清华大学电子工程系

媒体与认知 课堂 2

2021-2022 学年春季学期

作业 2

孙一 2019010976 2022 年 3 月 24 日

理论部分

1 单选题（15 分）

1.1 D

1.2 C

1.3 B

1.4 C

1.5 D

2 计算题（15 分）

设隐含层为 $\mathbf{z}=\mathbf{x}\mathbf{W}^T+\mathbf{b}$，其中 $\mathbf{x}\in R^{(1\times m)}$，$\mathbf{z}\in R^{(1\times n)}$，$\mathbf{W}\in R^{(n\times m)}$，$\mathbf{b}\in R^{(1\times n)}$ 均为已知，其激活函数如下：

$$\mathbf{y}=\tanh(\mathbf{z})=\frac{e^{\mathbf{z}}-e^{-\mathbf{z}}}{e^{\mathbf{z}}+e^{-\mathbf{z}}}$$

若训练过程中的目标函数为 L，且已知 L 对 $\mathbf{y}$ 的导数 $\frac{\partial L}{\partial \mathbf{y}}=[\frac{\partial L}{\partial y_1},\frac{\partial L}{\partial y_2},...,\frac{\partial L}{\partial y_n}]$ 和 $\mathbf{y}=[y_1,y_2,...,y_n]$ 的值。

2.1 请使用 y 表示出 $\frac{\partial \mathbf{y}}{\partial \mathbf{z}}$

解：激活函数 $\tanh(\mathbf{z})=\frac{e^{z}-e^{-\mathbf{z}}}{e^{z}+e^{-z}}$是逐元素操作

$$\because \frac{\partial y_i}{\partial z_j}=0\ (i\neq j)\qquad \frac{\partial y_i}{\partial z_i}=\frac{4}{(e^{z_i}+e^{-z_i})^2}=1-{y_i}^2$$

$$\therefore \frac{\partial \mathbf{y}}{\partial \mathbf{z}}=1-\mathbf{y}^2$$

2.2 请使用 y 和 $\frac{\partial L}{\partial \mathbf{y}}$ 表示 $\frac{\partial L}{\partial \mathbf{x}}$，$\frac{\partial L}{\partial \mathbf{W}}$，$\frac{\partial L}{\partial \mathbf{b}}$。

提示:$\frac{\partial L}{\partial \mathbf{x}}$，$\frac{\partial L}{\partial \mathbf{W}}$，$\frac{\partial L}{\partial \mathbf{b}}$ 与 x,W,b 具有相同维度。

矩阵计算的具体过程

$\because \boldsymbol{z} = \boldsymbol{x}W^T + \boldsymbol{b}$

$$\begin{bmatrix} z_1 & \cdots & z_n \end{bmatrix} = \begin{bmatrix} x_1 & \cdots & x_m \end{bmatrix} \begin{bmatrix} W_{11} & W_{12} & \cdots & W_{1m} \\ W_{21} & W_{22} & \cdots & W_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ W_{n1} & W_{n2} & \cdots & W_{nm} \end{bmatrix}^T + \begin{bmatrix} b_1 & \cdots & b_n \end{bmatrix}$$

$$= \begin{bmatrix} x_1 & \cdots & x_m \end{bmatrix} \begin{bmatrix} W_{11} & W_{21} & \cdots & W_{n1} \\ W_{12} & W_{22} & \cdots & W_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ W_{1m} & W_{2m} & \cdots & W_{nm} \end{bmatrix} + \begin{bmatrix} b_1 & \cdots & b_n \end{bmatrix}$$

$$= \begin{bmatrix} x_1W_{11} + x_2W_{12} + \cdots + x_mW_{1m} + b_1 & \cdots & x_1W_{n1} + x_2W_{n2} + \cdots + x_mW_{nm} + b_n \end{bmatrix}$$

$\therefore \frac{\partial z_i}{\partial x_j} = W_{ij}$，即权重矩阵$W$的第$i$行第$j$列的元素

$\therefore \frac{\partial z_i}{\partial W_{jk}} = x_k$，即$\boldsymbol{x}$的第$k$个元素

$\therefore \frac{\partial z_i}{\partial z_j} = 0\,(i \neq j)\,, \frac{\partial z_i}{\partial z_i} = 1$

至此我们求得了 z_i 对 x_j、W_{jk}、b_j 的导数，损失函数 L 通过 $\boldsymbol{y}$ 计算得到。由于激活函数逐元素计算的特点，y_i 只与 z_i 有关。比如分析 $\frac{\partial L}{\partial \mathbf{x_i}}$，我们需要找出 L 中所有和 x_i 有关的元素，分别让其对 x_i 求导，然后相加，填入与 $\boldsymbol{x}$ 同形的向量的第 i 个位置即可。

下面分别计算题目中要求的三个偏导数

$$\therefore \frac{\partial L}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial L}{\partial x_1} & \cdots & \frac{\partial L}{\partial x_m} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial z_2}\frac{\partial z_2}{\partial x_1} + \cdots + \frac{\partial L}{\partial y_n}\frac{\partial y_n}{\partial z_n}\frac{\partial z_n}{\partial x_1} & \cdots & \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial z_1}\frac{\partial z_1}{\partial x_m} + \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial z_2}\frac{\partial z_2}{\partial x_m} + \cdots + \frac{\partial L}{\partial y_n}\frac{\partial y_n}{\partial z_n}\frac{\partial z_n}{\partial x_m} \end{bmatrix}$$

$$= \left(\frac{\partial L}{\partial \boldsymbol{y}} * \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}}\right) \cdot \boldsymbol{W} = \frac{\partial L}{\partial \boldsymbol{y}} * (1 - \boldsymbol{y}^2) \cdot \boldsymbol{W}$$

$$\therefore \frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \cdots & \frac{\partial L}{\partial w_{1m}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \cdots & \frac{\partial L}{\partial w_{2m}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial w_{n1}} & \frac{\partial L}{\partial w_{n2}} & \cdots & \frac{\partial L}{\partial w_{nm}} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial z_1}\frac{\partial z_1}{\partial w_{11}} & \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial z_1}\frac{\partial z_1}{\partial w_{12}} & \cdots & \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial z_1}\frac{\partial z_1}{\partial w_{1m}} \\ \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial z_2}\frac{\partial z_2}{\partial w_{21}} & \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial z_2}\frac{\partial z_2}{\partial w_{22}} & \cdots & \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial z_2}\frac{\partial z_2}{\partial w_{2m}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial y_n}\frac{\partial y_n}{\partial z_n}\frac{\partial z_n}{\partial w_{n1}} & \frac{\partial L}{\partial y_n}\frac{\partial y_n}{\partial z_n}\frac{\partial z_n}{\partial w_{n2}} & \cdots & \frac{\partial L}{\partial y_n}\frac{\partial y_n}{\partial z_n}\frac{\partial z_n}{\partial w_{nm}} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial z_1}x_1 & \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial z_1}x_2 & \cdots & \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial z_1}x_m \\ \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial z_2}x_1 & \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial z_2}x_2 & \cdots & \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial z_2}x_m \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial y_n}\frac{\partial y_n}{\partial z_n}x_1 & \frac{\partial L}{\partial y_n}\frac{\partial y_n}{\partial z_n}x_2 & \cdots & \frac{\partial L}{\partial y_n}\frac{\partial y_n}{\partial z_n}x_m \end{bmatrix}$$

$$= \left[\left(\frac{\partial L}{\partial \boldsymbol{y}}\right) * \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}}\right)\right]^T \cdot \boldsymbol{x}$$

$$= \left(\frac{\partial L}{\partial \boldsymbol{y}}\right)^T * (1 - \boldsymbol{y}^2)^T \cdot \boldsymbol{x}$$

$$\therefore \frac{\partial L}{\partial b} = \begin{bmatrix} \frac{\partial L}{\partial b_1} & \cdots & \frac{\partial L}{\partial b_n} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial z_1}\frac{\partial z_1}{\partial b_1} & \cdots & \frac{\partial L}{\partial y_n}\frac{\partial y_n}{\partial z_n}\frac{\partial z_n}{\partial b_n} \end{bmatrix}$$

$$= \left(\frac{\partial L}{\partial \boldsymbol{y}} * \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}}\right) = \frac{\partial L}{\partial \boldsymbol{y}} * (1 - \boldsymbol{y}^2)$$

注：* 是按元素乘，· 是向量和矩阵乘

编程部分

3 编程作业报告

3.1 目的

使用多层感知机完成识别英文字符图像（非线性分类）任务，输入一张图像，模型输出识别结果。不区分大小写形式，输出时均转为大写字母。

3.2 环境配置

本次实验中我采用的各 package 版本如下：

```
python==3.8.8 numpy==1.19.5
pytorch==1.9.0 torchvision==0.10.0
```

`opencv==4.0.1 matplotlib==3.5.0`

3.3 完成交叉熵损失函数——losses.py

首先 logits (或者用$\boldsymbol{z}$表示) 经过 soft max 处理变为$\boldsymbol{q}$，然后通过交叉熵计算得到 L。$\boldsymbol{z}$和$\boldsymbol{q}$的形状是[batch_szie, output_size] ，L 是一个标量，$\boldsymbol{Y}$是各样本真值的*onehot*矩阵，不同样本之间没有相互影响。最终$\partial L/\partial \boldsymbol{z}$的形状和$\boldsymbol{z}$相同，所以我们只需要分析每个位置的导数应该是多少。对某一个样本i，其分类真值onehot向量y*的形状是$[class_num,]$，其中第y_i个值是1，其余全是0，因此这条样本的交叉熵是-log $\left(\boldsymbol{q}\left[i, y_{y_i}^*\right]\right)$。
$\boldsymbol{q}\left[i, y_{y_i}^*\right]$与这条样本中所有的$z$有关，但和其它样本中的$z$都无关，所以$L$对第$i$条样本的数据$\boldsymbol{z}\left[i,:\right]$的偏导数都通过$\boldsymbol{q}\left[i, y_{y_i}^*\right]$反映。

$$\because \frac{\partial L}{\partial \boldsymbol{z}[i,j]} = \frac{\partial L}{\partial \boldsymbol{q}\left[i,y_{y_i}^*\right]} \frac{\partial \boldsymbol{q}\left[i,y_{y_i}^*\right]}{\partial \boldsymbol{z}[i,j]} = -\frac{1}{\boldsymbol{q}\left[i,y_{y_i}^*\right]} \frac{\partial \boldsymbol{q}\left[i,y_{y_i}^*\right]}{\partial \boldsymbol{z}[i,j]}$$

$$\because \frac{\partial \boldsymbol{q}\left[i,y_{y_i}^*\right]}{\partial \boldsymbol{z}[i,j]} = \begin{cases} \boldsymbol{q}\left[i, y_{y_i}^*\right]\left(1-\boldsymbol{q}\left[i, y_{y_i}^*\right]\right) = -\boldsymbol{q}\left[i, y_{y_i}^*\right]\left(\boldsymbol{q}\left[i,j\right]-1\right) & j = y_{y_i}^* \\ -\boldsymbol{q}\left[i, y_{y_i}^*\right]\boldsymbol{q}\left[i,j\right] = -\boldsymbol{q}\left[i, y_{y_i}^*\right]\left(\boldsymbol{q}\left[i,j\right]-0\right) & j \neq y_{y_i}^* \end{cases}$$

$$\therefore \frac{\partial L}{\partial \boldsymbol{z}[i,j]} = -\frac{1}{\boldsymbol{q}\left[i,y_{y_i}^*\right]} \frac{\partial \boldsymbol{q}\left[i,y_{y_i}^*\right]}{\partial \boldsymbol{z}[i,j]} = \boldsymbol{q}\left[i,j\right] - y_j^*$$

$$\therefore \frac{\partial L}{\partial \boldsymbol{z}[i,:]} = \boldsymbol{q}\left[i,:\right] - y^*$$

$$\therefore \frac{\partial L}{\partial \boldsymbol{z}} = \boldsymbol{q} - \boldsymbol{Y}^*$$

```python
import torch
import torch.nn.functional as F

#TODO 1: Complete the CrossEntropyLoss function
        # step 1: calculate softmax(z) using stable softmax method
        # hint: you can use torch.exp(x) to calculate exp(x),
        # and remeber to convert label into one-hot version
        # e.g., if label = [0, 2] and n_classes=4,
        # then the one-hot version is [[1,0,0,0], [0,0,1,0]]

        # 1.1: calculate z_max
        z_max = logits.max(dim = 1, keepdim=True)
        # 这里 keepdim=True 是为了便于下一步做差不出现错位
```

```
        # 1.2: calculate exps = exp(z - z_max)
        exps = torch.exp(logits-z_max.values)

        # 1.3: calculate p = softmax(y - y_max)
        partition = exps.sum(dim=1, keepdim=True)
        p = exps/partition

        # step 2: convert label into one-hot version by using F.one_hot()
        # the converted label has shape [batch_size, n_classes]
        label_one_hot = F.one_hot(label, logits.size(1))

        # step 3: calculate cross entropy loss = - log q_i, and averaged by batch
        # save result of softmax and one-hot label in ctx for gradient computation
        loss = -torch.log((p * label_one_hot).sum(1)+1e-9)# 需防止溢出
        loss = torch.mean(loss)# 求 batch 内的平均
        ctx.save_for_backward(p, label_one_hot)
        return loss

……

        # step 4: get p and label from ctx and
        # calculate the derivative of loss w.r.t. pred (dL/dz)
        q, label_one_hot = ctx.saved_tensors

        # dl/dz 的形状和 z (logits) 的形状相同
        # 按照之前的理论推导，只要乘上 (q-Y) 即可
        grad_input = grad_output*(q-label_one_hot)
        # return None for gradient of label since
        # we do not need to compute dL/dlabel
        return grad_input, None
# End TODO 2
```

3.4 完成线性前向函数——network.py

这部分比较简单，直接计算 $y = xW^T + b$ 即可

```
@staticmethod
    def forward(ctx, x, W, b):
        '''
```

```
    Input:
    :param ctx: a context object that can be used to stash information
    for backward computation
    :param x: input features with size [batch_size, input_size]
    :param W: weight matrix with size [output_size, input_size]
    :param b: bias with size [output_size]
    Return:
    y :output features with size [batch_size, output_size]
    '''

    # TODO 1: calculate y = xW^T + b and save results in ctx
    y = torch.matmul(x,W.T)+b
    ctx.save_for_backward(x,W)
    # End TODO 1
    return y
```

3.5 完成线性后向函数——network.py

注意在 2.2 中的 $\left[\left(\frac{\partial L}{\partial \boldsymbol{y}}\right) * \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}}\right)\right]$ 在这里对应的就是 $\frac{\partial L}{\partial \boldsymbol{y}}$，即损失函数对线形层输出 $\boldsymbol{y}$ 的导数，在 2.2 中的 $\boldsymbol{y}$ 是激活层的输出，$\boldsymbol{z}$ 才是线形层的输出。2.2 中的 x 是行向量，这里的 **x** 是矩阵，形状是 [batch_size,input_size]。不过我们可以先分析 **x** 的每一行，按照 2.2 中的结论 $\frac{\partial L}{\partial x}$ 的每一行都是 $\left[\left(\frac{\partial L}{\partial \boldsymbol{y}}\right) * \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}}\right)\right] \cdot \boldsymbol{W}$，其中 $\left[\left(\frac{\partial L}{\partial \boldsymbol{y}}\right) * \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}}\right)\right]$ 也是行向量，而这里的 $\frac{\partial L}{\partial \boldsymbol{y}}$ 是矩阵，它的每一行和 $\boldsymbol{W}$ 相乘，得到的就是对 X 偏导数的每一行，所以应该是 grad_input = torch.matmul(grad_output, W)。

同理，2.2 中 $\frac{\partial L}{\partial W} = \left[\left(\frac{\partial L}{\partial \boldsymbol{y}}\right) * \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{z}}\right)\right]^T \cdot \boldsymbol{x}$，在这里对应的应该是 $\frac{\partial L}{\partial W} = \left[\left(\frac{\partial L}{\partial \boldsymbol{y}}\right)\right]^T \cdot$x。可以这么理解，看 2.2 中 $\frac{\partial L}{\partial W}$ 的某一行，只和某一个线性输出的偏导数有关，然后乘以不同的 x；而这里 $\frac{\partial L}{\partial W}$ 的某一行中的每个元素在前向计算 $y = xW^T + b$ 中都会参与 batch_size 遍，而且每次乘以的 x 对应 x 矩阵中某一列的元素，最终的结果恰好可以同样用 $\left[\left(\frac{\partial L}{\partial \boldsymbol{y}}\right)\right]^T \cdot$x 表示。

同理，$\frac{\partial L}{\partial b}$ 也可以同样地用 $\frac{\partial L}{\partial \boldsymbol{y}}$ 表示，只不过这里 $\frac{\partial L}{\partial \boldsymbol{y}}$ 是矩阵，需要对某一个维度进行塌缩处理，一方面可以根据结果的维度应该是 output_size 可知，应该塌缩行；另一方面，考虑前向传播 $y = xW^T + b$ 的计算过程，b_i 在 y_{ji} 的计算中都有出现，且对应的偏导数都是 1，所以应该把 y_{ji} 都加起来，即塌缩行。

```
@staticmethod
    def backward(ctx, grad_output):
        '''
        Input:
        :param ctx: a context object with saved variables
        :param grad_output: dL/dy, with size [batch_size, output_size]
        Return:
        grad_input: dL/dx, with size [batch_size, input_size]
        grad_W: dL/dW, with size [output_size, input_size],
        summed for data in the batch
        grad_b: dL/db, with size [output_size], summed
        for data in the batch
        '''

        # TODO 2: get x and W from ctx and calculate the gradients
        # by ctx.saved_variables
        x,W = ctx.saved_tensors
        # calculate dL/dx (grad_input) by using dL/dy (grad_output)
        # and W, eg., dL/dx = dL/dy*W
        # calculate dL/dW (grad_W) by using dL/dy (grad_output) and x
        # calculate dL/db (grad_b) using dL/dy (grad_output)
        # you can use torch.matmul(A, B) to compute matrix product of A and B
        grad_input = torch.matmul(grad_output, W)
        grad_W = torch.matmul(grad_output.T, x)
        grad_b = grad_output.sum(0)
        # End TODO 2
        return grad_input, grad_W, grad_b
```

3.6 完成线性层权值的初始化——network.py

```
# TODO 3: initialize weights and bias of the linear layer
# and set W and b trainable parameters
        # hint: you can refer homework 1
        #  (0, 1) 正态分布的 W
        W = torch.randn(output_size, input_size)
        b = torch.zeros(output_size)
        self.W = nn.Parameter(W, requires_grad = True)
```

```
        self.b = nn.Parameter(b, requires_grad = True)
        # End TODO 3
```

3.7 完成 MLP 模型——network.py

```
# TODO 4: Finish MLP with at least 2 layers
        else:
            # step 1: initialize the input layer
            layer = Linear(input_size, hidden_size[0])
            # step 2: append the input layer and the activation layer into layers
            layers.append(layer)
            layers.append(self.act)
            # step 3: construct the hidden layers and add it to layers
            for i in range(1, n_layers - 1):
                #initialize a hidden layer and activation layer
                # hint: Noting that the output size of a hidden layer is
                # hidden_size[i], so what is its input size?
                layer = Linear(hidden_size[i-1],hidden_size[i])
                layers.append(layer)
                layers.append(self.act)

            # step 4: initialize the output layer and append the layer into layers
            # hint: what is the output size of the output layer?
            # hint: here we do not need activation layer
            layer = Linear(hidden_size[-1], output_size)
            layers.append(layer)
            # End TODO 4
```

3.8 完成 MLP 模型和损失函数的定义——recognition.py

```
# TODO 1: initialize the MLP model and loss function
    # what is the input size of the MLP?
    # hint 1: we convert an image to a vector as the input of the MLP,
    # each image has shape [norm_size[0], norm_size[1]]
    # hint 2: Input parameters for MLP: input_size, output_size, hidden_size,
    # n_layers, act_type
    input_size = norm_size[0]*norm_size[1] # 输入数据的形状是 norm 数组拉平后的长度
```

```
    model  = MLP(
        input_size = input_size,
        output_size = n_letters,
        hidden_size=hidden_size,
        n_layers = n_layers,
        act_type=act_type
        )
    # loss function
    myloss = CrossEntropyLoss.apply
    # End TODO 1
```

3.9 计算 loss 和训练网络——recognition.py

```
#TODO 2: calculate losses and train the network using the optimizer
        for step, (ims, labels) in enumerate(trainloader):  # get a batch of data

            # step 1: set data type and device
            ims, labels = ims.to(device), labels.to(device)
            # step 2: convert an image to a vector as the input of the MLP
            input = ims.view(ims.size(0),-1)
            # hint: clear gradients in the optimizer
            optimizer.zero_grad()
            # step 3: run the model which is the forward process
            logits = model(input)
            # step 4: compute the loss, and call backward propagation function
            loss = myloss(logits, labels)
            loss.backward()
            # step 5: sum up of total loss, loss.item() return
            # the value of the tensor as a standard python number
            # this operation is not differentiable
            total_loss += loss.item()
            # step 6: call a function, optimizer.step(),
            # to update the parameters of the model
            optimizer.step()
            # End TODO 2
```

3.10 读取保存的模型————recognition.py

```
# TODO 3: load configurations from saved model, initialize the model.
    # Note: you can complete this section by referring to Part 4: test.
    # load configurations from saved model, initialize and test the model
    # step 1: load configurations from saved model using torch.load(model_path)
    checkpoint = torch.load(model_path)
    # and get the configs dictionary, configs = checkpoint['configs'],
    configs = checkpoint['configs']
    # then get each config from configs, eg., norm_size = configs['norm_size']
    norm_size = configs['norm_size']
    output_size = configs['output_size']
    hidden_size = configs['hidden_size']
    n_layers = configs['n_layers']
    act_type = configs['act_type']
    # step 2: initialize the model by MLP()
    model = MLP(norm_size[0] * norm_size[1], output_size,
                hidden_size, n_layers, act_type)
    # step 3: load model parameters we saved in model_path
    # hint: similar to what we do in Part 4: test.
    model.load_state_dict(checkpoint['state_dict'])
    print('[Info] Load model from {}'.format(model_path))
    # End TODO 3
```

3.11 训练/测试

3.11.1 使用 sgd 优化器，只调整动量因子大小

momentum	Epoch 50 validation accuracy	Test accuracy
0.88	65.0%	69.5%
0.9	67.2%	71.0%
0.92	67.5%	70.8%
0.95	74.8%	78.0%
0.98	62.0%	61.8%

表 1: sgd 优化器准确率

可以看出，动量因子小于 0.95 时，准确率随因子值的增大而增大，并在 0.95 左右达到最好的效果；如果动量因子太大，会出现 loss 局部震荡的现

象（见图 1），而且预测准确率也会下降。准确率见表 1

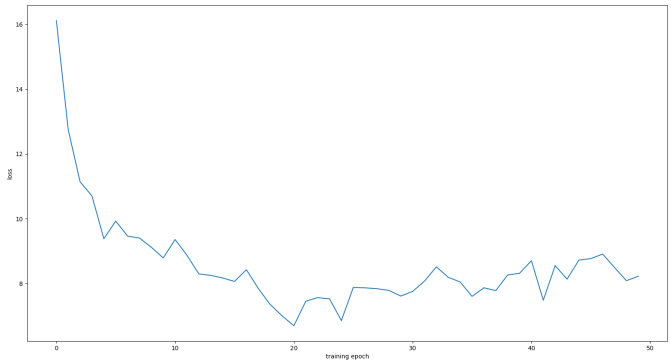

图 1: sgd momentum=0.98

3.11.2 使用 adam 优化器，hsize=64, lr=2e-3, 只调整 weight-decay

adam 优化器下，准确率随 weight-decay 变化的数据见表 2。可以看出，weight-decay 在 0.15 下有最好的预测效果。

weight-decay	Epoch 50 validation accuracy	Test accurancy
0.08	79.2%	82.5%
0.1	80.5%	80.2%
0.12	77.8%	79.8%
0.15	80.8%	84.2%
0.18	80.0%	82.8%

表 2: adam 优化器准确率

3.11.3 测试结果可视化

选择实验中效果最好的模型配置：adam 优化器、hsize=64, lr=2e-3, weight-decay=0.15、lr= 2e-3、momentum=0，test 结果可视化如图 2。

3.12 预测

用 3.11.3 中的模型进行预测，结果如下：

```
(base) E:\2022_1\MR\hw2>python recognition.py --mode predict
--im_path data/character_classification/new_images/predict01.png
```

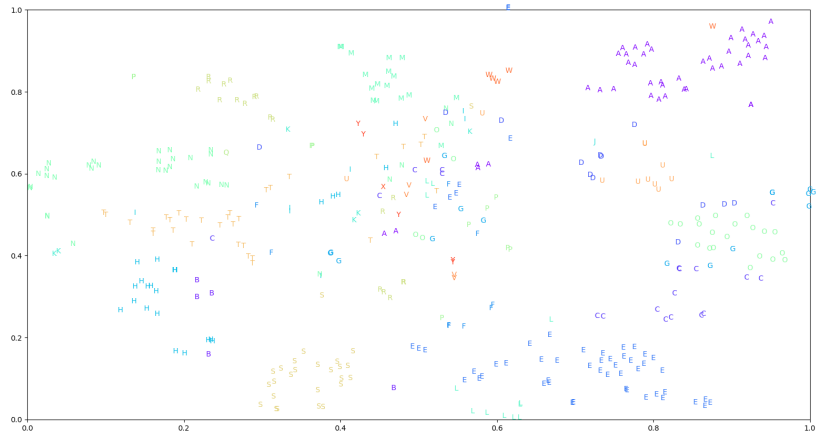

图 2: adam weight-decay=0.15 test 可视化

```
[Info] Load model from saved_models/recognition.pth
Prediction: A

(base) E:\2022_1\MR\hw2>python recognition.py --mode predict
--im_path data/character_classification/new_images/predict02.png
[Info] Load model from saved_models/recognition.pth
Prediction: B
```

4 总结

1. 关于反向传播求导数，我目前的想法是：第一，对谁求导，求完导后结果的形状和谁相同；第二，A 对 B 求导，结果的对应 $[i, j]$ 位置是 A 中每一个元素对 $B[i, j]$ 导数之和，这样就可以把任意形状的求导问题转换为标量对标量的求导。

2. 由于该分类的任务训练数据比较小，在增加隐含层层数时可能会导致性能下降。比如执行”python recognition.py –mode train –momentum 0.95 –hsize 32,32 –layer 3” 和”python recognition.py –mode train –hsize 32,32 –lr 2e-3 –optim_type adam –momentum 0 –weight_decay 0.15 –layer 3” 时，预测准确率有较为明显的下降，甚至可能低于 15%。同时在该任务中 Relu 为较合适的激活函数，在选择其他激活函数时可能会有严重的性能下降。

3. 整个实验中预测准确率还不是很高的主要原因是训练数据数量较少的缘故，当然也有可能通过各参数的调整实现更好的预测精度，不过参数可能的取值组合非常之多，手动尝试的效率太低，因此本质上还是要增加训练数据集的大小以及进一步优化网络模型。

4. 代码部分的反向传播部分的梯度推导直接使用了 2.2 中的结论，因为 latex 画矩阵实在是太麻烦了……这里先偷个懒了 orz