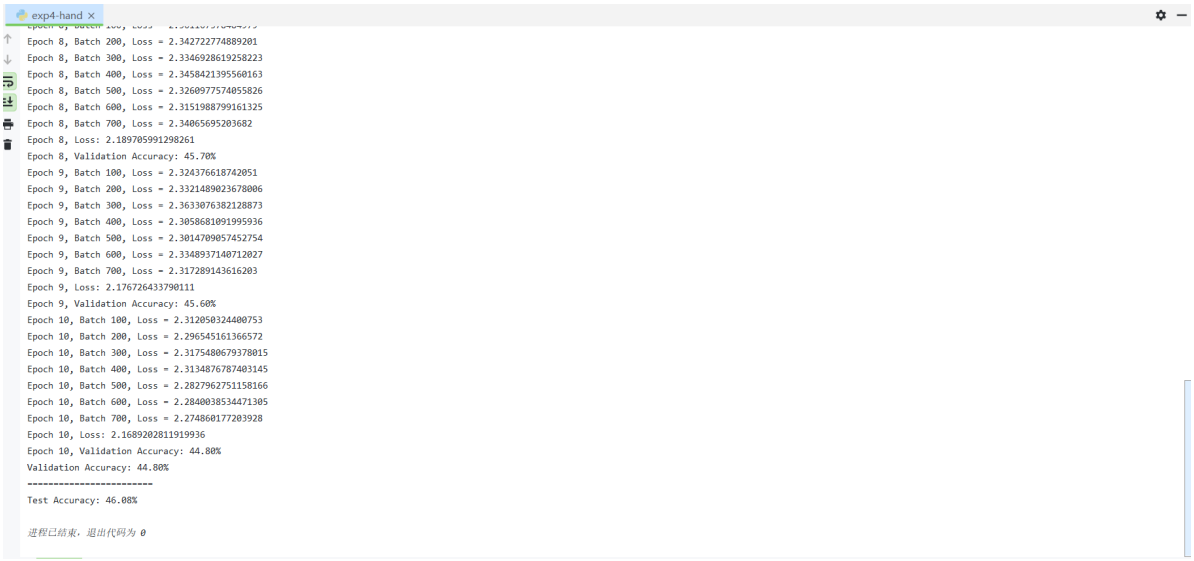
l_2 = 0.1时：



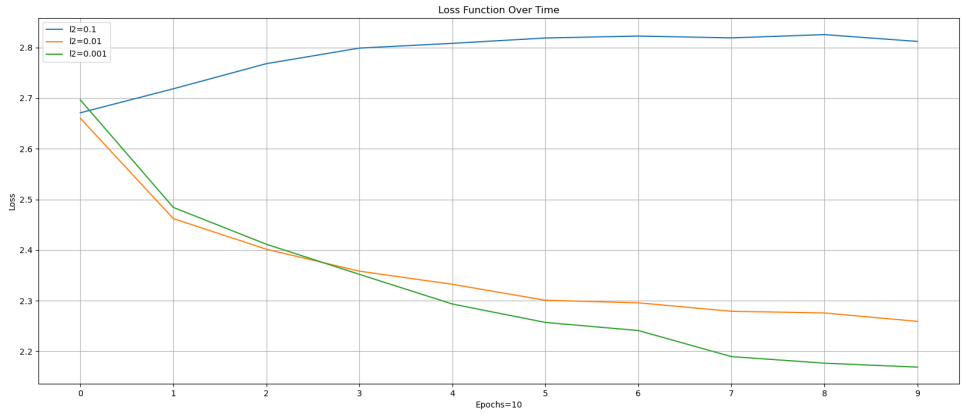
l_2 = 0.01时：



l_2 = 0.001时：



但从准确率来看，$l_2 = 0.01$ 和 $l_2 = 0.001$区别在准确率上区别不大，而$l_2 = 0.1$ 效果很不好，随后再综合考虑其对应的损失值变化情况。



可以发现当l_2系数较大时，不利于模型的收束，当系数较小时对模型的影响较小，更容易过拟合

6.2 增加迭代轮次

在保持原正则化系数$l_2 = 0.01$的基础上，将迭代次数修改为30次，`batch_size = 64`，可以发现在第十次左右时，验证集上的准确率可以达到48.10%，随后一直在收敛的过程，最终最佳准确率能够达到50%左右。

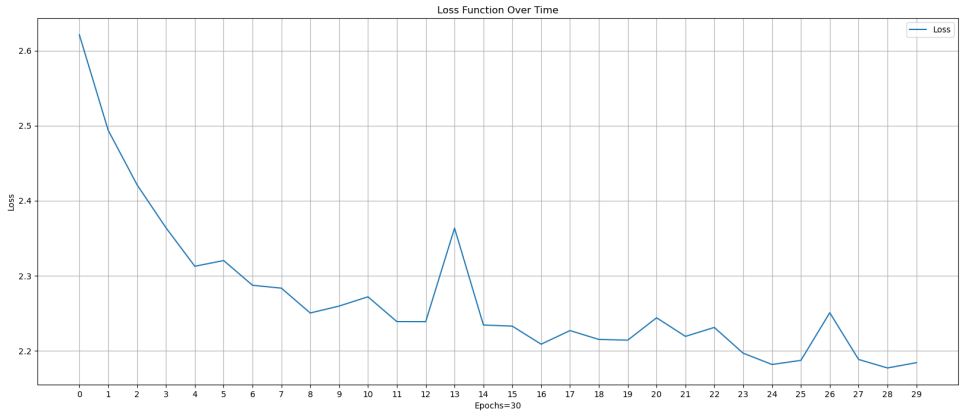
训练过程准确率变化：

```
Epoch 1, Validation Accuracy: 38.70%
Epoch 2, Validation Accuracy: 40.70%
Epoch 3, Validation Accuracy: 40.20%
Epoch 4, Validation Accuracy: 43.50%
Epoch 5, Validation Accuracy: 45.50%
Epoch 6, Validation Accuracy: 43.10%
Epoch 7, Validation Accuracy: 44.00%
Epoch 8, Validation Accuracy: 45.60%
Epoch 9, Validation Accuracy: 48.10%
Epoch 10, Validation Accuracy: 46.80%
Epoch 11, Validation Accuracy: 44.90%
Epoch 12, Validation Accuracy: 48.70%
Epoch 13, Validation Accuracy: 45.80%
Epoch 14, Validation Accuracy: 41.30%
Epoch 15, Validation Accuracy: 47.30%
Epoch 16, Validation Accuracy: 46.40%
Epoch 17, Validation Accuracy: 46.50%
Epoch 18, Validation Accuracy: 47.10%
Epoch 19, Validation Accuracy: 47.30%
Epoch 20, Validation Accuracy: 48.50%
Epoch 21, Validation Accuracy: 46.50%
Epoch 22, Validation Accuracy: 47.20%
Epoch 23, Validation Accuracy: 48.00%
Epoch 24, Validation Accuracy: 49.50%
Epoch 25, Validation Accuracy: 49.60%
Epoch 26, Validation Accuracy: 50.00%
Epoch 27, Validation Accuracy: 46.50%
Epoch 28, Validation Accuracy: 49.20%
Epoch 29, Validation Accuracy: 50.20%
Epoch 30, Validation Accuracy: 49.20%
========================
Test Accuracy: 49.29%
```

过程图：

```
exp4-hand ×
Epoch 28, Batch 200, Loss = 2.229295780499215
Epoch 28, Batch 300, Loss = 2.237525738112473
Epoch 28, Batch 400, Loss = 2.2192230664241572
Epoch 28, Batch 500, Loss = 2.2515837568233477
Epoch 28, Batch 600, Loss = 2.2263150657772353
Epoch 28, Batch 700, Loss = 2.275775041159745
Epoch 28, Loss: 2.188430628367071
Epoch 28, Validation Accuracy: 49.20%
Epoch 29, Batch 100, Loss = 2.236769785972297
Epoch 29, Batch 200, Loss = 2.2929143160638246
Epoch 29, Batch 300, Loss = 2.2538366629358406
Epoch 29, Batch 400, Loss = 2.2152546628163954
Epoch 29, Batch 500, Loss = 2.232171276420015
Epoch 29, Batch 600, Loss = 2.2286391746927454
Epoch 29, Batch 700, Loss = 2.228388140010144
Epoch 29, Loss: 2.177094001579585
Epoch 29, Validation Accuracy: 50.20%
Epoch 30, Batch 100, Loss = 2.2758508102932917
Epoch 30, Batch 200, Loss = 2.237874179516836
Epoch 30, Batch 300, Loss = 2.239214105892339
Epoch 30, Batch 400, Loss = 2.2114743495097633
Epoch 30, Batch 500, Loss = 2.214289558916813
Epoch 30, Batch 600, Loss = 2.2633253083412472
Epoch 30, Batch 700, Loss = 2.270123389403212 7
Epoch 30, Loss: 2.1841727735193603
Epoch 30, Validation Accuracy: 49.20%
Validation Accuracy: 49.20%
========================
Test Accuracy: 49.29%

进程已结束，退出代码为 0
```

损失函数变化图：



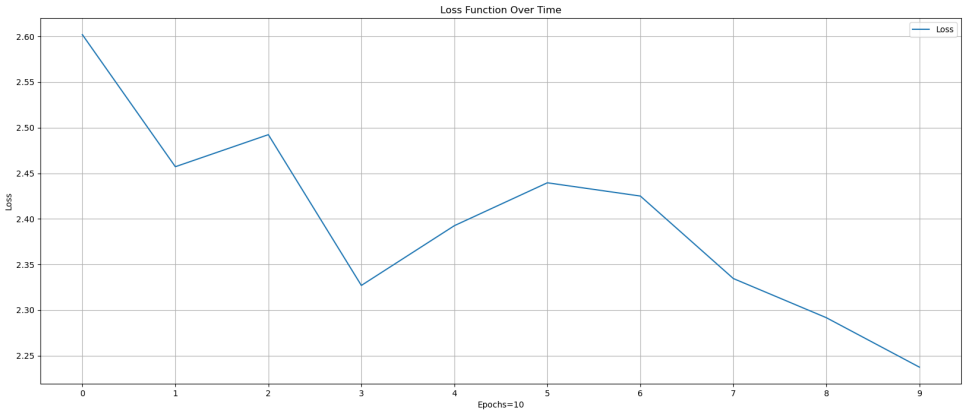
6.3 修改批次大小

在保持原正则化系数$l_2 = 0.01$的基础上，将迭代次数为10次，设置：`batch_size = 32`。也就是说梯度的更新次数比原先多了一倍，来观察其对应的准确率和损失函数变化。

运行过程准确率展示：

```
exp4-hand ×
Epoch 9, Batch 1000, Loss = 2.3558862783571164
Epoch 9, Batch 1100, Loss = 2.2900213869945922
Epoch 9, Batch 1200, Loss = 2.306081468346112
Epoch 9, Batch 1300, Loss = 2.2794110539839085
Epoch 9, Batch 1400, Loss = 2.2862970213803195
Epoch 9, Batch 1500, Loss = 2.306690817560279
Epoch 9, Loss: 2.291598071801388
Epoch 9, Validation Accuracy: 47.00%
Epoch 10, Batch 100, Loss = 2.2815549579625367
Epoch 10, Batch 200, Loss = 2.300328338693511 4
Epoch 10, Batch 300, Loss = 2.302926853100649
Epoch 10, Batch 400, Loss = 2.283016942997393
Epoch 10, Batch 500, Loss = 2.3416750412769924
Epoch 10, Batch 600, Loss = 2.29572690874695
Epoch 10, Batch 700, Loss = 2.299339387673227
Epoch 10, Batch 800, Loss = 2.2952071394250417
Epoch 10, Batch 900, Loss = 2.305986046914826
Epoch 10, Batch 1000, Loss = 2.3209772342465804
Epoch 10, Batch 1100, Loss = 2.293210326083473
Epoch 10, Batch 1200, Loss = 2.3058035179242258
Epoch 10, Batch 1300, Loss = 2.2775356210226 93
Epoch 10, Batch 1400, Loss = 2.2890312884315716
Epoch 10, Batch 1500, Loss = 2.3240606229759266
Epoch 10, Loss: 2.237388518814722
Epoch 10, Validation Accuracy: 48.70%
Validation Accuracy: 48.70%
========================
Test Accuracy: 47.92%

进程已结束，退出代码为 0
```

其损失函数的变化：



发现损失函数的图像似乎并不是那么平滑，猜测可能是由于内部更新次数太多导致局部出现了过拟合的情况。

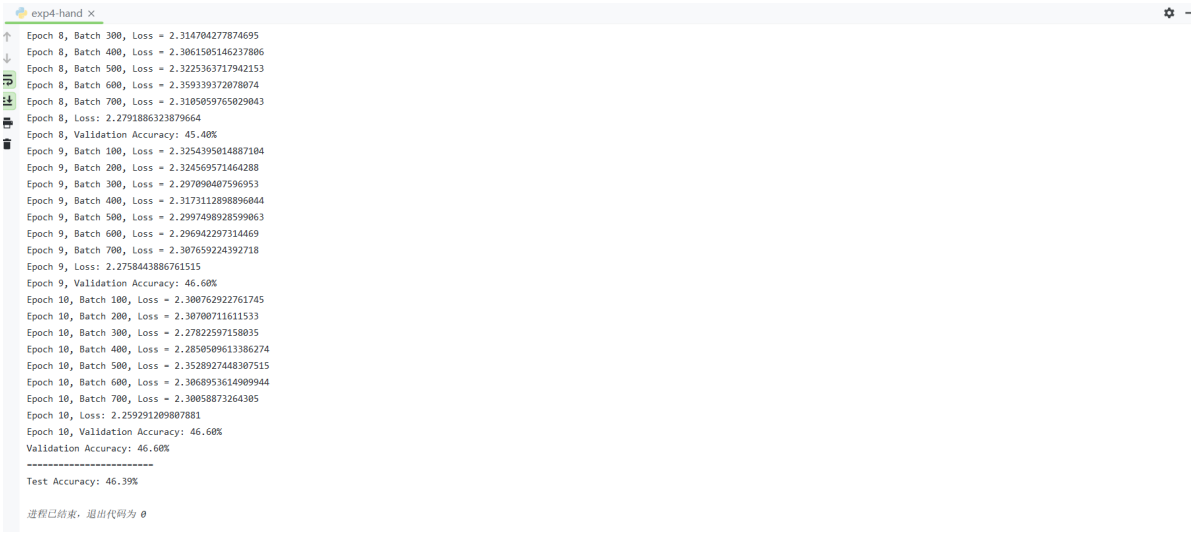
6.4 调整神经元个数

在保持原正则化系数$l_2 = 0.01$的基础上，将迭代次数为10次，设置：`batch_size = 64`。仍旧采用三层神经网络，隐藏层激活函数为ReLU，输出层激活函数为softmax。但是修改隐藏层神经元为128个，64个，32个神经元来观察运行的结果和运行的速度。

先来观察隐藏层神经元对准确率的影响：

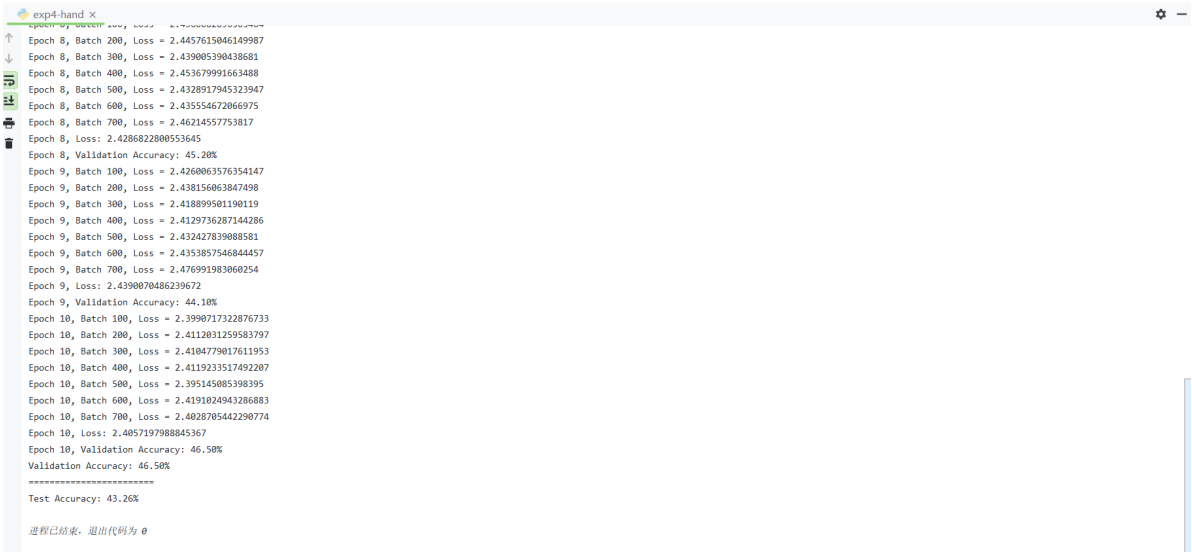
和原先保持一致，隐藏层神经元为1024个时：

本次运行中验证集准确率收束到46-50%左右，测试集准确46.39%。



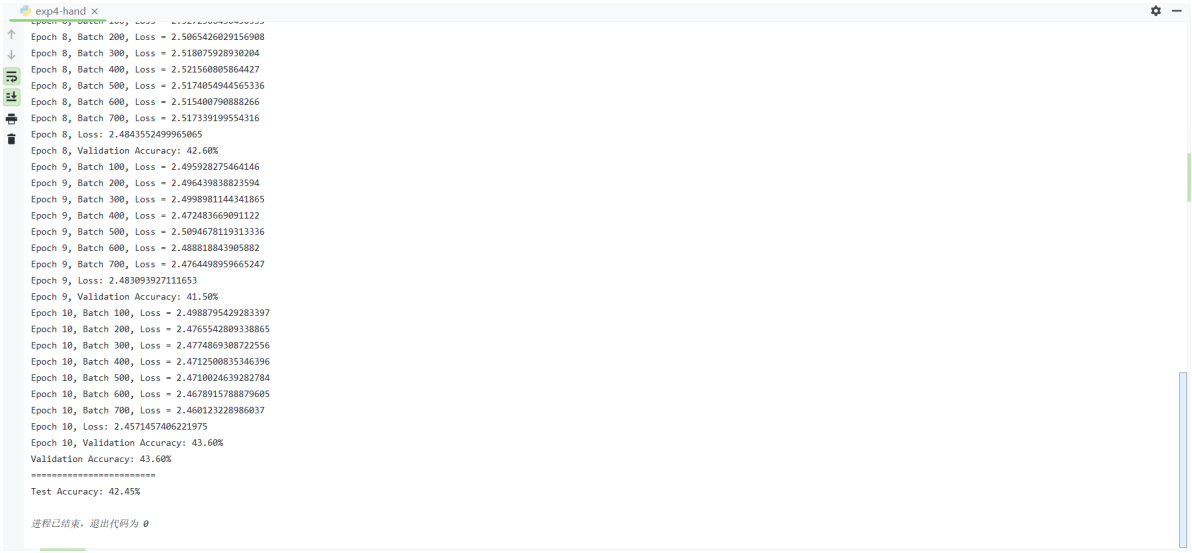
隐藏层神经元为128个时：

本次运行准确率在十次左右并没有收敛，最终验证集的准确里达到了46.5%，而测试集的准确率达到43.26%



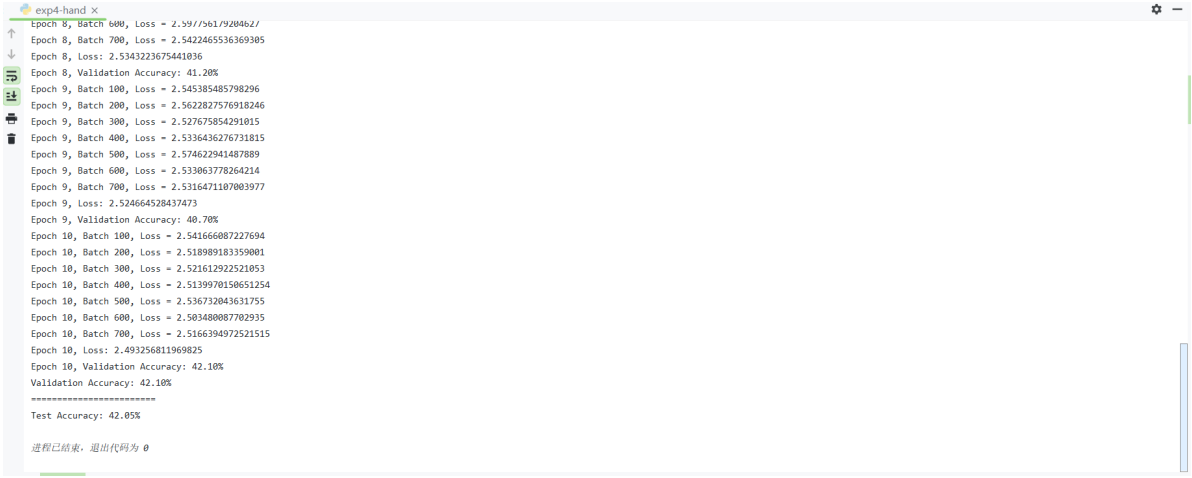
隐藏层神经元为64个时：

本次运行准确率在十次左右并没有收敛，最终验证集的准确里达到了43.6%，而测试集的准确率达到42.45%



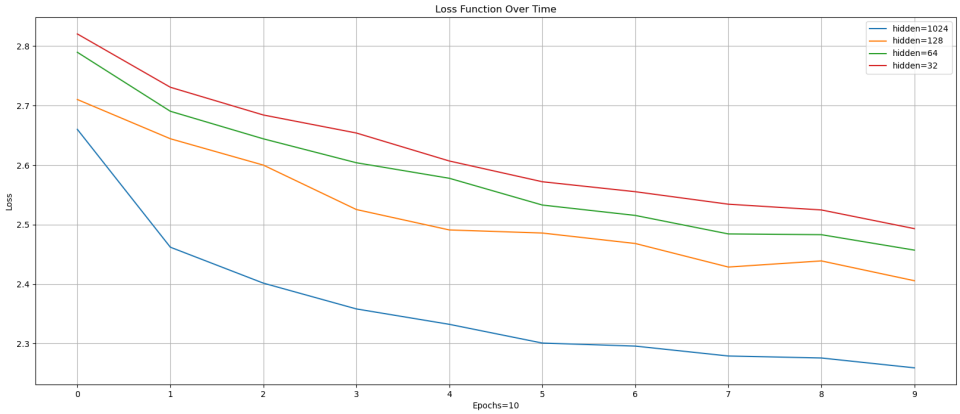
隐藏层神经元为32个时：

本次运行准确率在十次左右并没有收敛，最终验证集的准确里达到了42.1%，而测试集的准确率达到42.05%



从整体上来看，由于后面三组不同的神经元的准确率在10次训练中还未收束，三者仍有提升的空间，而其在验证集和测试集上的准确率大致相似，也就是说在本次构建的三层神经网络的模型中，隐藏层神经元的个数对收束速度有着更大的影响。而对于准确率的影响并不是很大。

再来观察其在其余条件相同条件相同时，损失值的变化情况，由该图像，也可大致看出其上述的结论：



为了进一步考究，我将迭代次数修改为30轮，观察其准确率和损失函数的变化情况，通过简单描述来阐述准确率的变化，和损失函数的整体图像。

和原先保持一致，隐藏层神经元为1024个时：

```
exp4-hand ×
Epoch 28, Batch 200, Loss = 2.2313730830477048
Epoch 28, Batch 300, Loss = 2.2240891816973534
Epoch 28, Batch 400, Loss = 2.25099733447562 84
Epoch 28, Batch 500, Loss = 2.2351743761795353
Epoch 28, Batch 600, Loss = 2.2164566998856743
Epoch 28, Batch 700, Loss = 2.2426355178487634
Epoch 28, Loss: 2.203852803255445
Epoch 28, Validation Accuracy: 47.30%
Epoch 29, Batch 100, Loss = 2.2347803446949452
Epoch 29, Batch 200, Loss = 2.240199654982672
Epoch 29, Batch 300, Loss = 2.2458881885978386
Epoch 29, Batch 400, Loss = 2.2468231675242722
Epoch 29, Batch 500, Loss = 2.2467874081727524
Epoch 29, Batch 600, Loss = 2.2355222042467506
Epoch 29, Batch 700, Loss = 2.2573695121612216
Epoch 29, Loss: 2.1996607249742666
Epoch 29, Validation Accuracy: 48.40%
Epoch 30, Batch 100, Loss = 2.2865359458122514
Epoch 30, Batch 200, Loss = 2.2687351300669922
Epoch 30, Batch 300, Loss = 2.2315498648911105
Epoch 30, Batch 400, Loss = 2.2730812981979733
Epoch 30, Batch 500, Loss = 2.269144530167029
Epoch 30, Batch 600, Loss = 2.2284770985201443
Epoch 30, Batch 700, Loss = 2.2260879911024696
Epoch 30, Loss: 2.179932742643238
Epoch 30, Validation Accuracy: 50.00%
Validation Accuracy: 50.00%
==========
Test Accuracy: 49.25%

进程已结束，退出代码为 0
```

隐藏层神经元为128个时：

```
exp4-hand ×
Epoch 28, Batch 200, Loss = 2.304677169880879
Epoch 28, Batch 300, Loss = 2.3079631309738966
Epoch 28, Batch 400, Loss = 2.283270272008283 4
Epoch 28, Batch 500, Loss = 2.317425058165458
Epoch 28, Batch 600, Loss = 2.311891868211502
Epoch 28, Batch 700, Loss = 2.2826755389269047
Epoch 28, Loss: 2.3412783136620514
Epoch 28, Validation Accuracy: 44.50%
Epoch 29, Batch 100, Loss = 2.3076956590574658
Epoch 29, Batch 200, Loss = 2.277038648176444
Epoch 29, Batch 300, Loss = 2.3617424094602217
Epoch 29, Batch 400, Loss = 2.3109476600455685
Epoch 29, Batch 500, Loss = 2.320321631287028
Epoch 29, Batch 600, Loss = 2.269807649263032
Epoch 29, Batch 700, Loss = 2.2817399385582267
Epoch 29, Loss: 2.273427800947568
Epoch 29, Validation Accuracy: 48.60%
Epoch 30, Batch 100, Loss = 2.295791312008938
Epoch 30, Batch 200, Loss = 2.3057329326128047
Epoch 30, Batch 300, Loss = 2.2948791341859383
Epoch 30, Batch 400, Loss = 2.272350000005392
Epoch 30, Batch 500, Loss = 2.301405959447428
Epoch 30, Batch 600, Loss = 2.262264037000632
Epoch 30, Batch 700, Loss = 2.2863257596544493
Epoch 30, Loss: 2.2639656761009173
Epoch 30, Validation Accuracy: 47.30%
Validation Accuracy: 47.30%
==========
Test Accuracy: 47.47%

进程已结束，退出代码为 0
```

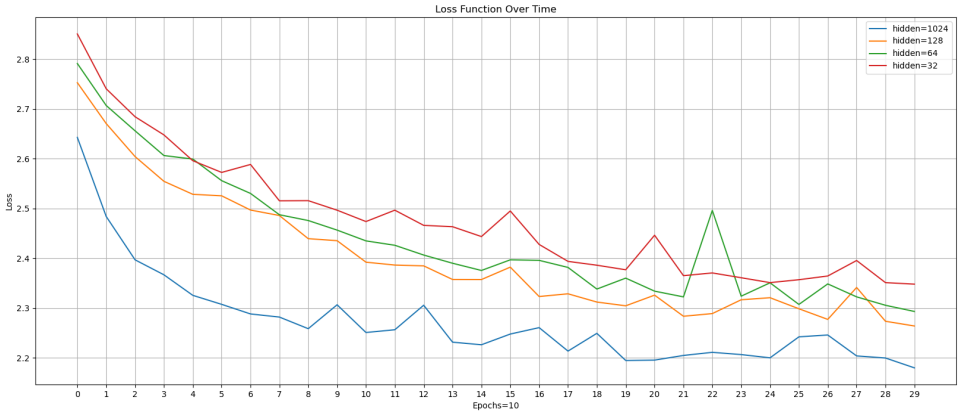
隐藏层神经元为64个时：

```
exp4-hand ×
Epoch 28, Batch 200, Loss = 2.3240026080246867
Epoch 28, Batch 300, Loss = 2.328244643007248
Epoch 28, Batch 400, Loss = 2.3269221035997654
Epoch 28, Batch 500, Loss = 2.30742645168855
Epoch 28, Batch 600, Loss = 2.2960887061767496
Epoch 28, Batch 700, Loss = 2.3312127853646802
Epoch 28, Loss: 2.3224294960040743
Epoch 28, Validation Accuracy: 46.40%
Epoch 29, Batch 100, Loss = 2.3378041143771666
Epoch 29, Batch 200, Loss = 2.297061623949185
Epoch 29, Batch 300, Loss = 2.3408701207216462
Epoch 29, Batch 400, Loss = 2.317527359096045
Epoch 29, Batch 500, Loss = 2.300657738024946
Epoch 29, Batch 600, Loss = 2.3141299700862263
Epoch 29, Batch 700, Loss = 2.3029525840307965
Epoch 29, Loss: 2.305521228554298
Epoch 29, Validation Accuracy: 47.50%
Epoch 30, Batch 100, Loss = 2.327639761382387
Epoch 30, Batch 200, Loss = 2.3012132800720093
Epoch 30, Batch 300, Loss = 2.299122803686157
Epoch 30, Batch 400, Loss = 2.315239343049853
Epoch 30, Batch 500, Loss = 2.308751710054098
Epoch 30, Batch 600, Loss = 2.3104640808025723
Epoch 30, Batch 700, Loss = 2.3129357967591844
Epoch 30, Loss: 2.2931219462864094
Epoch 30, Validation Accuracy: 47.40%
Validation Accuracy: 47.40%
------------------------
Test Accuracy: 46.60%

进程已结束，退出代码为 0
```

隐藏层神经元为32个时：

```
exp4-hand ×
Epoch 28, Batch 200, Loss = 2.3654025626300266
Epoch 28, Batch 300, Loss = 2.3621736987064983
Epoch 28, Batch 400, Loss = 2.3517367715305926
Epoch 28, Batch 500, Loss = 2.36705477650221
Epoch 28, Batch 600, Loss = 2.4110458981778584
Epoch 28, Batch 700, Loss = 2.351009071807512
Epoch 28, Loss: 2.395428984048809
Epoch 28, Validation Accuracy: 42.30%
Epoch 29, Batch 100, Loss = 2.40202801160514
Epoch 29, Batch 200, Loss = 2.3513813360033966
Epoch 29, Batch 300, Loss = 2.3551731889891827
Epoch 29, Batch 400, Loss = 2.3595291320228022
Epoch 29, Batch 500, Loss = 2.3571122884502484
Epoch 29, Batch 600, Loss = 2.371934992438387
Epoch 29, Batch 700, Loss = 2.3831805451297723
Epoch 29, Loss: 2.3509645080222485
Epoch 29, Validation Accuracy: 45.70%
Epoch 30, Batch 100, Loss = 2.3465215881199066
Epoch 30, Batch 200, Loss = 2.3687931367652357
Epoch 30, Batch 300, Loss = 2.3507571516874624
Epoch 30, Batch 400, Loss = 2.380406184685797
Epoch 30, Batch 500, Loss = 2.3490922484781587
Epoch 30, Batch 600, Loss = 2.359026760924254
Epoch 30, Batch 700, Loss = 2.3501273646470193
Epoch 30, Loss: 2.348000145208859
Epoch 30, Validation Accuracy: 45.80%
Validation Accuracy: 45.80%
------------------------
Test Accuracy: 45.56%

进程已结束，退出代码为 0
```

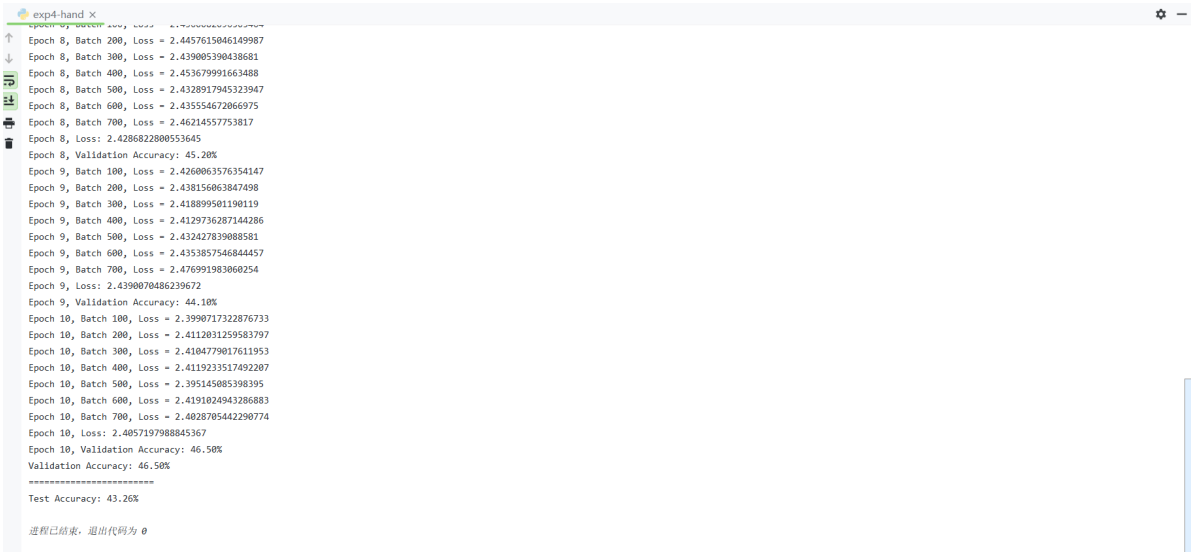
从上述的运行结果来看，更多的神经元有着更强的泛化能力和收敛速度，但是相比于1024个隐藏层神经元、128个神经元与其相比的区别并没有很大，或者采用256个神经元可能得到的效果会更加接近。我们再观察其对应的损失函数的变化。



6.5 改变层次结构

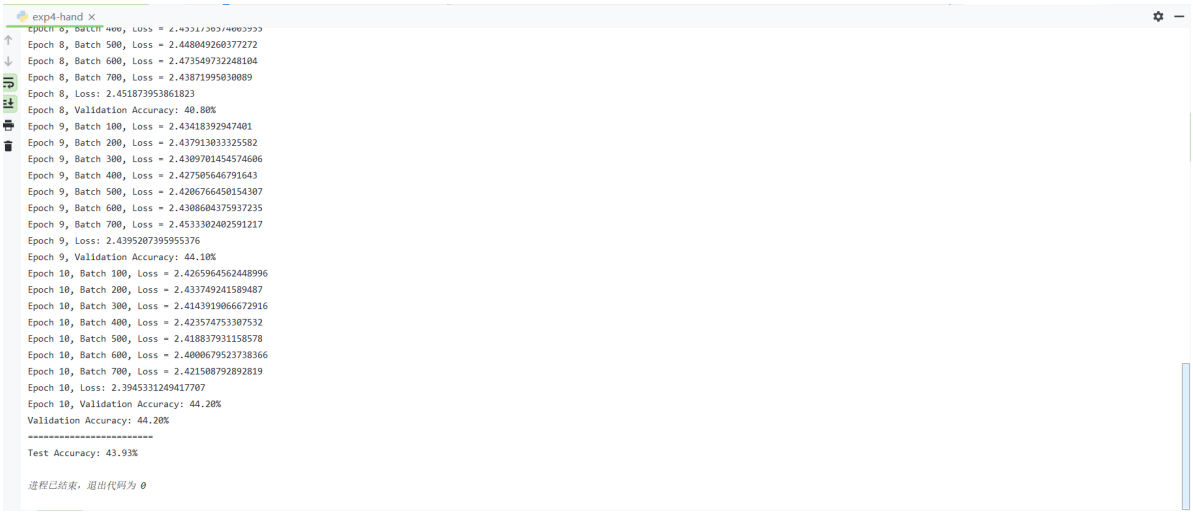
通过上述测试发现，第二层采用128个神经元的效果不错，相比于1024有着更快的迭代速度，相比于32和64个有着更好的准确率，因此再次基础上扩宽模型层数，来看迭代10次、30次的情况的准确率和损失值的变化情况。

原始模型：3024+128+10，隐藏层激活函数为ReLU，输出层激活函数为softmax，迭代10次



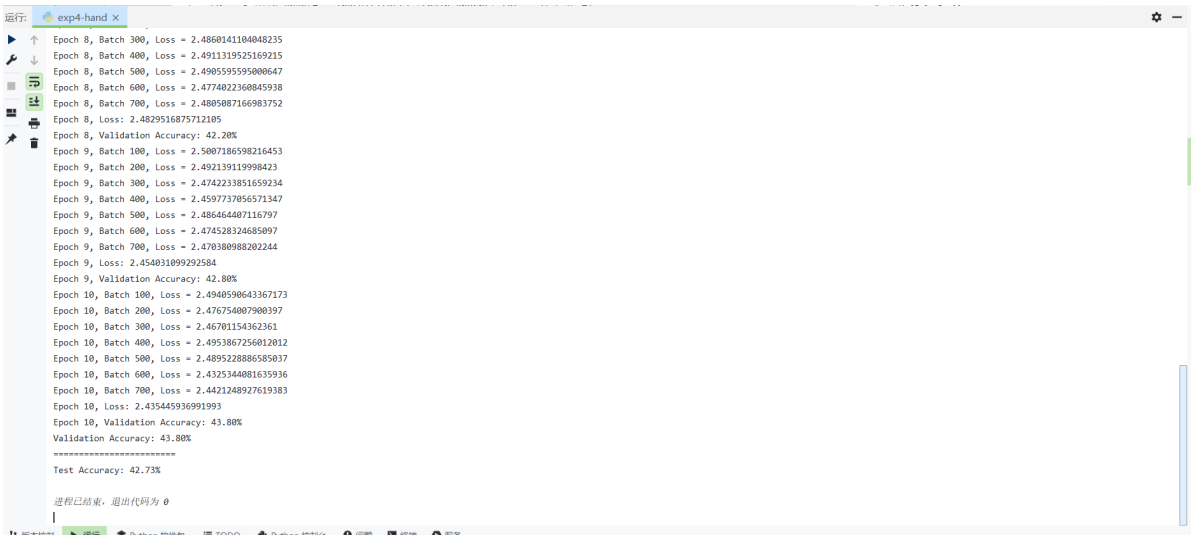
测试一：模型结构改变为：3024+128+64+10，隐藏层激活函数为ReLU，输出层激活函数为softmax，迭代10次

运行后结果为：



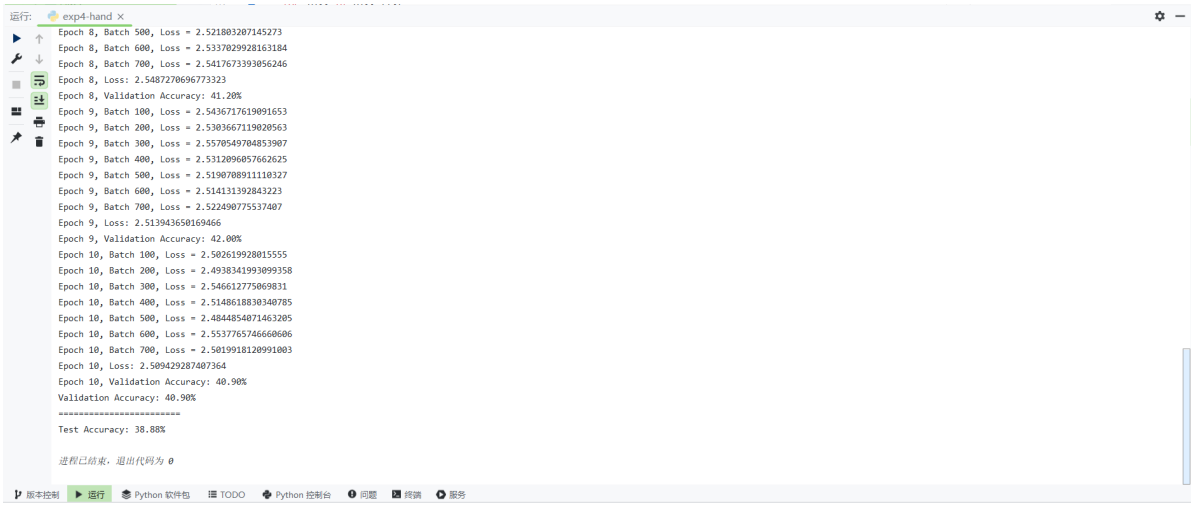
测试二：模型结构改变为：3024+128+32+10，隐藏层激活函数为ReLU，输出层激活函数为softmax，迭代10次

运行后结果为：



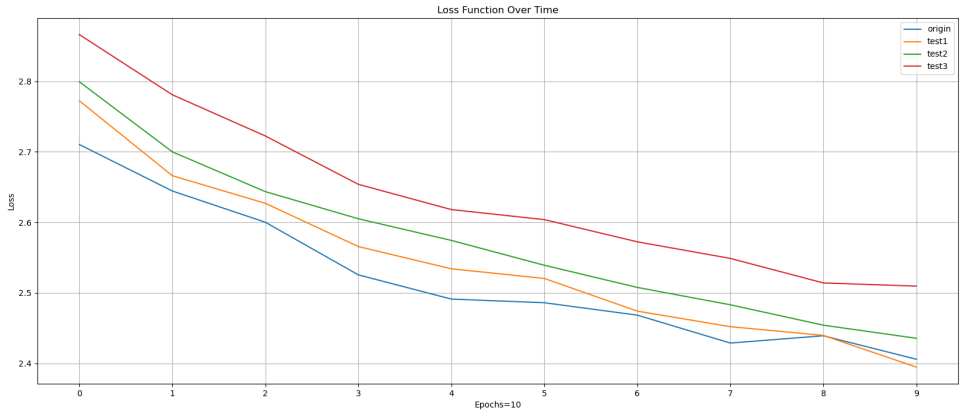
测试三：模型结构改变为：3024+128+64+32+10，隐藏层激活函数为ReLU，输出层激活函数为softmax，迭代10次

运行后结果为：



可以发现随着模型的改变，准确率可以得到一定的提高，由于整个过程都尚未收敛，因此认为改变模型结构对准确率提升有所帮助。

其相应的损失值变化情况如下：



7. 引入耐心与学习率衰减(优化学习率)

在上述尝试增强学习率的策略之后，我试图来实现一个动态的学习率变化机制。

考虑到之前训练的轮次较少，基本不会出现已经到了最优解附近的情况，但是如果继续扩大迭代次数，很有可能会在某些特定点附近非常接近最优解而损失函数在不断“蹦迪”。

那么我设计一个计数器，每轮训练结束后计算对应的损失值和准确率，记录到目前为止最好的模型和对应的损失值（这里不考虑准确率是因为样本太少，部分样本在参数轻微改变的情况下不会造成太大的准确率改变，此处损失值的变化更加敏感）。每当新的模型不如最优的模型时，计数器+1。每当计数器 达到一个阈值，那么学习率进行一个约定好的衰减，即乘以衰减因子。

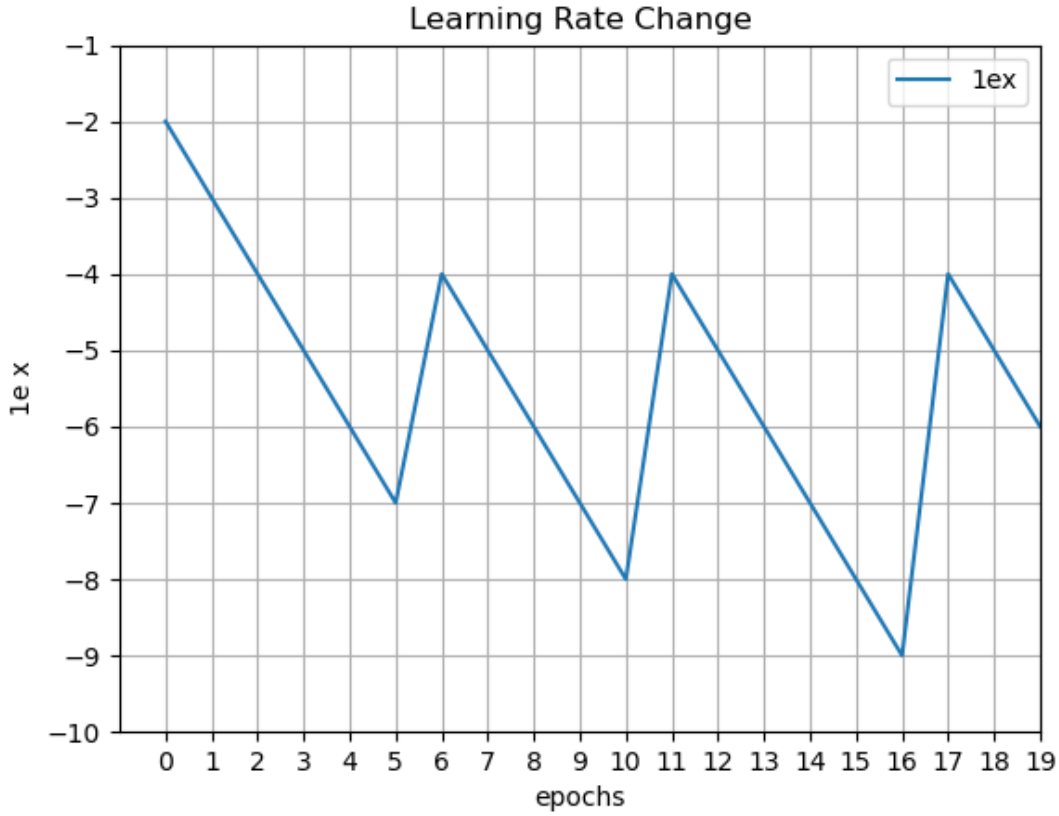
并且为了更好的跳出局部最优解，每当学习率衰减到某个临界时，将学习率进行回弹，使其试图跳出局部最优解。

这部分的代码也非常简单，即在`train`过程中引入如下代码：

```
    def train(self, X, y, epochs, batch_size, initial_lr, patience,
decay_factor=0.5, task='Exp4', val_size=0.2):
        # ... 保持不变
```

```
        best_loss = float('inf')
        best_accuracy = 0.0

        best_weight = None
        best_biases = None
        limit = 1e-7

        patience_counter = 0
        learning_rate = initial_lr

        for epoch in range(epochs):
            # ... 保持不变

            current_loss = self.loss_func(y, self.forward(X,training=False))

            y_pred = self.predict(X_test)
            current_accuracy = self.calculate_accuracy(y_test, y_pred)

            if current_loss < best_loss:
                best_loss = current_loss
                best_weight = self.weights
                best_biases = self.biases
                patience_counter = 0
            else:
                patience_counter += 1

            if patience_counter >= patience:
                learning_rate *= decay_factor
                if learning_rate < limit:
                    learning_rate = 1e-4
                    limit *= 0.1
                self.weights = best_weight
                self.biases = best_biases
                patience_counter = 0
                print(f"#########\n Learning rate reduced to {learning_rate}
\n#########")
        # ... 保持不变
```

假定初始学习率为 $learning\_rate = 0.01$，初始阈值为$limit = 1e-7$，回弹为$\cdot 1e-4$，那么学习率的变化大致如下：



8. 引入He Initialization(试图优化初始化)

He初始化和Xavier初始化是两种用于神经网络权重初始化的方法，它们的主要目的是解决网络训练过程中梯度消失或爆炸的问题。

1. **Xavier初始化（也称为Glorot初始化）**：
    - 由Xavier Glorot和Yoshua Bengio提出，主要用于Sigmoid和Tanh激活函数。
    - 初始化公式：

$$W \sim U(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}})$$

    - 或者

$$W \sim N(0, \sqrt{\frac{2}{n_{in}+n_{out}}})$$

      其中，n_{in}, n_{out} 是输出层神经元的数量。
2. **He初始化**：
    - 由Kaiming He等人提出，主要用于ReLU及其变种激活函数。
    - 初始化公式：

$$W \sim N(0, \sqrt{\frac{2}{n_{in}}})$$

      其中，n_{in}是输入层神经元的数量。
3. **普通初始化方法**：

    - 随机初始化：通常采用均匀分布或标准正态分布随机初始化权重，但容易导致梯度消失或梯度爆炸。
    - 零初始化：所有权重初始化为零，虽然简单，但会导致网络无法学习，因为对称性问题使得所有神经元更新相同的梯度。

- **理论基础**：
    - **Xavier初始化**基于保持前向传播和反向传播的方差一致，适用于Sigmoid和Tanh。
    - **He初始化**专门针对ReLU及其变种激活函数设计，目的是在前向传播时保持输出的方差稳定。
- **具体公式**：
    - Xavier初始化考虑了输入和输出层神经元的数量，使用的是均匀分布或正态分布。
    - He初始化只考虑输入层神经元的数量，通常采用正态分布。
- **应用场景**：
    - 如果使用Sigmoid或Tanh激活函数，通常选择Xavier初始化。
    - 如果使用ReLU或其变种激活函数，通常选择He初始化。

这些初始化方法相比普通的初始化方法，更能有效地避免梯度消失和梯度爆炸问题，提高训练效率和效果。

引入这一部分也非常的简单，即在代码的`init`部分，增加参数判断即可：

```
    def __init__(self, layers, activations, loss='mse', l2_lambda=0.01,
special_init = False):
        # 保持不变

        for i in range(len(layers) - 1):
            input_dim = self.layers[i]
            output_dim = self.layers[i + 1]

            if special_init:
                # He initialization for ReLU
                if activations[i] == 'relu':
                    weight = np.random.randn(input_dim, output_dim) *
np.sqrt(2 / input_dim)
                # Xavier initialization for sigmoid and tanh
                elif activations[i] in ['sigmoid', 'tanh']:
                    weight = np.random.randn(input_dim, output_dim) *
np.sqrt(2 / (input_dim + output_dim))
                else:
                    weight = np.random.randn(input_dim, output_dim) * 0.01
            else:
                weight = np.random.randn(input_dim, output_dim) * 0.1
            # 剩余保持不变
```

9. 引入dropout层(减少过拟合)

Dropout层是一种正则化技术，用于防止神经网络的过拟合。它通过在训练过程中随机地丢弃一部分神经元，使网络不依赖于某些特定的神经元或特征，从而增强模型的泛化能力。

9.1 Dropout的工作原理

1. **训练阶段**：
    - 在每个训练批次中，网络中的一部分神经元会被随机“丢弃”，即其输出被设为0。丢弃的比例由超参数`dropout rate`决定，通常设为0.2到0.5。
    - 这样，网络在每次前向传播时都会形成一个新的、较小的子网络，从而迫使模型学习到更加鲁棒的特征。
    - 丢弃的神经元并不会参与反向传播的梯度计算。
2. **测试阶段**：
    - 在测试阶段，所有的神经元都参与计算，但每个神经元的输出会按训练时的丢弃比例缩放，以补偿训练阶段的丢失神经元。比如，如果`dropout rate`为0.5，那么测试时每个神经元的输出会乘以0.5。

9.2 Dropout的优点

- **减少过拟合**：通过随机丢弃神经元，防止模型过于依赖某些特定的神经元，增强了模型的泛化能力。
- **提高训练效率**：丢弃神经元减少了前向和反向传播计算的数量，虽然每次训练的子网络较小，但总体训练时间并不会显著增加。

9.3 Dropout的缺点

- **增加了训练时间**：由于每次训练使用的是不同的子网络，训练过程中需要更多的迭代次数来达到收敛。
- **影响模型收敛性**：在某些情况下，dropout可能会导致训练过程不稳定，需要更精细的超参数调优。

9.4 实现

这一部分的实现也比较的简单，一是在初始化阶段引入`dropout_rate`参数、而是简单修改`forward`和`backward`函数，在每次`forward`中产生新的`mask_list`并带回进行计算。

`init`部分：

```
    def __init__(self, layers, activations, loss='mse', l2_lambda=0.01,
dropout_rate=0,
                 special_init = False):
        # 其余保持不变
        self.dropout_rate = dropout_rate
        self.dropout_masks = []
```

`forward`部分：

```
    def forward(self, x, training=True):
        self.activations = [x]
        self.linearcombination = [x]
        self.dropout_masks = []
        for w, b, activation_func in zip(self.weights, self.biases,
self.activation_funcs):
            z = np.dot(self.activations[-1], w) + b
            a = activation_func(z)

            if training and self.dropout_rate > 0:
                mask = np.random.binomial(1, 1 - self.dropout_rate,
size=a.shape) / (1 - self.dropout_rate)
```

```
                a *= mask
                self.dropout_masks.append(mask)
            else:
                self.dropout_masks.append(np.ones_like(a))

            self.linearcombination.append(z)
            self.activations.append(a)
        return self.activations[-1]
```

`backward` 部分：

```
    def backward(self, y_true):
        error = self.loss_derivative(y_true, self.activations[-1])
        for i in reversed(range(len(self.weights))):
            error *= self.activation_derivs[i](self.linearcombination[i + 1])

            if self.dropout_rate > 0:
                error *= self.dropout_masks[i]

            self.d_weights[i] = np.dot(self.activations[i].T, error) + self.l2_lambda * self.weights[i]
            self.d_biases[i] = np.sum(error, axis=0, keepdims=True)
            error = np.dot(error, self.weights[i].T)
```

则完成了dropout层的引入。

10. 整体测试

针对如上的若干策略，设计这样的模型结构，运行后观察其准确率和损失变化

10.1 参数设计

神经网络设计：3072+128+64+10

激活函数：隐藏层采用ReLU，输出层采用softmax

正则化参数：$l_2 = 0.01$

dropout_rate : 0.1

耐心值：5

衰减因子：0.1

初始学习率：0.01

初始化方案：He Initialization

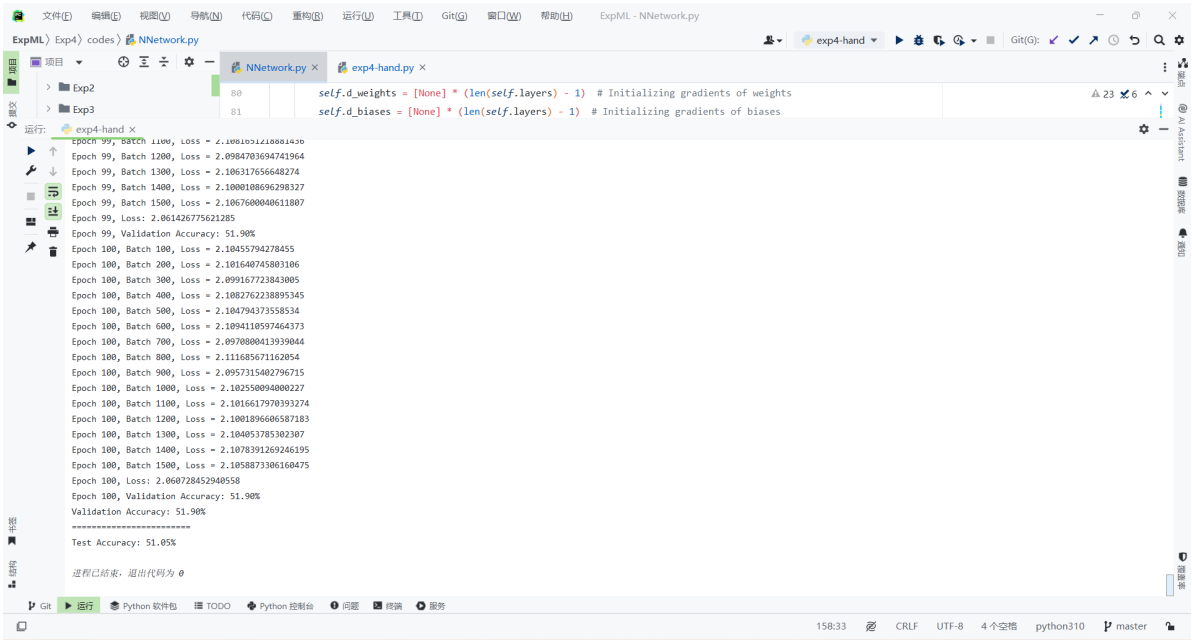
损失函数：交叉熵

迭代次数：100次

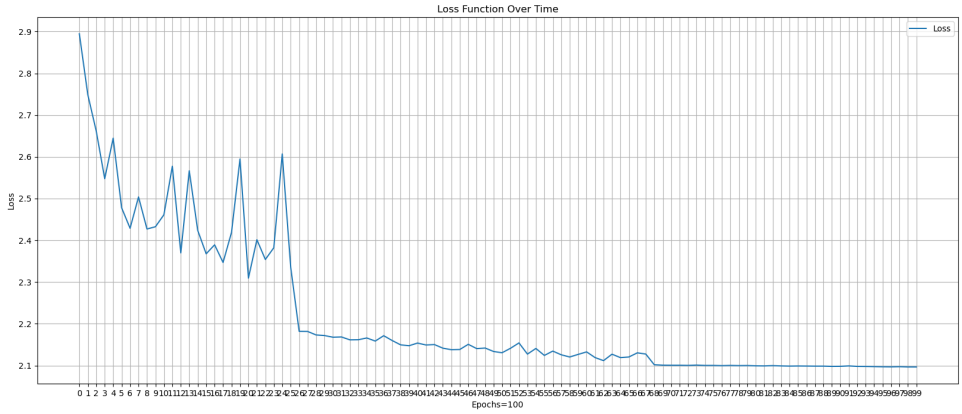
10.2 运行结果

运行后可以得到如下结果：

虽然最终的验证集准确率在51%左右，但是根据测试过程可以看出其大概在50次迭代后就已经接近50%，在最后25次的迭代次数中大致验证加早已超过50%。



损失值变化图：



11. 利用torch实现BP神经网络

使用PyTorch库定义了一个用于分类CIFAR-10数据集图像的神经网络。

11.1 网络设计

1. **导入库**:
    - 代码导入了`torch`、`torch.nn`、`torch.optim`以及`torchvision`等库，用于构建和训练神经网络。

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
```

2. **定义神经网络类**:
    - `BPNeuralNet`类继承自`nn.Module`，包括一个扁平化层和两个全连接层。

```
class BPNeuralNet(nn.Module):
    def __init__(self):
        super(BPNeuralNet, self).__init__()
        self.Flatten = nn.Flatten()
        self.fc1 = nn.Linear(3*32*32, 1024)
        self.ReLU = nn.ReLU()
        self.fc2 = nn.Linear(1024, 10)
```

- **层次结构**:
  - `Flatten`层将输入图像展平。
  - `fc1`全连接层将输入从3*32*32维度转换为1024维度。
  - `ReLU`激活函数添加非线性性。
  - `fc2`全连接层将1024维度转换为10维度（对应CIFAR-10的10个类别）。

3. **前向传播**:
   - `forward`方法定义了前向传播的过程。

```
def forward(self, x):
    x = self.Flatten(x)
    x = self.fc1(x)
    x = self.ReLU(x)
    x = self.fc2(x)
    return x
```

4. **预测函数**:
   - `predict`方法用于预测，并通过softmax函数将输出转换为概率分布。

```
def predict(self, x):
    x = self.forward(x)
    return torch.nn.functional.softmax(x, dim=1)
```

11.2 数据加载和预处理

1. **数据转换**:
   - 使用`transforms.Compose`定义了数据的转换和归一化操作。

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

2. **加载数据集**:
   - 加载CIFAR-10训练集和测试集，并使用`DataLoader`创建数据加载器。

```
trainSet = torchvision.datasets.CIFAR10(root='../data', train=True, download=False, transform=transform)
trainLoader = torch.utils.data.DataLoader(trainSet, batch_size=32, shuffle=True, num_workers=2)

testSet = torchvision.datasets.CIFAR10(root='../data', train=False, download=False, transform=transform)
testLoader = torch.utils.data.DataLoader(testSet, batch_size=32, shuffle=False, num_workers=2)
```

11.3 模型训练

1. **设备选择**:
    - 使用GPU（如果可用）进行训练。

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

2. **定义模型、损失函数和优化器**:
    - 定义了神经网络模型、交叉熵损失函数和随机梯度下降（SGD）优化器。

```
model = BPNeuralNet().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

3. **训练循环**:
    - 进行30个epoch的训练，每200个batch打印一次损失。

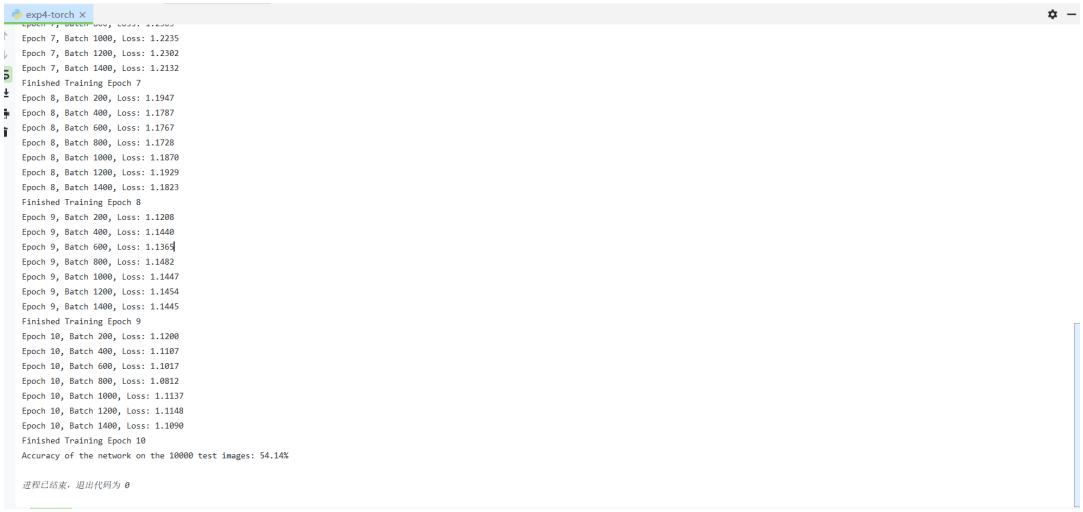
```
epochs = 30

for epoch in range(epochs):
    running_loss = 0.0
    for i, data in enumerate(trainLoader, 0):
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        if i % 200 == 199:
            print(f'Epoch {epoch + 1}, Batch {i + 1}, Loss: {running_loss / 200:.4f}')
            running_loss = 0.0
    print('Finished Training')
```

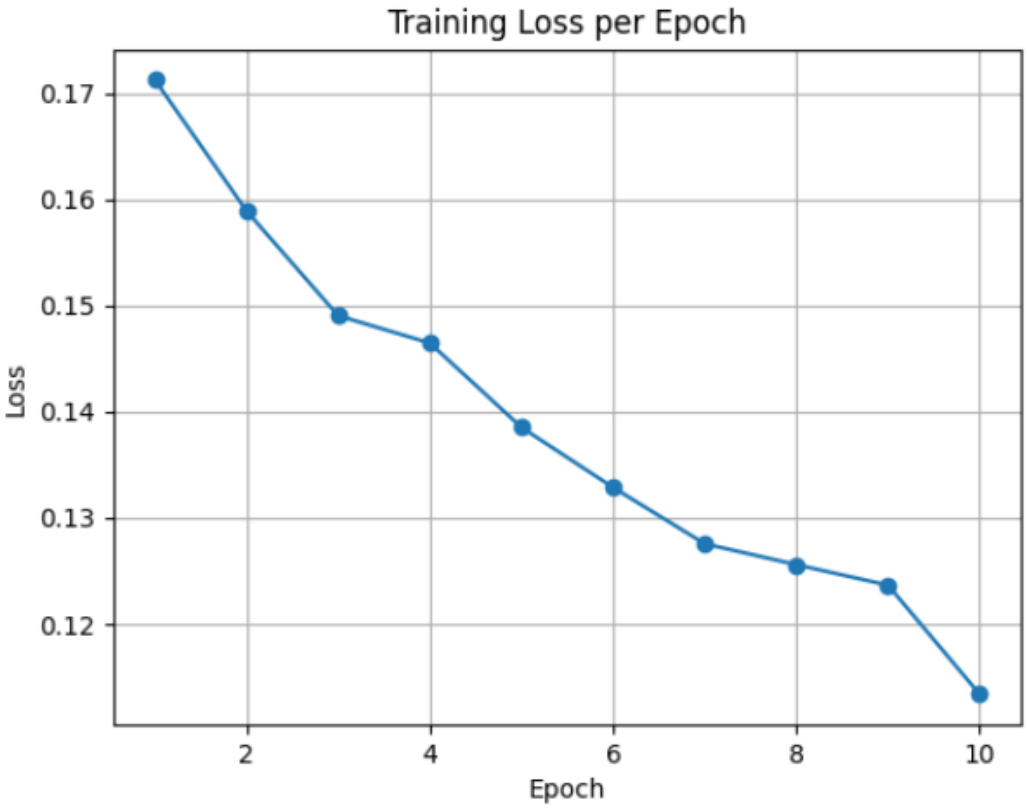
11.4 模型评估

1. **计算测试集准确率**:
    - 在测试集上评估模型的性能，并打印准确率。

```python
total = 0
correct = 0
with torch.no_grad():
    for data in testLoader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 *
correct / total:.2f}%')
```

2. 查看准确率以及绘制损失图像：





总的来说，torch实现的bp神经网络在优化的计算层面都比我个人手动实现的好的多，其batch_size采用的32，训练次数为10，结构为3072+1024+10的基础上远超我个人实现的代码。

代码结构

```
/Exp4/
|------- code/
|        |------- exp4-hand.py 执行手写BPNet的主函数
|        |------- NNetwork.py BP神经网络类
|        |------- exp4-torch.py 基于torch写的BP神经网络
|        |------- plot.py 部分图像绘制脚本
|
|-------- data/
|        |------- cifar-10-batches-py 基于python的cifar10数据集
|
|-------- pic/
|        |------- submit  部分试探过程截图
|        |------- other png files 生成的不同损失函数图像
|
|-------- backward.md 反向传播Markdown
|
|-------- backward.pdf 反向传播pdf
|
|-------- 实验报告.md 实验报告Markdown
|
|-------- 实验报告.pdf 实验报告pdf
```

心得体会

这次实验让我对机器学习中的神经网络训练过程有了更深入的理解和实际操作的经验。

数据预处理是机器学习中至关重要的一步，通过对数据的归一化处理，提升了模型的训练效果。模型的构建包括选择合适的激活函数和损失函数，ReLU和softmax的组合在本次实验中效果显著，比传统的sigmoid组合有更好的表现。

引入小批次梯度下降法后，模型的训练速度和稳定性显著提高。通过实验对比不同的激活函数组合，发现ReLU和softmax的组合在训练效率和准确率上都更优于其他组合。小批次处理有助于更细致地调整模型参数，使得模型能够更快地收敛到较优解。

通过引入L2正则化，减缓了模型过拟合的问题，提高了模型的泛化能力。正则化的引入对于处理复杂模型和大数据集尤为重要。

本次实验对比了不同的优化算法，通过对于 不同优化方法之间的比较，积累了一些调参和优化的经验。并且优化算法的选择对于模型的训练效果有显著影响，需要根据具体问题进行选择和调试。

通过对比手动实现的BP神经网络和基于PyTorch实现的模型，深刻体会到了深度学习框架在简化实现过程和提高计算效率上的优势。使用PyTorch框架进行模型训练和评估，大大减少了代码实现的复杂度，同时提高了模型的训练效率和准确性。

通过对实验结果的分析和可视化，更加直观地理解了不同参数和算法对模型训练效果的影响。绘制损失函数和准确率曲线，帮助我更好地判断模型的训练情况和优化效果。

总的来说，这次实验不仅巩固了我对神经网络理论知识的理解，还让我在实践中积累了宝贵的经验。从数据预处理、模型构建到优化算法的选择和使用深度学习框架，每一步都让我受益匪浅。这些经验为我今后的研究和实际应用打下了坚实的基础。