人工神经网络

Lab 3：全连接神经网络(FCN)

数据科学与计算机学院　17大数据与人工智能

17341015　陈鸿峥

目录

1　框架概览　2

2　模块实现　2

2.1　网络层 2

2.2　激活函数 4

2.3　模块 6

2.4　损失函数 7

2.5　优化器 9

3　全连接网络搭建　10

4　实验设置　11

5　实验结果　11

6　总结　12

7　附录：理论推导　12

8　参考资料　15

一、 框架概览

虽然本次作业只是让我们实现一个3层的全连接神经网络，不过为了更加深入理解神经网络框架的原理，在本次实验中我借鉴了PyTorch的API接口，尝试用更好的模块化方式实现一个**简易版的神经网络框架**。由于基础设施（老师提供的模板）是基于PyTorch的，因此我的框架也是基于PyTorch的Tensor类进行封装，命名为TinyTorch。

TinyTorch的整体架构如图1所示，主要分为网络层(Layer)、模块(Module)、损失函数(Loss)、优化器(Optimizer)四个部分。数据通过前端读入后，经过神经网络模块前向传播(forward)，计算损失及其梯度，并反向传播(backward)，输出结果。

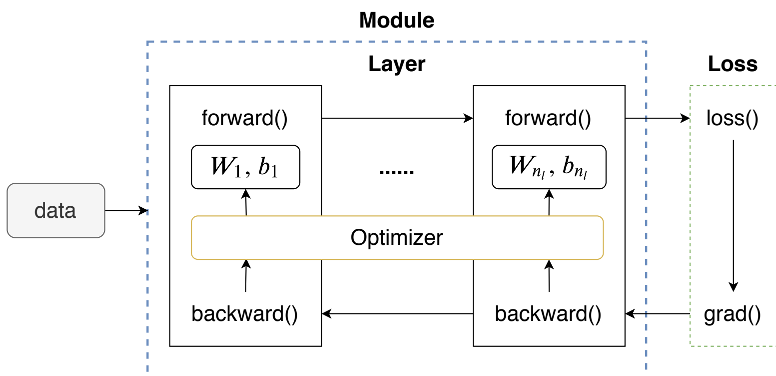

图 1: TinyTorch概览

各模块功能如下：

- 网络层：包含神经网络层（如线性层和卷积层）和激活函数层。
- 模块：将网络层打包封装起来的完整神经网络，继承自基类`nn.Module`，用户可自定义传播方式。
- 损失函数：计算损失函数及其梯度。
- 优化器：用于更新网络层参数。

下面会详细介绍各个模块的实现方式。

二、 模块实现

为打包成可在Python中调用的库，需要在顶层文件夹添加`__init__.py`文件，内容为空即可。

1. 网络层

在`tinytorch/nn.py`中实现，首先实现基类`Layer`，内容如下。作为网络层的抽象，其需要有前向传播和后向传播功能，故分别实现了`forward`和`backward`两个函数。在前向传播中需要将上一层的输出（也即当前层的输入）先保存起来，方便反向传播进行计算。而具体前后向传

播怎么算，则由子类进行定义。同时，为了方便像PyTorch中一样用括号形式直接调用，这里重载了`__call__`函数。

```python
class Layer(object): # Base class
    def __init__(self, name):
        self.name = name
        self.inputs = None
        self.params = None

    def forward(self, inputs):
        self.inputs = inputs.clone().detach() # for backprop
        return self._forward(inputs)

    def backward(self, delta):
        return self._backward(delta)

    def _forward(self, inputs):
        raise NotImplementedError

    def _backward(self, delta):
        raise NotImplementedError

    def parameters(self):
        return self.params

    def __call__(self, inputs):
        return self.forward(inputs)
```

有了基类便可实现线性层`Linear`，其前向传播计算方式如下

$$\mathbf{z}^{[l]} = \text{Linear}(\mathbf{a}^{[l-1]}) = \mathbf{a}^{[l-1]}W + \mathbf{b}$$

其中$W \in \mathbb{R}^{d_in \times d_out}$为权重参数，$\mathbf{b} \in \mathbb{R}^{1 \times d_out}$为偏置参数，$\mathbf{a}^{[k-1]}$为上一层激活后的输出。注意这里将权重参数右乘输入向量，只是为了方便存储和计算，与后面的推导可能存在差异。

反向传播（具体理论推导见第7节）

$$\begin{cases} \dfrac{\partial \mathcal{L}}{\partial W} = {\mathbf{a}^{[l-1]}}^{\mathrm{T}} \delta^{[l]} \\ \dfrac{\partial \mathcal{L}}{\partial \mathbf{b}} = \delta^{[l]} \end{cases}$$

同时一并计算$\tilde{\delta}^{[l-1]} = \delta^{[l]} W^{\mathrm{T}}$。另外注意由于是批训练，故$\partial \mathcal{L}/\partial \mathbf{b}$计算时是对批内所有值求和。

初始化W和$\mathbf{b}$直接调用`torch.nn.init.kaiming_uniform_`，存储在`params`字典中。同时，开设两个元素`d_w`和`d_b`存储梯度。

进而可重载前向和后向传播函数，并得到完整线性层实施如下。

```
class Linear(Layer):
    def __init__(self, in_feat, out_feat,
                 w_init=kaiming_uniform_,
                 b_init=kaiming_uniform_):
        """
        w: (in_feat, out_feat)
        b: (1, out_feat)
        Ref: https://pytorch.org/docs/stable/nn.html#linear
        """
        super().__init__("Linear")

        self.params = {
            "w": w_init(torch.empty([in_feat, out_feat])),
            "b": b_init(torch.empty([1, out_feat])),
            "d_w": torch.zeros([in_feat, out_feat]),
            "d_b": torch.zeros([1, out_feat])
        }
        self.inputs = None

    def _forward(self, inputs):
        """
        inputs: (N, in_feat)
        inputs * w: (N, out_feat)
        b: (1, out_feat) # broadcasting
            inputs * w + b
        """
        return self.inputs @ self.params["w"] + self.params["b"]

    def _backward(self, delta):
        """
        delta_{l+1}: (N, out_feat)
        inputs: (N, in_feat)
        delta_l: (N, in_feat)
        """
        self.params["d_w"] = self.inputs.T @ delta
        self.params["d_b"] = torch.sum(delta, axis=0) # need to sum over batch
        return delta @ self.params["w"].T
```

2. 激活函数

激活函数的基类Activation实现在tinytorch/nn.py中，作为特殊的网络层，依然从Layer继

承得到。由于激活函数层反向传播的模式是固定的，

$$\delta^{[l]} = \tilde{\delta}^{[l]} \odot h'(\mathbf{z}^{[l]})$$

故可以再做一层抽象，新增的激活函数只需重载导函数$h'(\cdot)$即可。基类定义如下。

```
class Activation(Layer): # Base class

    def __init__(self, name):
        super().__init__(name)

    def _forward(self, inputs):
        return self.func(inputs)

    def _backward(self, grad):
        return self.d_func(self.inputs) * grad # element-wise

    def func(self, x):
        raise NotImplementedError

    def d_func(self, x):
        raise NotImplementedError
```

其他激活函数可参见tinytorch/activation.py。由于这一部分就是将激活函数用向量形式表示，并求出其导函数，故在此不再赘述。这里关注Sigmoid和Tanh的导函数

$$\sigma'(\mathbf{x}) = \sigma(\mathbf{x})(1 - \sigma(\mathbf{x}))$$
$$\tanh'(\mathbf{z}) = 1 - \tanh^2(\mathbf{x})$$

完整代码实现如下。

```
class ReLU(Activation):

    def __init__(self):
        super().__init__("ReLU")

    def func(self, x):
        """
        x: (N, out_feat)
        """
        # Since torch has no maximum function,
        # use numpy to implement here
        return np.maximum(x, 0.0) # element-wise

    def d_func(self, x):
```

```python
        """
        x: (N, out_feat)
        dx = 1 if x > 0
             0 if x <= 0
        """
        return x > 0.0

class Sigmoid(Activation):

    def __init__(self):
        super().__init__("Sigmoid")

    def func(self, x):
        return 1.0 / (1.0 + torch.exp(-x))

    def d_func(self, x):
        return self.func(x) * (1.0 - self.func(x))

class Tanh(Activation):

    def __init__(self):
        super().__init__("Tanh")

    def func(self, x):
        return (torch.exp(x) - torch.exp(-x)) / (torch.exp(x) + torch.exp(-x))

    def d_func(self, x):
        return 1.0 - self.func(x) ** 2
```

3. 模块

模块基类Module定义在`tinytorch/nn.py`中，为了最大程度减少用户的代码编写量，因此这里将反向传播部分也实现在`backward`中，但由于没有用重载运算符的方式实现自动微分(automatic differentiation, AD)功能[1]，故还是需要用户将计算出来的损失函数的梯度值输入，然后调用`backward`函数进行反向传播。同时，在初始化阶段，也需要用户调用`add_layers`函数，将网络层**依序**添加到列表中。这样回传时就可以直接逆向遍历网络层，逐一调用网络层的`backward`实现反向传播。代码如下。

```python
class Module(object):
    """NN base class"""
    def __init__(self, name):
        super(Module, self).__init__()
```

[1]这需要自己定义Tensor类，并对运算符进行重载，但由于老师的框架直接调用PyTorch的DataLoader，已经将数据打包成了PyTorch自带的Tensor类，因此没有办法重新进行封装。

```python
        self.name = name
        self.layers = []
        self.params = []

    def forward(self, inputs):
        raise NotImplementedError

    def backward(self, delta):
        grad = delta
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

    def parameters(self):
        return self.params

    def add_layers(self,layers):
        self.layers = layers
        for layer in layers:
            if not isinstance(layer,Activation):
                self.params.append(layer.parameters())

    def __call__(self, inputs):
        return self.forward(inputs)
```

4. 损失函数

损失函数定义在`tinytorch/loss.py`中，同样有基类`Loss`。派生类需要重载前向计算的`loss`函数，以及反向传播的`grad`函数。这一部分还是因为没有实现AD，故没有办法像PyTorch一样做到`loss.backward()`回传，只能分开实现计算梯度后再回传。

下面以交叉熵函数为例，前向传播公式为（已经结合了Softmax和NLLLoss）

$$\mathcal{L}=\frac{1}{N}\sum_{i=1}^{N}\mathrm{Loss}(\mathbf{x}^{(i)},y^{(i)})=-\frac{1}{N}\sum_{i=1}^{N}\log\frac{\exp(\mathbf{x}^{(i)}[y^{(i)}])}{\sum_{j=1}^{d}\exp(\mathbf{x}^{(i)}{}_j)}=-\frac{1}{N}\sum_{i=1}^{N}\left(\mathbf{x}^{(i)}[y^{(i)}]-\log\left(\sum_{j=1}^{d}\exp(\mathbf{x}^{(i)}{}_j)\right)\right) \tag{1}$$

这一部分要特别小心，非常容易出错。在具体实施上需要关注以下两个点：

- **避免指数爆炸**：直接计算Softmax会导致指数过大，因此需要对分子分母同除exp(max)，相当于指数作差，确保指数都小于等于0，进而指数项落在$(0,1]$的区间
- **避免对数爆炸**：由于计算机精度原因，一些过小的值会被直接当作0处理（数值下溢），若直接取对数会导致出现`-inf`，故应该把Softmax计算和NLLLoss计算合并，变成(1)后面的公式，对指数项求和后可确保值不再为0，进而可以正常取对数。

反向传播的公式为（推导见第7节）

$$\tilde{\delta}^{[n_l]} = \mathbf{a}^{[n_l]} - \mathbf{y}$$

最终代码实现如下。

```python
class CrossEntropyLoss(Loss):
    """
    Softmax + NLLLoss
    Ref: https://pytorch.org/docs/stable/nn.html#torch.nn.CrossEntropyLoss
    """
    def __init__(self):
        self.probs = None # used for backprop

    def loss(self, pred, target):
        """
        pred: (N, class)
        target: (N, )

        loss(x,class) = -x[class] + log(\sum_j exp(x[j]))

        Ref: https://docs.scipy.org/doc/numpy/user/basics.indexing.html#indexing-multi-dimensional-arrays
        """
        n = pred.shape[0] # batch_size
        shifted_pred = pred - torch.max(pred, axis=1, keepdims=True).values # avoid explosion
        exp = torch.exp(shifted_pred)
        log_probs = shifted_pred - torch.log(torch.sum(exp, axis=1, keepdims=True)) # avoid log 0
        self.probs = torch.exp(log_probs) # stored for backprop
        return -torch.sum(log_probs[torch.arange(n),target]) / n

    def grad(self, pred, target):
        """
        pred: (N, class)
        target: (N, )
            delta = pred - target
        """
        n = pred.shape[0]
        self.probs[torch.arange(n), target] -= 1 # one-hot encoding
        return self.probs / n
```

5. 优化器

优化器定义在`tinytorch/optim.py`中，基类`Optimizer`给出梯度归零的初始化方法`zero_grad`，需要在每轮迭代开始前调用。

带动量的SGD更新公式如下。

$$\begin{cases} \boldsymbol{\nu}_{t+1} = \gamma \boldsymbol{\nu}_t - \eta \nabla \mathcal{L}(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{\nu}_{t+1} \end{cases}$$

注意需要为每个参数单独创建$\boldsymbol{\nu}$值，且初始化为$\mathbf{0}$。

```
class Optimizer(object): # Base class
    def __init__(self, params, lr):
        """
        params: All network layer parameters (in a list),
                needed to be added before training,
                stored by references/objects, thus can be modified later
                For linear layer, there are w, b, d_w, d_b four params.
        lr: learning rate
        """
        self.params = params
        self.lr = lr

    def zero_grad(self):
        """
        Needed to be call before each epoch
        """
        for param in self.params:
            param["d_w"] = torch.zeros(param["w"].shape)
            param["d_b"] = torch.zeros(param["b"].shape)

    def step(self):
        raise NotImplementedError

class SGD(Optimizer):
    def __init__(self, params, lr=0.001, momentum=0.9):
        super().__init__(params, lr)
        self.momentum = momentum

    def step(self):
        """
        SGD with momentum, need to store v for each param

        v_{t+1}=\gamma v_t - \eta\nabla L(\theta_t)
```

```
        \theta_{t+1} = \theta_t + v_{t+1}

        Ref: https://d2l.ai/chapter_optimization/momentum.html
        """
        for param in reversed(self.params):
            for item in ["w", "b"]:
                if param.get("v_{}".format(item),None) == None:
                    param["v_{}".format(item)] = torch.zeros(param[item].shape)
                param["v_{}".format(item)] = self.momentum * param["v_{}".format(
                    ↪ item)] - self.lr * param["d_{}".format(item)]
                param[item] += param["v_{}".format(item)]
```

三、 全连接网络搭建

虽然在TinyTorch中调用了PyTorch中一些基本计算函数，如`sum`、`max`等，但其实可以全部换成NumPy实现，因为PyTorch与TinyTorch并不是紧耦合关系。

三层神经网络定义在`fcn.py`中，全部采用全连接层，中间激活函数用ReLU，维度变换如下

$$INPUT(784) \to FC(256) \to FC(128) \to FC(10)$$

```
class Net(nn.Module):
    def __init__(self):
        super().__init__("FCN")
        self.fcn1 = nn.Linear(28*28,256)
        self.fcn2 = nn.Linear(256,128)
        self.fcn3 = nn.Linear(128,10)
        self.relu1 = ReLU()
        self.relu2 = ReLU()
        self.add_layers([self.fcn1,
                         self.relu1,
                         self.fcn2,
                         self.relu2,
                         self.fcn3])

    def forward(self,x):
        x = self.fcn1(x)
        x = self.relu1(x)
        x = self.fcn2(x)
        x = self.relu2(x)
        x = self.fcn3(x)
        return x
```

采用交叉熵损失函数及带冲量的SGD进行优化。具体训练使用接口基本与PyTorch一致，唯一不同是在反向传播部分，TinyTorch需要用下面的方式进行调用。

```
net.backward(criterion.grad(output,label))
```

其他部分按照老师所给的框架直接执行即可。

四、 实验设置

实验MNIST数据集上进行测试，并将TinyTorch与PyTorch进行比较。完整代码可见`fcn.py`和`fcn_torch.py`，后者是PyTorch实现的神经网络，采用的网络结构、优化器及损失函数均与TinyTorch相同。

实验在CPU上进行，超参数设置如表1所示。

表 1: 超参数

学习率	`lr`	0.02
动量	`momentum`	0.9
批量大小	`batch_size`	128
迭代次数	`epoch`	20

五、 实验结果

实验结果如图2所示，TinyTorch与PyTorch在测试集的精度最终都达到100%，而测试集精度都达到96%以上，可见TinyTorch已经非常接近PyTorch的水平，而存在的一些差异则是在网络层的初始化、权重衰减、以及优化器的实现上面。

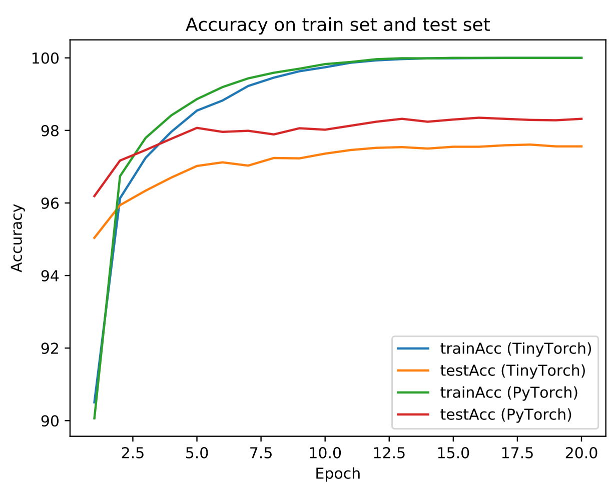

图 2: 训练集及测试集精度

图3展示了两个框架的训练损失，可以看到两者的训练损失都在快速下降，证明梯度下降的正确性。

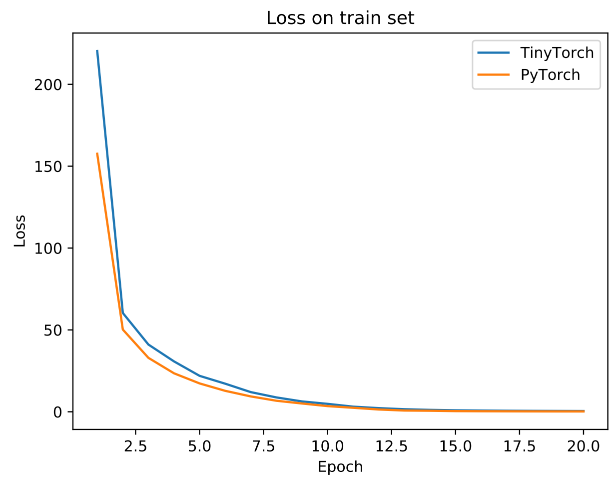

图 3: 训练Loss

六、 总结

本次实验自己搭建了TinyTorch的神经网络框架，虽然只有大约200行代码，且没有将自动微分功能实现，但还是熟悉了深度学习框架的整体设计及搭建流程。最终可以用自己写的框架搭建起三层神经网络，并且做出来的结果也与PyTorch近似，基本算是达成目标。后续如果有时间的话，则尝试将内置的Tensor类替换掉，然后实现自己的AD系统，真正搞懂深度学习框架的内部构造，而不再将其当成一个黑箱使用。

七、 附录：理论推导

设$\mathbf{x}^{(i)}$代表第i个训练样本，$\mathbf{z}^{[l]}$代表第l层的输出，$\mathbf{a}^{[l]}$代表激活后的输出。有三层神经网络的前向传播

$$
\begin{aligned}
\mathbf{z}^{[1]} &= W^{[1]}\mathbf{x}^{(i)} + \mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= h(\mathbf{z}^{[1]}) \\
\mathbf{z}^{[2]} &= W^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} \\
\mathbf{a}^{[2]} &= h(\mathbf{z}^{[2]}) \\
\mathbf{z}^{[3]} &= W^{[3]}\mathbf{a}^{[2]} + \mathbf{b}^{[3]} \\
\hat{\mathbf{y}}^{(i)} &= \mathbf{a}^{[3]} = g(\mathbf{z}^{[3]})
\end{aligned}
$$

其中真实结果$\mathbf{y}^{(i)}$为独热码(one-hot encoding)。

考虑损失函数(loss)为交叉熵

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = -\mathbf{y}^{\mathrm{T}} \ln \hat{\mathbf{y}}^{(i)}$$

且激活函数$h(\cdot)$为ReLU函数，$g(\cdot)$为softmax函数。

接下来推导反向传播(backpropagation, BP)算法，利用链式法则求导数。比如想得到隐含层的权重，则计算

$$\frac{\partial \mathcal{L}}{\partial W^{[3]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[3]}} \frac{\partial \mathbf{a}^{[3]}}{\partial \mathbf{z}^{[3]}} \frac{\partial \mathbf{z}^{[3]}}{\partial W^{[3]}}$$

注意上式并非良定，其中涉及到实值函数对向量求导，也涉及到向量对向量求导（雅可比矩阵），还有向量对矩阵求导。

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[3]}} &= \frac{\partial}{\partial \mathbf{a}^{[3]}}(-\mathbf{y}^{\mathrm{T}} \ln \hat{\mathbf{a}}^{[3]}) \qquad \text{实数对向量求导} \\
&= -\frac{\mathbf{y}}{\mathbf{a}^{[3]}} \qquad \text{逐元素相除}
\end{aligned}$$

设$\mathbf{u} = \exp(\mathbf{z}^{[3]})$为逐元素指数，分两步计算

$$\begin{aligned}
\frac{\partial \mathbf{a}^{[3]}}{\partial \mathbf{u}} &= \frac{\partial}{\partial \mathbf{u}} \operatorname{softmax}(\mathbf{z}^{[3]}) \\
&= \frac{\partial}{\partial \mathbf{u}} \frac{\exp(\mathbf{z}^{[3]})}{\mathbf{1}^{\mathrm{T}} \exp(\mathbf{z}^{[3]})} \\
&= \frac{\partial}{\partial \mathbf{u}} \frac{\mathbf{u}}{\mathbf{1}^{\mathrm{T}} \mathbf{u}} \\
&= \mathbf{u} \frac{\partial (1/\mathbf{1}^{\mathrm{T}} \mathbf{u})}{\partial \mathbf{u}^{\mathrm{T}}} + \frac{1}{\mathbf{1}^{\mathrm{T}} \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{u}} \qquad \text{乘法法则} \\
&= -\frac{1}{(\mathbf{1}^{\mathrm{T}} \mathbf{u})^3} \mathbf{u} \mathbf{1}^{\mathrm{T}} + \frac{1}{\mathbf{1}^{\mathrm{T}} \mathbf{u}} I \\
\frac{\partial \mathbf{u}}{\partial \mathbf{z}^{[3]}} &= \frac{\partial \exp(\mathbf{z}^{[3]})}{\partial \mathbf{z}^{[3]}} \\
&= \operatorname{diag}(\exp(\mathbf{z}^{[3]})) \\
&= \operatorname{diag}(\mathbf{u})
\end{aligned}$$

由雅可比矩阵的链式法则

$$\begin{aligned}
\frac{\partial \mathbf{a}^{[3]}}{\partial \mathbf{z}^{[3]}} &= \frac{\partial \mathbf{a}^{[3]}}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{z}^{[3]}} \\
&= \left(-\frac{1}{(\mathbf{1}^{\mathrm{T}} \mathbf{u})^3} \mathbf{u} \mathbf{1}^{\mathrm{T}} + \frac{1}{\mathbf{1}^{\mathrm{T}} \mathbf{u}} I\right) \operatorname{diag}(\mathbf{u}) \\
&= -\frac{1}{(\mathbf{1}^{\mathrm{T}} \mathbf{u})^3} \mathbf{u} \mathbf{u}^{\mathrm{T}} + \frac{1}{\mathbf{1}^{\mathrm{T}} \mathbf{u}} \operatorname{diag}(\mathbf{u}) \\
&= -\frac{\mathbf{u}}{\mathbf{1}^{\mathrm{T}} \mathbf{u}} \left(\frac{\mathbf{u}}{\mathbf{1}^{\mathrm{T}} \mathbf{u}}\right)^{\mathrm{T}} + \frac{1}{\mathbf{1}^{\mathrm{T}} \mathbf{u}} \operatorname{diag}(\mathbf{u}) \\
&= -\mathbf{a}^{[3]} {\mathbf{a}^{[3]}}^{\mathrm{T}} + \operatorname{diag}(\mathbf{a}^{[3]})
\end{aligned}$$

进而

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[3]}} &:= \delta^{[3]} \qquad \text{损失函数对未激活前}\mathbf{z}\text{的导数} \\
&= \left(\frac{\partial \mathbf{a}^{[3]}}{\partial \mathbf{z}^{[3]}}\right)^{\mathrm{T}} \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[3]}} \\
&= -\left(-\mathbf{a}^{[3]}\mathbf{a}^{[3]\mathrm{T}} + \mathrm{diag}(\mathbf{a}^{[3]})\right)^{\mathrm{T}} \left(\frac{\mathbf{y}}{\mathbf{a}^{[3]}}\right) \\
&= \mathbf{a}^{[3]}\left(\mathbf{a}^{[3]\mathrm{T}}\frac{\mathbf{y}}{\mathbf{a}^{[3]}}\right) - \mathrm{diag}(\mathbf{a}^{[3]})\left(\frac{\mathbf{y}}{\mathbf{a}^{[3]}}\right) \\
&= \mathbf{a}^{[3]}\mathbf{1}^{\mathrm{T}}\mathbf{y} - \mathbf{y} \\
&= \mathbf{a}^{[3]} - \mathbf{y} \qquad \text{独热码满足}\mathbf{1}^{\mathrm{T}}\mathbf{y} = 1
\end{aligned}
$$

之后可计算

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W^{[3]}} &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[3]}}\frac{\partial \mathbf{z}^{[3]}}{\partial W^{[3]}} &&= \delta^{[3]}\mathbf{a}^{[2]\mathrm{T}} \qquad \text{线性变换求导法则} \\
\frac{\partial \mathcal{L}}{\partial b^{[3]}} &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[3]}}\frac{\partial \mathbf{z}^{[3]}}{\partial b^{[3]}} &&= \delta^{[3]}
\end{aligned}
$$

再往前推一层

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[2]}} &= W^{[3]\mathrm{T}}\delta^{[3]} \\
\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{[2]}} &:= \delta^{[2]} \\
&= \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[2]}}\frac{\partial \mathbf{a}^{[2]}}{\partial \mathbf{z}^{[2]}} \\
&= \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{[2]}} \odot h'(\mathbf{z}^{[2]}) \\
&= \left(W^{[3]\mathrm{T}}\delta^{[3]}\right) \odot h(\mathbf{z}^{[2]}) \odot (\mathbf{1} - h(\mathbf{z}^{[2]})) \\
\frac{\partial \mathcal{L}}{\partial W^{[2]}} &= \delta^{[2]}\mathbf{x}^{\mathrm{T}} \\
\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[2]}} &= \delta^{[2]} \\
\frac{\partial \mathcal{L}}{\partial \mathbf{x}} &= W^{[2]\mathrm{T}}\delta^{[2]}
\end{aligned}
$$

总结来说，对于第n_l层，

$$
\delta^{[n_l]} = -(\mathbf{y} - \mathbf{a}^{[n_l]}) \odot g'(\mathbf{z}^{[n_l]}) = \tilde{\delta}^{[n_l]} \odot g'(\mathbf{z}^{[n_l]})
$$

对于第$l = n_l - 1, n_l - 2, \ldots, 1$层，

$$
\delta^{[l]} = \left(W^{[l+1]\mathrm{T}}\delta^{[l+1]}\right) \odot h'(\mathbf{z}^{[l]}) = \tilde{\delta}^{[l]} \odot h'(\mathbf{z}^{[l]})
$$

则有权重和偏置的梯度

$$\nabla_{W^{[l]}}\mathcal{L}(W,b) = \delta^{[l]}\mathbf{a}^{[l-1]^{\mathrm{T}}}$$

$$\nabla_{b^{[l]}}\mathcal{L}(W,b) = \delta^{[l]}$$

八、 参考资料

- UFLDL Tutorial, Multi-Layer Neural Network
- Andrew Ng, Kian Katanforoosh, Anand Avati, *Stanford CS 229 Lecture Notes: Deep Learning*
- Andrew Ng, Kian Katanforoosh, *Stanford CS 229 Lecture Notes: Backpropagation*
- Kaare Brandt Petersen, Michael Syskind Pedersen, *Matrix Cookbook*
- 矩阵求导术 - 长躯鬼侠的文章 - 知乎
- Daiwk，机器学习中的矩阵、向量求导
- Guibo Wang, *Build a Deep Learning Framework From Scratch*
- Tianqi Chen, UW CSE 599G1: Deep Learning System Course - TinyFlow