陈毓锐 18307130095

代码和输出说明：

Class.py 定义了若干个父类：Layer，Model，Loss，在 BP.py 中具体实现。

在 BP.py 中：
类定义部分：
全连接层 LinearLayer()，卷积层 Convolution()，继承了 Layer，他们都有 get_output()，get_gradient()，update()，其功能分别是传入一个 input，计算每层的输出并保存；传入最后一层的梯度，反向传播得到每一层权重和输出的梯度；更新权重。

LinearLayer(inputlen=256,hidden=[10],activation=ReLu(),dropout=0, bias=False)
它可以包含多个隐层，指定 hidden 数组作为隐层的每一层神经元个数，改变 activation 激活函数类型，是否 dropout。bias=True 的作用是让每一层的其中一个神经元作为常数。

Convolution(img_size=[16, 16], cout=3, kernel_size=5, pool_size=3 ,activation=ReLu(), pooling=MaxPool())
可以指定输出通道 Cout，核大小 kernel_size，池化操作的大小 pool_size，激活函数和池化层函数，它只有一层。

MSE()，Cross_Entropy()则都继承 Loss
他们都有 get_loss_value()和 get_gradient(), 功能分别为传入 input 和 label 计算损失函数值，和计算梯度。

Relu()，Identity()继承自 Activation，他们是激活函数，都有 h()，传入 input 得到输出；get_gradient()，给 input 得到梯度。

MLP()继承自 Model，它将若干 Layer 整合在一起，forward()将前一层的输出作为后一层的输入，从而得到最后输出；backprop()则传入损失函数的梯度，反向传播至每一层，得到各自参数的梯度；update()则更新每一个 Layer 的参数。默认的模型为两个 LinearLayer，前一个多层，且激活函数为 Relu()，后一个为单层，激活函数为 Identity()。

Dataloader()则是将数据打乱归一化，并按照 batch size 取出的数据装载工具。并且在之后的训练中 **batch size 都取 8**。

MaxPool(size=2)，池化操作，有 get_out()和 get_gradient()

函数部分：
train()将 loss 和 model 结合，对数据进行拟合，期间的输出轮数是 batch 的训练个数，后面的 loss value 是 100 个 batch 的平均损失函数。每训练 2500 个 batch 就进行一次在验证集和测试集上的预测，从而得到最好的模型和迭代次数。
Verify()则是传入模型和数据集，得到预测准确率。
rotate()，stretch()，vague()则是对数据集的操作：图形旋转，拉伸和标签模糊化。(用了 cv2)

1.

在 hidden=[10], loss function 为 MSE, 激活函数为 Leaky Relu 时, 预测准确率已经为 90.3 %。

```
444500 : 0.15377012272021928
444600 : 0.15953017120402513
444700 : 0.1689535032146399
444800 : 0.15643071718090668
444900 : 0.15768602130902643
445000 : 0.15631445795152918
vaild:
Accuracy: 0.905
test:
Accuracy: 0.9
best acc: 0.903   ----  best iteration: 259999
```

更改 hidden=[32,32,64,64,32,16]，模型更为复杂，具有更好的泛化性能，预测准确率达到了 95%，但是训练的 batch 数远比 hidden=[10]时多，收敛较慢。

```
999800 : 0.0842785177222993
999900 : 0.08832255691766527
1000000 : 0.08546906391980291
vaild:
Accuracy: 0.938
test:
Accuracy: 0.95
best acc: 0.95   ----  best iteration: 987499
```

2.

只要记录上一步的权重值，根据公式即可实现，并且在更新权重后将最新值更新到 last_w, last_b。
利用动量法，在大部分情况下能够减少训练的次数。

```
def update(self, lr=0.001, momentum=0, regularization=0):
    for i in range(len(self.w)):
        delta_w = self.w[i]-self.last_w[i]
        delta_b = self.b[i]-self.last_b[i]
        if self.bias is False:
            self.w[i] = self.w[i] - lr * self.grad_w[i] + momentum * delta_w - lr * regularization * self.w[i]
            self.b[i] = self.b[i] - lr * self.grad_b[i] + momentum * delta_b
        else:
            self.b[i] = self.b[i] +\
                        (- lr * self.grad_b[i] + momentum * delta_b)*1
            self.w[i] = self.w[i] +\
                        (- lr * self.grad_w[i] + momentum * delta_w - lr * regularization * self.w[i])*self.bias_mat[i]

        self.last_w[i] = self.w[i]
        self.last_b[i] = self.b[i]
```

3.

最初实现已经用 numpy 的向量操作。在每一个类的 get_gradient()中都用了向量操作。

4.

同上图，实现二范数的正则化，只要在每步迭代时减去 lr*regularization*w 即可。
加入正则化后会使 loss value 更大，同时也收敛更早。

```
947400 : 0.131768652145308
947500 : 0.12272275259437743
vaild:
Accuracy: 0.909
test:
Accuracy: 0.91
best acc: 0.91   ----  best iteration: 919999
```

5.

交叉熵的向量实现

```python
class Cross_Entropy(Loss):
    # 默认y为one-hot形式
    def get_gradient(self):
        # give the label y and compute the gradient of the last layer
        y = self.y
        input = self.input
        p = (y*self.p).sum(axis=1, keepdims=True)
        batch_size, _ = y.shape
        self.grad = -(y-self.p)/batch_size

        return self.grad

    def get_loss_value(self, y, input):
        # give the input and label to calculate the loss value
        self.y = y
        self.input = input
        batch_size, _ = y.shape
        e_ip = np.exp(input)
        value = e_ip/e_ip.sum(axis=1, keepdims=True)
        self.p = value
        value = np.log(value)
        loss = np.sum(value*y)/batch_size
        self.loss = -loss

        return self.loss
```

交叉熵损失函数不一定比 MSE 要更优秀：同样的模型（hidden=[32,32,64,64,32,16]），MSE 能够到 95%的预测准确率（第一问中的图），而交叉熵只有 93.1%,；而第一层用卷积的话，交叉熵则可以提升准确率。但用交叉熵损失函数可以加快收敛的速度。

```
59900 : 0.08599778774890453
60000 : 0.09101030463249614
vaild:
Accuracy: 0.93
test:
Accuracy: 0.931
best acc: 0.931   ----  best iteration: 59999
```

6.

在全连接层实例化之初构造一个 bias_mat 的 list，其中每个元素都为一个第 1 个点为 0 其他全为 1 的列向量，0 的位置表明了将每一层的第 1 个神经元作为常数，在更新的时候，每一层的权重 w 要乘以对应的 bias_mat 中的列向量，即不更新第一个神经元的权重。

```
else:
    self.b[i] = self.b[i] +\
                (- lr * self.grad_b[i] + momentum * delta_b)*1
    self.w[i] = self.w[i] +\
                (- lr * self.grad_w[i] + momentum * delta_w - lr * regularization * self.w[i])*self.bias_mat[i]
```

7.

每次 forward 时都要对每层神经元随机一个 0-1 向量 dropout_vec（create_dropout），为 0 的概率为 dropout 值，之后每层输出都要乘以对应向量，即把某些神经元输出置 0；在 backpropagation 时，loss 对每一层输出的导数要乘以后一层的 dropout_vec，即在此次更新中忽略该层输出的影响。

```
    def create_dropout(self):
        ## 随机每层dropout的神经元向量
        dropout = 1-self.dropout
        inputlen,_ = self.w[0].shape
        self.dropout_vec = []
        self.dropout_vec.append(np.random.binomial(1,dropout,size=[inputlen,]))
        for i in range(self.l-1):
            length,_ = self.w[i+1].shape
            self.dropout_vec.append(np.random.binomial(1,dropout,size=[length,]))
```

```
def get_output(self, input):
    if self.dropout > 0 and self.ifdrop:
        # ifdrop指是在预测还是在训练

        self.create_dropout() ## 随机得到dropout的神经元向量
        self.out[0] = self.activation.h(np.dot(input*self.dropout_vec[0], self.w[0]) + self.b[0])
        for i in range(self.l - 1):
            self.out[i] = self.out[i]*self.dropout_vec[i+1]
            yi = np.dot(self.out[i], self.w[i + 1]) + self.b[i + 1]
            self.out[i + 1] = self.activation.h(yi)
        self.output = self.out[self.l - 1]
        return self.output
```

Dropout 训练完后预测时，需要将训练完的参数乘上一个系数以满足几何平均的近似。

```
    def change_weight_to_fix_dropout(self):
        ## 训练完后改变权重以适应dropout模型的预测
        p = 1-self.dropout
        for i in range(self.l):
            self.w[i] = self.w[i]*p
            self.b[i] = self.b[i]*p
        self.ifdrop = False

    def change_weight_to_train(self):
        ## 改回参数为训练模式
        p = 1-self.dropout
        for i in range(self.l):
            self.w[i] = self.w[i]/p
            self.b[i] = self.b[i]/p
        self.ifdrop = True
```

Dropout 类似于集成训练，训练多个模型，所以训练时间会大大延长，并且使用 dropout 时，loss value 会变大（dropout 掉一些神经元后计算的 loss value），dropout 过大时，预测准确率会下降。
在使用 dropout=0.2 时，需要 50 万个 batch 才能得到最优值

```
549800 : 0.6022726543395064
549900 : 0.6940247240470305
550000 : 0.6190214421902859
vaild:
Accuracy: 0.938
test:
Accuracy: 0.933
best acc: 0.937   ----  best iteration: 539999
550100 : 0.6324420535881594
```

8.

更新模型的时候，只更改最后一层的权重即可。

```
    def update(self, lr=0.001, momentum=0, regularization=0):
        ## 更新每一层的参数
        for layer in self.Layers:
            layer.update(lr, momentum, regularization)

    def update_only_last_layer(self, lr=0.001, momentum=0, regularization=0):
        ## 只更新最后一层的参数
        self.Layers[-1].update(lr, momentum, regularization)
```

fine-tuning 对模型预测准确率提升作用不明显，有时会产生过拟合的现象。

9.

利用 cv2 的包可以对图片进行拉伸。
Stretch 函数对图片进行了随机旋转随机平移。

```
## 图片拉伸
def stretch(data, label):
    batch_size = data.shape[0]
    new_label = label + 0
    new_data = np.zeros(shape=[batch_size, 256])
    for i in range(batch_size):
        slice = data[i]
        img = slice.reshape([16,16])
        percent_l = random.uniform(0.7, 1)
        percent_w = random.uniform(0.7, 1)
        n_l = int(percent_l*16)
        n_w = int(percent_w*16)
        new_img = cv.resize(img, (n_l, n_w))
        delta_l = i % (16-n_l)
        delta_w = i % (16-n_w)
        ss = np.zeros(shape=[16,16])
        ss[delta_w:delta_w+n_w, delta_l:delta_l+n_l] = new_img
        s = ss.flatten()
        new_data[i] = s

    return new_data, new_label
## 图片旋转
def rotation(data):
    batch_size = data.shape[0]
    res = data.reshape([batch_size, 16, 16])
    res = res.transpose([0,2,1])
    res = res.reshape([batch_size, 256])
    return res
```

数据增广对于模型提升有限，拉伸图片能够提升 0.7%左右的准确率，但是对于全连接神经网络，图片旋转反而会使模型预测准确率下降。

10.

定义了卷积层加入到 MLP 中
卷积层的激活函数为 Leaky Relu，池化层为 maxpooling。
在卷积操作中，先对 input 进行平移复制，后用爱因斯坦求和，总体性能提升了 3 倍左右。

```
def get_output(self, input):
    batch_size = input.shape[0]
    input = input.reshape([batch_size, self.size[0], self.size[1]])
    self.in_size = input.shape
    r = self.r
    transf_in = np.zeros(shape=[self.kernel_size, self.kernel_size, batch_size, self.size[0] - 2 * r, self.size[1] - 2 * r])
    a = self.size[0] - 2 * r
    b = self.size[1] - 2 * r
    ## 用爱因斯坦求和以加快卷积操作
    for i in range(self.kernel_size):
        for j in range(self.kernel_size):
            transf_in[i, j] = input[:, i:i+a, j:j+b]
    self.out[0] = np.einsum('ijkln,hij->hkln', transf_in, self.kernel)+self.b.reshape([self.cout,1,1,1])
    self.out[1] = self.activation.h(self.out[0])
    self.out[2] = self.pooling.get_out(self.out[1])
    res = self.out[2].transpose((1,0,2,3))
    self.outshape = res.shape
    res = res.reshape([batch_size, self.outlen])
    self.output = res

    return res
```

```
def get_gradient(self, input, out_grad):
    input = input.reshape(self.in_size)

    out_grad = out_grad.reshape(self.outshape)
    out_grad = out_grad.transpose((1,0,2,3))

    self.grad_out[2] = self.pooling.get_gradient(out_grad)
    self.grad_out[1] = self.activation.get_gradient(self.out[1])*self.grad_out[2]

    cout, batch_size, a, b = self.grad_out[1].shape
    for i in range(cout):
        grad_slice = self.grad_out[1][i]
        input_slice = input
        self.grad_b[i] = np.sum(grad_slice)
        for j in range(self.kernel_size):
            for k in range(self.kernel_size):
                self.grad_kernel[i, j, k] = np.sum(grad_slice * input_slice[:, j:j + a, k:k + b])
    return 0
```

卷积层对模型的提升效果较为明显，在 kernel size=5，max pool size=3，输出通道 Cout=32 时，结合全连接层，模型预测准确率能达到 97.9%。并且使用卷积神经网络，loss value 能够相对更快地收敛，说明卷积比较贴合真实模型。

最优结果：

在模型参数为：

cnn = Convolution(cout=32, pool_size=3, kernel_size=5)

a = MLP(256,10,[32,32,64,64,32,16], convolution_layer=cnn)

卷积层输出通道 64，核大小为 5，池化操作 size 为 3；全连接层向量为[32,32,64,64,32,16]，激活函数为 Leaky Relu（a=0.01），输出层没有激活函数

损失函数选择交叉熵

随机数种子 seed = 1000

lr=0.001， momentum=0.9，regularization=0，每 5000 个 batch 学习率乘以 0.9

增加了平移和拉伸后的 5000 个新数据进入到训练集中

所有数据被归一化，即均值为 0，方差为 1

所有层的参数初始化用 He 方法

模型在训练 40000 个 batch 后达到最优，batch size=8，在测试集上的预测准确率为 98.1%

```
67200 : 0.005865777828229241
67300 : 0.008680359563778611
67400 : 0.007004811409858109
67500 : 0.006439124913743364
vaild:
Accuracy: 0.977
test:
Accuracy: 0.98
best acc: 0.981   ----  best iteration: 39999
```