



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。

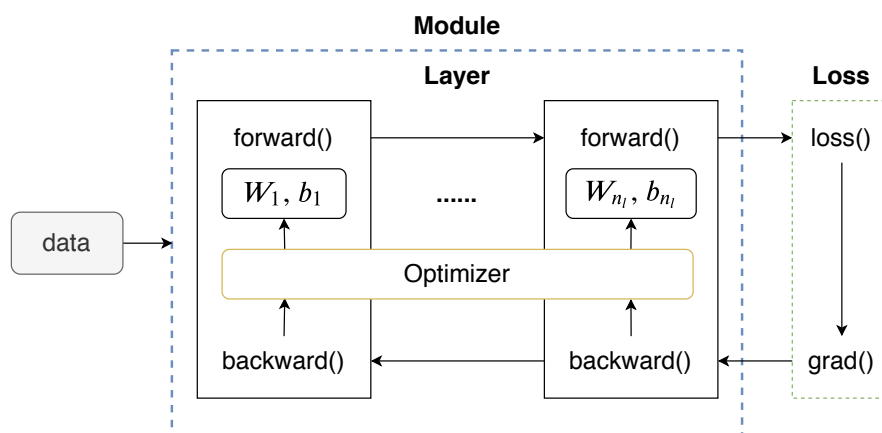


图 1: TinyTorch概览

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。

```

4 class Layer(object): # Base class
5     def __init__(self, name):
6         self.name = name
7         self.inputs = None
8         self.params = None
9
10    def forward(self, inputs):
11        self.inputs = inputs.clone().detach() # for backprop
12        return self._forward(inputs)
13
14    def backward(self, delta):
15        return self._backward(delta)
16
17    def _forward(self, inputs):
18        raise NotImplementedError
19
20    def _backward(self, delta):
21        raise NotImplementedError
22
23    def parameters(self):
24        return self.params
25
26    def __call__(self, inputs):
27        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

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

其中 $W^{[l]} \in \mathbb{R}^{d_{in} \times d_{out}}$ 为权重参数， $\mathbf{b}^{[l]} \in \mathbb{R}^{1 \times d_{out}}$ 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）

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

同时一并计算 $\tilde{\delta}^{[l-1]} = \delta^{[l]}W^{[l]\top}$ 。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 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。

```

46 class Linear(Layer):
47     def __init__(self, in_feat, out_feat,
48                 w_init=kaiming_uniform_,
49                 b_init=kaiming_uniform_):
50         """
51         w: (in_feat, out_feat)
52         b: (1, out_feat)
53         Ref: https://pytorch.org/docs/stable/nn.html#linear
54         """
55         super().__init__("Linear")
56
57         self.params = {
58             "w": w_init(torch.empty([in_feat, out_feat])),
59             "b": b_init(torch.empty([1, out_feat])),
60             "d_w": torch.zeros([in_feat, out_feat]),
61             "d_b": torch.zeros([1, out_feat])
62         }
63         self.inputs = None
64
65     def _forward(self, inputs):
66         """
67         inputs: (N, in_feat)
68         inputs * w: (N, out_feat)
69         b: (1, out_feat) # broadcasting
70         inputs * w + b
71         """
72         return self.inputs @ self.params["w"] + self.params["b"]
73
74     def _backward(self, delta):
75         """
76         delta_{l+1}: (N, out_feat)
77         inputs: (N, in_feat)
78         delta_l: (N, in_feat)
79         """
80         self.params["d_w"] = self.inputs.T @ delta
81         self.params["d_b"] = torch.sum(delta, axis=0) # need to sum over batch
82         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。

```

29 class Activation(Layer): # Base class
30
31     def __init__(self, name):
32         super().__init__(name)
33
34     def _forward(self, inputs):
35         return self.func(inputs)
36
37     def _backward(self, grad):
38         return self.d_func(self.inputs) * grad # element-wise
39
40     def func(self, x):
41         raise NotImplementedError
42
43     def d_func(self, x):
44         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。

```

5 class ReLU(Activation):
6
7     def __init__(self):
8         super().__init__("ReLU")
9
10    def func(self, x):
11        """
12        x: (N, out_feat)
13        """
14        # Since torch has no maximum function,
15        # use numpy to implement here
16        return np.maximum(x, 0.0) # element-wise
17
18    def d_func(self, x):

```

```

19     """
20     x: (N, out_feat)
21     dx = 1 if x > 0
22         0 if x <= 0
23     """
24     return x > 0.0
25
26 class Sigmoid(Activation):
27
28     def __init__(self):
29         super().__init__("Sigmoid")
30
31     def func(self, x):
32         return 1.0 / (1.0 + torch.exp(-x))
33
34     def d_func(self, x):
35         return self.func(x) * (1.0 - self.func(x))
36
37 class Tanh(Activation):
38
39     def __init__(self):
40         super().__init__("Tanh")
41
42     def func(self, x):
43         return (torch.exp(x) - torch.exp(-x)) / (torch.exp(x) + torch.exp(-x))
44
45     def d_func(self, x):
46         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。

```

84 class Module(object):
85     """NN base class"""
86     def __init__(self, name):
87         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。

```

88     self.name = name
89     self.layers = []
90     self.params = []
91
92     def forward(self, inputs):
93         raise NotImplementedError
94
95     def backward(self, delta):
96         grad = delta
97         for layer in reversed(self.layers):
98             grad = layer.backward(grad)
99
100    def parameters(self):
101        return self.params
102
103    def add_layers(self, layers):
104        self.layers = layers
105        for layer in layers:
106            if not isinstance(layer, Activation):
107                self.params.append(layer.parameters())
108
109    def __call__(self, inputs):
110        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）

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N \text{Loss}(\mathbf{x}^{(i)}, y^{(i)}) \quad (1)$$

$$= -\frac{1}{N} \sum_{i=1}^N \log \frac{\exp(\mathbf{x}^{(i)}[y^{(i)}])}{\sum_{j=1}^d \exp(\mathbf{x}^{(i)}_j)} \quad (2)$$

$$= -\frac{1}{N} \sum_{i=1}^N \left(\mathbf{x}^{(i)}[y^{(i)}] - \log \left(\sum_{j=1}^d \exp(\mathbf{x}^{(i)}_j) \right) \right) \quad (3)$$

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 $-\infty$ ，故应该把Softmax计算和NLLLoss计算合并，变成公式(3)，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。

```

14 class CrossEntropyLoss(Loss):
15     """
16     Softmax + NLLLoss
17     Ref: https://pytorch.org/docs/stable/nn.html#torch.nn.CrossEntropyLoss
18     """
19     def __init__(self):
20         self.probs = None # used for backprop
21
22     def loss(self, pred, target):
23         """
24         pred: (N, class)
25         target: (N, )
26
27         loss(x,class) = -x[class] + log(\sum_j exp(x[j]))
28
29         Ref: https://docs.scipy.org/doc/numpy/user/basics.indexing.html#indexing-
30             ↪ multi-dimensional-arrays
31         """
32         n = pred.shape[0] # batch_size
33         shifted_pred = pred - torch.max(pred, axis=1, keepdims=True).values # avoid
34             ↪ explosion
35         exp = torch.exp(shifted_pred)
36         log_probs = shifted_pred - torch.log(torch.sum(exp, axis=1, keepdims=True))
37             ↪ # avoid log 0
38         self.probs = torch.exp(log_probs) # stored for backprop
39         return -torch.sum(log_probs[torch.arange(n),target]) / n
40
41     def grad(self, pred, target):
42         """
43         pred: (N, class)
44         target: (N, )
45         delta = pred - target
46         """
47         n = pred.shape[0]
48         self.probs[torch.arange(n), target] -= 1 # one-hot encoding

```



```
46         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。

$$\begin{cases} \nu_{t+1} = \gamma \nu_t - \eta \nabla \mathcal{L}(\theta_t) \\ \theta_{t+1} = \theta_t + \nu_{t+1} \end{cases}$$

注意需要为每个参数单独创建 ν 值，且初始化为0。

```
3 class Optimizer(object): # Base class
4     def __init__(self, params, lr):
5         """
6         params: All network layer parameters (in a list),
7                 needed to be added before training,
8                 stored by references/objects, thus can be modified later
9                 For linear layer, there are w, b, d_w, d_b four params.
10        lr: learning rate
11        """
12        self.params = params
13        self.lr = lr
14
15    def zero_grad(self):
16        """
17        Needed to be call before each epoch
18        """
19        for param in self.params:
20            param["d_w"] = torch.zeros(param["w"].shape)
21            param["d_b"] = torch.zeros(param["b"].shape)
22
23    def step(self):
24        raise NotImplementedError
25
26    class SGD(Optimizer):
27        def __init__(self, params, lr=0.001, momentum=0.9):
28            super().__init__(params, lr)
29            self.momentum = momentum
30
31        def step(self):
32            """
33            SGD with momentum, need to store v for each param
```

```

34
35     v_{t+1}=\gamma v_t - \eta \nabla L(\theta_t)
36     \theta_{t+1} = \theta_t + v_{t+1}
37
38     Ref: https://d2l.ai/chapter\_optimization/momentum.html
39     """
40     for param in reversed(self.params):
41         for item in ["w", "b"]:
42             if param.get("v_{}".format(item), None) == None:
43                 param["v_{}".format(item)] = torch.zeros(param[item].shape)
44                 param["v_{}".format(item)] = self.momentum * param["v_{}".format(
45                     ↪ item)] - self.lr * param["d_{}".format(item)]
46                 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fcnn.py中，全部采用全连接层，中间激活函数用ReLU，维度变换如下

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

```

33 class Net(nn.Module):
34     def __init__(self):
35         super().__init__("FCN")
36         self.fcn1 = nn.Linear(28*28,256)
37         self.fcn2 = nn.Linear(256,128)
38         self.fcn3 = nn.Linear(128,10)
39         self.relu1 = ReLU()
40         self.relu2 = ReLU()
41         self.add_layers([self.fcn1,
42                         self.relu1,
43                         self.fcn2,
44                         self.relu2,
45                         self.fcn3])
46
47     def forward(self,x):
48         x = self.fcn1(x)
49         x = self.relu1(x)
50         x = self.fcn2(x)
51         x = self.relu2(x)
52         x = self.fcn3(x)
53         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。

160

```
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。

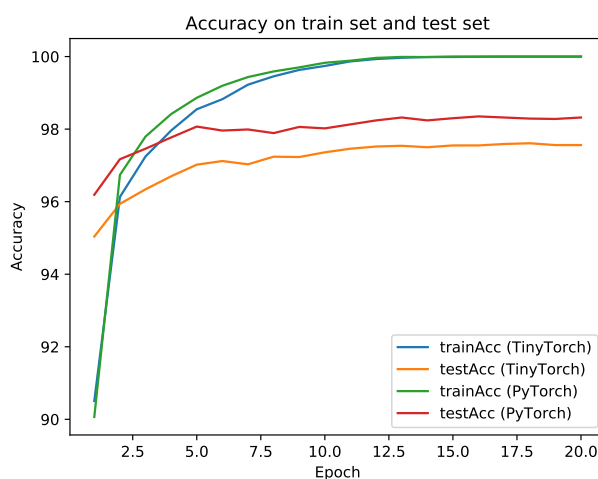


图 2: 训练集及测试集精度

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。

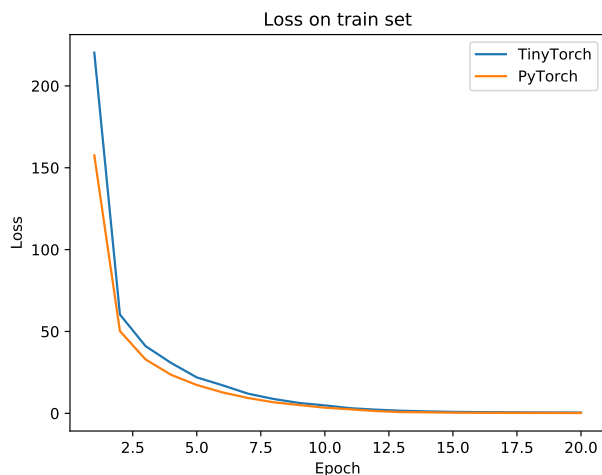


图 3: 训练Loss

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

$$\mathbf{z}^{[1]} = W^{[1]}\mathbf{x}^{(i)} + \mathbf{b}^{[1]}$$

$$\mathbf{a}^{[1]} = h(\mathbf{z}^{[1]})$$

$$\mathbf{z}^{[2]} = W^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]}$$

$$\mathbf{a}^{[2]} = h(\mathbf{z}^{[2]})$$

$$\mathbf{z}^{[3]} = W^{[3]}\mathbf{a}^{[2]} + \mathbf{b}^{[3]}$$

$$\hat{\mathbf{y}}^{(i)} = \mathbf{a}^{[3]} = g(\mathbf{z}^{[3]})$$

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

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

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。

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

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

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

由雅可比矩阵的链式法则

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

进而

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

之后可计算

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

再往前推一层

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

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 $l = n_l - 1, n_l - 2, \dots, 1$ 层，

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

则有权重和偏置的梯度

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

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

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