神经网络教程

2021 年 1 月 22 日

1 复合函数

在高中的时候我们都学过函数，以及函数的复合运算：对于函数 $f : A \to B$，$g : C \to D$，其中 A, B, C, D 都是非空集合，若 $B \subset C$，那么 f 与 g 的函数复合 $g \circ f$ 存在，并且 $g \circ f : A \to D$，所谓 $g \circ f$ 的定义是

$$\begin{aligned} g \circ f : A &\longrightarrow D \\ x &\longmapsto g(f(x)). \end{aligned} \tag{1.1}$$

神经网络，就是一个写在纸上很占地方的很复杂的复合函数，虽然用来复合的那些「基础」函数每一个都很简单，但是经过反复次复合之后，也能产生令人惊讶的效果.

2 感知机

设

$$\begin{aligned} g : \mathbb{R} &\longrightarrow \{-1, 0, 1\} \\ x &\longmapsto \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ 1, & x > 0. \end{cases} \end{aligned} \tag{2.1}$$

这个 g 也被叫做符号函数 (sign function)，也就是说它的作用之一是用来判断任何一个实数 x 的符号的，比如说随便从 $\mathbb{R}$ 中取出一个数 x，把 x 扔进 g，然后看 g 吐出来什么，如果 g 返回了一个 -1 就说明 x 带个负号.

简单起见我们考虑一个拿 3 个数作为输入的函数，或者也把它叫做「向量函数」，因为它的输入是一个三维向量（而一个三维向量可看做一个空间坐标，所以说函数输入有 3 个数）：

$$\begin{aligned} f_{\boldsymbol{w}} : \mathbb{R}^3 &\longrightarrow \mathbb{R} \\ \boldsymbol{x} \overset{\text{def}}{=} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} &\longmapsto \boldsymbol{\omega} \cdot \boldsymbol{x} = \omega_0 x_0 + \omega_1 x_1 + \omega_2 x_2 \end{aligned} \tag{2.2}$$

公式的意思是说 $\boldsymbol{x}$ 定义为一个三维列向量，然后被映为行向量 ω 与 $\boldsymbol{x}$ 的向量内积，这里 $\boldsymbol{\omega}$ 我们把它称作「参数」，后边我们会看到，一个神经网络（确定了结构之后）就是由这些参数定义.

再定义一个输入层函数:

$$
\begin{aligned}
p : \mathbb{R}^2 &\longrightarrow \mathbb{R}^3 \\
\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} &\longmapsto \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}
\end{aligned}
\tag{2.3}
$$

设 $\mathcal{X}$ 是输入空间，它有两个维度，对任意一个 $\boldsymbol{x} = (x_1, x_2)^\mathsf{T} \in \mathcal{X}$，我们发现，表达式 $g(f(p(\boldsymbol{x})))$ 突然变得有意义了，换句话说，我们可以把 $\mathcal{X}$ 到 $\{-1, 0, 1\}$ 的映射用复合函数 $g \circ f \circ p$ 来表示（函数的复合运算满足结合律，所以先复合哪两个没关系），因为我们一开始说了，神经网络无非就是复杂点的复合函数，所以经过合理地定义这三个函数，我们得到了一个神经网络，而这种最为简单的神经网络，也叫做感知机(perceptron).

感知机 $g \circ f \circ p$ 能做什么呢? 按照函数的复合运算法则，我们知道它是 $\mathcal{X} \to \{-1, 0, 1\}$，我们明白了: 通过正确地设定参数向量（也叫权向量）$\boldsymbol{\omega}$，我们能影响 $\mathcal{X} \to \{-1, 0, 1\}$ 的具体对应法则，换句话说，我们能够用它来对数据做分类!

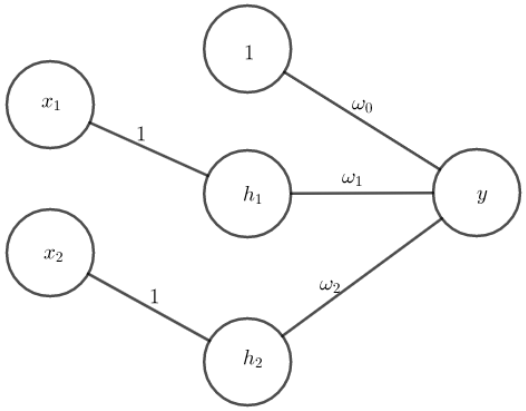

图 2-1: 感知机图示

图 2-1就是 $g \circ f \circ p$ 画出来的样子，p 可以看做是第一层（也就是最左边的那一层），f 看做是第二层，g 看做是第三层. 也可以直观地说: 神经网络的这种连接图就好像是函数的复合路径图. 我们这里画出来的是三层的是因为我们把「添加哑节点」这个操作都用了专门的一层（第一层）来表现，所谓哑节点其实就是第二层最上边那个值恒为 1 的点（我们在圆圈里面标了一个 1），但其实这是一样的，因为 x_1 到 h_1 的连接权是 1，同理 $x_2 = h_2$，所以这个网络相当于是两层的，画三层是为了和复合函数看起来接近.

它的计算过程是这样: 拿到了一个输入 $\boldsymbol{x}$, 它来自二维输入空间 $\boldsymbol{x} \in \mathcal{X} \subset \mathbb{R}^2$, 首先 p 把 $\boldsymbol{x} = (x_1, x_2)^\mathsf{T}$ 映为 $(1, h_1, h_2)^\mathsf{T}$，其中 $h_1 = x_1, h_2 = x_2$（看那两条连接线的权重），这时我们来到了第二层，第二层拿到 $(1, h_1, h_2)$ 作为输入，它会输出什么呢? 第二层对应的就是 f，所以会第二层输出 $\omega_0 1 + \omega_1 h_1 + \omega_2 h_2$，第二层的输出又会作为第三层的输入，所以第三层输出是 $g(\boldsymbol{\omega} \cdot \boldsymbol{h})$，这里 g 也就是 sign，$\boldsymbol{h}$ 也就是 $(1, h_1, h_2)$.

所以图 2-1表示的这个感知机 $g \circ f \circ p$ 它实际上是个超平面，而且是个二维空间中的超平面，直接在坐标轴上画出来会是一条直线，然后比方说可以判定直线上方的是一类，直线上的是一类，直线下方的是一类，通过改变参数向量 $\boldsymbol{\omega}$ 的值可以调整这条直线（也就是超平面）的截距和斜率，使得在测试样本集上的分类正确率得以提升，寻找最优的那个参数 $\boldsymbol{\omega}^\star$ 的过程被称为「训练」.

并且我们还知道，在每一层内部，神经元和神经元之间是不连接的，每一层的神经元只单向地连接到下一层 (只有使用后馈传播 (back propagation) 算法的时候才会反过来)，并且在预测的时候，每一层把这一层的输出结果交给下一层，作为下一层的输入，然后下一层再交给更下一层，直到到达最终的那层（也

就是复合函数套在最外部的那个函数）. 也就是说只有相邻的两层有连接，每一层内部没有连接，不过也有一些其他版本的神经网络不是这样的.

3 神经网络

具备感知能力的感知机经过反反复复地反复复合可以得到神奇的神经网络, 所以我们一开始就说神经网络一般可以看做是一个很复杂的复合函数.

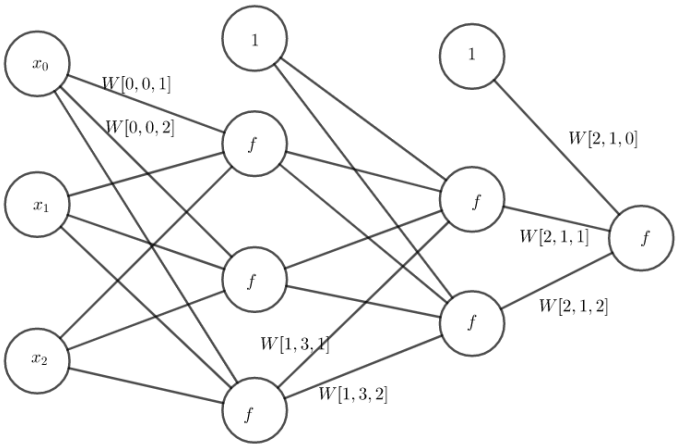

图 3-1: 一个神经网络连接图

图 3-1展示的是一个神经网络，它和前面展示的感知机相比就是多了好几层，所以在一些教程中，把神经网络称作「多层感知机」. 为了简单起见，我们可以把「层」(layer) 看做是一个神经网络的基本组成 (building block)，一个层是一个函数，它的输入是一个数组 (array)，输出也是一个数组，就是这么简单，并且层和层串联起来，让每一层的输出作为下一层的输入，就得到了神经网络. 每一层都内置有参数，这些参数能够决定这一层将如何对输入进行处理得到输出，层的参数确定了，再加上神经网络由哪些层构成确定了，整个神经网络也就确定了. 有时，在一个神经网络中，并不总是简单地将相邻两层连接，但是那种复杂的情况我们不涉及，只要知道在一个神经网络中，还可以指定层和层（甚至神经元和神经元）之间的连接方式就可以了.

我们来一步步地构建我们自己的神经网络实现, 借助 Python 和 `torch.tensor` 的超能力让我们很容易办到这一点，首先我们来认识 `tensor`，实际上 `tensor` 有确切的数学定义，但是，从实用的角度来说，认为 `torch.tensor`, `numpy.ndarray`, 和高维数组是同一回事是没有问题的:

```
import torch

tensor_1 = torch.tensor([1,2,3])
tensor_2 = torch.tensor([[1,2,3],[4,5,6]])
tensor_3 = torch.tensor([
    [[1,2,3],
     [4,5,6]],
    [[7,8,9],
     [10,11,12]]
])
```

这里 `tensor_1` 是向量，`tensor_2` 是矩阵，`tensor_3` 是高维数组，拿高维数组作为数组的元素还能组成更高维的数组，它们统称为 tensor.

```
tensor_1
tensor_2
tensor_3

print(tensor_1.shape)
print(tensor_2.shape)
print(tensor_3.shape)
```

通过运行上列命令，我们就对 tensor 有了一个基本的理解.

现在我们来实现并且试用一个最简单的 layer:

```
def identity_layer(x: torch.tensor) -> torch.tensor:
    return x.clone()

input_value = torch.rand(2, 4)
output_value = identity_layer(input_value)

input_value == output_value
```

可以看到，`identity_layer` 正如其名，保持了输入的 identity.

然后我们再来实现一个线性层:

```
class LinearLayer:

    weight: torch.tensor

    def __init__(self, initial_weight: torch.tensor) -> None:
        self.weight = initial_weight

    def __call__(self, x: torch.tensor) -> torch.tensor:
        return self.weight @ x

linear_layer = LinearLayer(torch.tensor([[1,0,0],[0,2,0],[0,0,3]]))
input_value = torch.tensor([[3],[4],[5]])
output_value = linear_layer(input_value)
```

输出为:

```
tensor([[ 3],
        [ 8],
        [15]])
```

可以看到，`LinearLayer` 产生的是一个线性算子，而这个线性算子的工作方式又由 `self.weight` 决定，所以我们称这样的 `self.weight` 为「参数」或者「可学习的参数」(learnable parameters)，有时，仅有一个 `weight` 对于一个线性层来说不足以使它称职，所以为了使它发挥它的全部能力，我们再给它加上一个「偏置」(bias)，当然，和 `weight` 一样，`bias` 也是可学习参数:

```
class LinearLayer:

    weight: torch.tensor
    bias: torch.tensor

    def __init__( self, initial_weight: torch.tensor, bias: torch.tensor) -> None:
        self.weight = initial_weight
```

```
        self.bias = bias

    def __call__(self, x: torch.tensor) -> torch.tensor:
        return (self.weight @ x) + self.bias

linear_layer = LinearLayer(
    torch.tensor([[1,0,0],[0,2,0],[0,0,3]]),
    torch.tensor([[100],[100],[100]])
)

input_value = torch.tensor([[3],[4],[5]])
output_value = linear_layer(input_value)
```

上面我们对 `input_value` 做了一个矩阵乘法和向量加法:

$$\text{output\_value} = \begin{bmatrix} 100 \\ 100 \\ 100 \end{bmatrix} + \begin{bmatrix} 1 & & \\ & 2 & \\ & & 3 \end{bmatrix} \text{input\_value} \tag{3.1}$$

输出为:

```
tensor([[103],
        [108],
        [115]])
```

哪怕这样简单的一个线性层也能做一些事情：我们首先随机产生一个线性层 f_1，然后我们随机生成一些特征数据 $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$，然后我们计算出正确的标签数据 $\boldsymbol{y}_1 = f_1(\boldsymbol{x}_1), \cdots, \boldsymbol{y}_n = f_2(\boldsymbol{x}_n)$，然后我们再随机产生一个线性层 f_2，设 f_1 和 f_2 的参数分别是 (W_1, B_1) 和 (W_2, B_2)，下面我们将说明，通过使用梯度下降方法，将能够学习到 (W_2, B_2) 对于 (W_1, B_1) 的调整量，在 (W_2, B_2) 加上这个调整量之后，它就变得跟 (W_1, B_1) 几乎一样了，这时候也就相当于，我们仅仅从数据 $\{(\boldsymbol{x}_1, \boldsymbol{y}_1), \cdots, (\boldsymbol{x}_n \boldsymbol{y}_n)\}$ 中挖掘信息，并将这些信息传授给 f_2（的参数），就能够使得 f_2 具备足以模仿 f_1 的能力．我们一起来看吧:

```
weights = torch.rand(3, 4)
bias = torch.rand(3, 1)
linear_layer_0 = LinearLayer(weights, bias)
```

这里 `linear_layer_0` 相当于 f_1，就是那个即将被模仿的函数，然后:

```
n_samples = 1000
inputs = torch.rand(4, n_samples)
outputs = linear_layer_0(inputs)
```

这里 `inputs` 和 `outputs` 就携带着 `linear_layer_0` 的信息, 假设现在我们已经彻底忘记一开始设定的那组参数 `weights` 和 `bias` 了，我们能否通过 `inputs` 和 `outputs` 这组数据把 `weights` 和 `bias` 重现呢? 首先假设我们还记得 `weights` 的维度，也记得 `bias` 的维度，那我们就开始随机猜吧:

```
guess_weights = torch.rand(3, 4, requires_grad=True)
guess_bias = torch.rand(3, 1, requires_grad=True)
```

那么这个随机猜出来的值,猜得多准呢?比较虽然我们忘记了参数 `weights` 和 `bias`,但是数据 `inputs` 和 `outputs` 还在:

```
guess_model = LinearLayer(guess_weights, guess_bias)
test_results = guess_model(inputs)
error = torch.sum(torch.sum(torch.pow(test_results - outputs, 2), dim=0))
error.item()
```

输出为:

```
1838.34326171875
```

我们怎么通过调节 guess_weights 和 guess_bias 的值使得 error 变小呢? 求导, 也就是求梯度:

```
error.backward(retain_graph=True)
```

那么求出来的导数, 也就是 error 关于 guess_* 的梯度等于多少呢?

```
print(guess_weights.grad)
print(guess_bias.grad)
```

输出为:

```
tensor([[1049.4083,  956.4597,  893.3478, 1061.4321],
        [ 827.7568,  818.9000,  906.5511,  823.2509],
        [ 433.6943,  380.7273,  354.0989,  273.8049]])
tensor([[1956.2354],
        [1569.1124],
        [ 675.5887]])
```

那么既然知道了梯度, 该怎么操作 guess_* 使得 error 减小呢?

```
lr = 1e-4

guess_weights = (guess_weights - lr * guess_weights.grad)\
.clone().detach().requires_grad_(True)

guess_bias = (guess_bias - lr * guess_bias.grad)\
.clone().detach().requires_grad_(True)
```

经过这样的修改之后, error 是否真的减小了呢?

```
guess_model = LinearLayer(guess_weights, guess_bias)
test_results = guess_model(inputs)
error = torch.sum(torch.sum(torch.pow(test_results - outputs, 2), dim=0))
error.item()
```

输出为:

```
719.6346435546875
```

那么一直重复这个计算误差–计算梯度–修改参数的过程, 最终能使 error 减小到相当小的水平:

```
weights = torch.rand(3, 4)
bias = torch.rand(3, 1)
linear_layer_0 = LinearLayer(weights, bias)

n_samples = 1000
inputs = torch.rand(4, n_samples)

outputs = linear_layer_0(inputs)

guess_weights = torch.rand(3, 4, requires_grad=True)
guess_bias = torch.rand(3, 1, requires_grad=True)

print(torch.sum(torch.pow(weights - guess_weights, 2)))
print(torch.sum(torch.pow(bias - guess_bias, 2)))
```

```
15
16 max_rounds = 1000
17 lr = 1e-4
18 for i in range(max_rounds):
19     guess_model = LinearLayer(guess_weights, guess_bias)
20     test_results = guess_model(inputs)
21
22     error = torch.sum(torch.sum(torch.pow(test_results - outputs, 2), dim=0))
23
24     if (i % (max_rounds/10)) == 0:
25         print(f"round: {i}, error: {error.item()}")
26
27     error.backward(retain_graph=True)
28
29     guess_weights = (guess_weights - lr * guess_weights.grad)\
30     .clone().detach().requires_grad_(True)
31
32     guess_bias = (guess_bias - lr * guess_bias.grad)\
33     .clone().detach().requires_grad_(True)
34
35 print(torch.sum(torch.pow(weights - guess_weights, 2)))
36 print(torch.sum(torch.pow(bias - guess_bias, 2)))
```

以上其实就算是演示了一遍完整的梯度下降算法.

神经网络其实可以看做是一个函数近似器，假如说我有一个表达式未知的函数 f，但是我有关于 f 的一些资料 $\{(\boldsymbol{x}_1, f(\boldsymbol{x}_1)), \cdots, (\boldsymbol{x}_n, f(\boldsymbol{x}_n))\}$，那么我可以首先确定一下神经网络的结构，结构越复杂，这个神经网络就能学习越复杂的函数，确定了神经网络的结构之后，再随机初始化神经网络的可学习参数的值，确定了神经网络的值以后，我们可以把神经网络看做是一个函数 g，它对于 $\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n\}$ 中的任何一个 $\boldsymbol{x}$，都能够计算出 $g(\boldsymbol{x})$ 具体的值来，那么当我们计算出了一些 $G = \{g(\boldsymbol{x}_i) : i \in \Lambda\}$，再拿去和 $F = \{f(\boldsymbol{x}_i) : i \in \Lambda\}$ 做一个比较，就立刻知道应该如何改正 g 的参数，使得在下一轮的比较中 G 与 F 的差别变得更小，这样日拱一卒，最终能使 g 足够接近 f，这时我们说，g 通过从数据和错误中学习，已经习得了在一定程度上模仿 f 的能力.

我们再介绍一个 layer:

```
1 class SigmoidLayer:
2     def __call__(self, x: torch.tensor) -> torch.tensor:
3         return torch.sigmoid(x)
```

考虑表 3-1所示数据集:

表 3-1: 数据产生自 or 函数

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

有了 `SigmoidLayer`，我们就能够搭建出有能力模仿 or 函数的神经网络. 首先 Sigmoid 函数的表

达式是这样

$$\mathrm{Sigmoid}(x) = \frac{1}{1+\mathrm{e}^{-x}} \tag{3.2}$$

观察它的表达式我们知道，它的值域是 $(0,1)$，也就是说，Sigmoid 函数输出的值，具有概率性质，从某种意义上，我们可以将这个值视为对概率 $\Pr\{y=1|x_1,x_2\}$ 的一个估计，况且我们看到 y 是离散的，所以学习通过学习它的概率而不是学习它本身，也算是一个明智的选择，否则如果直接学习 y，梯度就不好计算.

由于我们实现的 SigmoidLayer 只是按元素取 Sigmoid，所以它没有参数，至少在我们的这个实现里面没有:

```
def calc_error(weight: torch.tensor, bias: torch.tensor) -> float:
    linear_layer = LinearLayer(weight, bias)
    sigmoid_layer = SigmoidLayer()

    predict = sigmoid_layer(linear_layer(inputs))
    error = torch.sum(torch.pow(predict - outputs, 2))

    return error

def gradient_update(lr: float, origin: torch.tensor) -> torch.tensor:
    return (origin - lr * origin.grad).clone().detach().requires_grad_(True)
```

我们定义了几个辅助函数，它能够使得接下来的代码变得清晰易读:

```
inputs = torch.tensor([[0,0],[0,1],[1,0],[1,1]], dtype=torch.float).T
outputs = torch.tensor([[0, 1, 1, 1]], dtype=torch.float)

guess_weights = torch.rand(1, 2, requires_grad=True)
guess_bias = torch.rand(1, 1, requires_grad=True)

max_rounds = 100000
lr = 1e-2
for i in range(max_rounds):
    error = calc_error(guess_weights, guess_bias)
    if i % (max_rounds/10) == 0:
        print(f"Round: {i}, \t Error: {error.item()}")

    error.backward(retain_graph=True)
    guess_weights = gradient_update(lr, guess_weights)
    guess_bias = gradient_update(lr, guess_bias)
```

首先我们将要被学习的 or 函数的数据输入，然后在随机初始化我们的网络，再根据权重不断地修正我们的网络，使该网络逐渐达到尽善尽美的地步:

```
final_linear = LinearLayer(guess_weights, guess_bias)
final_sigmoid = SigmoidLayer()
def final_model(x: torch.tensor) -> torch.tensor:
    return final_sigmoid(final_linear(x))

print(final_model(inputs))
```

输出为:

```
tensor([[0.0378, 0.9763, 0.9763, 1.0000]], grad_fn=<SigmoidBackward>)
```

可以看到我们的模型已经学到了 or 函数.

4 总结

在当前的学习阶段，将神经网络理解为复合函数，将神经网络学习参数的过程理解为求导数再微调是没有问题的，事实上，梯度下降的衍生算法占据着主要的位置．然而，在理解了基本概念之后，我们建议读者立即做进一步的学习，以检验自己的学习效果，顺便趁热打铁做进一步的学习，从而让自己对神经网络的认识更上一层楼．