基于 tensor 实现的神经网络

2333105 Ning Zhiheng

1 理论基础

1.1 矩阵求导

众所周知，对于实值标量函数 $y = f(\boldsymbol{X})$ 其中 $y \in \mathbb{R}$，$\boldsymbol{X} \in \mathbb{R}^{p\times q}$，$f: \mathbb{R}^{p\times q} \to \mathbb{R}$。我们有下式成立：

$$dy = \mathrm{tr}(\frac{\partial f}{\partial \boldsymbol{X}^T} d\boldsymbol{X}) \tag{1.1}$$

由于没有既成的矩阵复合求导公式，并不能简单遵循实数求导的链式法则认为 $\frac{\partial y}{\partial \boldsymbol{X}} = \frac{\partial f}{\partial \boldsymbol{G}(\boldsymbol{X})}\frac{\partial \boldsymbol{G}(\boldsymbol{X})}{\partial \boldsymbol{X}}$，其中 $f: \mathbb{R}^{m\times n} \to \mathbb{R}, \boldsymbol{G}: \mathbb{R}^{p\times q} \to \mathbb{R}^{m\times n}$ 是成立的。

下面我们从微分的本质来推导矩阵复合函数的求导法则，给定 $y = f(\boldsymbol{M})$，$\boldsymbol{M} = \boldsymbol{AXB}$，求 $\frac{\partial y}{\partial \boldsymbol{X}}$

$$dy = \mathrm{tr}(\frac{\partial f}{\partial \boldsymbol{M}^T} d\boldsymbol{M}) = \mathrm{tr}(\frac{\partial f}{\partial \boldsymbol{M}^T} \boldsymbol{A} d\boldsymbol{X}\boldsymbol{B}) = \mathrm{tr}((\boldsymbol{A}^T \frac{\partial f}{\partial \boldsymbol{M}} \boldsymbol{B}^T)^T d\boldsymbol{X}) \tag{1.2}$$

即 $\frac{\partial y}{\partial \boldsymbol{X}} = \boldsymbol{A}^T \frac{\partial f}{\partial M} \boldsymbol{B}^T$。
（为了节省篇幅，本文后面给定的抽象函数都认为是实值标量函数），

1.2 监督学习

先回忆一下监督学习的范式,定义样本空间 $\boldsymbol{X} \in \mathcal{X}$,标记空间 $\boldsymbol{Y} \in \mathcal{Y}$ 以及包含了从 $\mathcal{X}$ 到 $\mathcal{Y}$ 的假设空间 $\mathcal{H} = \mathcal{X} \times \mathcal{Y}$,同时考虑损失函数 $\mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$，$\boldsymbol{P}$ 为样本空间中 $\mathcal{H}$ 的测度，我们要做的就是最小化损失函数，即

$$\min_{\boldsymbol{w},\boldsymbol{b}} E_p(L(\hat{\boldsymbol{Y}}, h_{\boldsymbol{w},\boldsymbol{b}}(\boldsymbol{X}))) \tag{1.3}$$

其中，$\hat{\boldsymbol{Y}}$ 为真实值，$h \in \mathcal{H}$。
给定样本集 $\mathcal{D} = \{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_n}, y_n)\}$，其中 $\boldsymbol{x_i} \in \mathbb{R}^d$，$y_i \in \mathbb{R}$，$|\mathcal{D}| = n$。并且 $\mathcal{D}$ 中每个样本集都是独立从样本空间采样的，我们有 $\lim\limits_{n\to+\infty} \frac{1}{n} L(\hat{\boldsymbol{Y}}, h_{\boldsymbol{w},\boldsymbol{b}}(\boldsymbol{X})) = E_p(L(\hat{\boldsymbol{Y}}, h_{\boldsymbol{w},\boldsymbol{b}}(\boldsymbol{X})))$，这是根据 Hoeffding 不等式给出的, 其中 $\frac{1}{n} L(\hat{\boldsymbol{Y}}, h_{\boldsymbol{w},\boldsymbol{b}}(\boldsymbol{X}))$ 是经验损失函数。
接着，问题便转换成优化经验损失函数，即：

$$\boldsymbol{w}, \boldsymbol{b} = \arg\min_{\boldsymbol{w},\boldsymbol{b}} L(\hat{\boldsymbol{Y}}, h_{\boldsymbol{w},\boldsymbol{b}}(\boldsymbol{X})) \tag{1.4}$$

下面使用梯度下降法来求解这个问题（完整思想不再赘述，请读者自行查阅）。梯度下降法的核心是 **沿着梯度反方向的方向，函数局部下降最快**。即可以通过迭代的方式，不断优化参数逼近最小值，表达式如下：

$$\begin{aligned} \boldsymbol{w}^{(k+1)} &= \boldsymbol{w}^{(k)} - \alpha \nabla_{\boldsymbol{w}} L \\ \boldsymbol{b}^{(k+1)} &= \boldsymbol{b}^{(k)} - \alpha \nabla_{\boldsymbol{b}} L \end{aligned} \tag{1.5}$$

其中 α 是步长，$\boldsymbol{w}^{(k)}$、$\boldsymbol{b}^{(k)}$ 是第 k 次迭代的参数。
按照上述流程，假如我们一直迭代，便可以得到函数取**全局最小值**时的参数 $\boldsymbol{w}, \boldsymbol{b}$。细心的读者可能会发现，既然用这么简单的方法就可以求解，那为什么深度学习（神经网络）中还有如此多调参的“艺术”？
这需要解释三个核心的点：

1) 沿着梯度反方向函数值确实会减少，但如果 α 太大，可能就可能适得其反，如图 1.1，所以正确的调参模式应该为：

$$\begin{aligned} \boldsymbol{w}^{(k+1)} &= \arg\min_{\boldsymbol{w}} L(\boldsymbol{w}^{(k)} - \alpha \nabla_{\boldsymbol{w}} L) \\ \boldsymbol{b}^{(k+1)} &= \arg\min_{\boldsymbol{b}} L(\boldsymbol{b}^{(k)} - \alpha \nabla_{\boldsymbol{b}} L) \end{aligned} \tag{1.6}$$

   由于 $\boldsymbol{w}, \boldsymbol{b}$ 是变值，每一次迭代取得最小值的 α 便不是唯一的，可以使用精确步长搜索方法确定 α。为了简单实践，本文采用简单批量随机梯度下降法，并设置 $\alpha = 0.01$，这可以近似保证函数的值呈现下降趋势。

2) 简单梯度下降法是一个贪心的算法，因此不能保证函数在**全局**上下降得最快，若求解一个可解问题所需的时间很多，那么算法便失去了意义，这时候我们说这个问题是“无解”的。

3) 通常来说我们希望损失函数为凸函数，因为梯度下降法一旦收敛便是该函数的全局最小值点；而对于非凸函数来说，由于存在多个局部最小值点，一旦算法跳进局部陷阱，便很难跳出来。

所以为了更好更快地使函数收敛到最小值，神经网络超参数调优也是一个值得研究的领域。

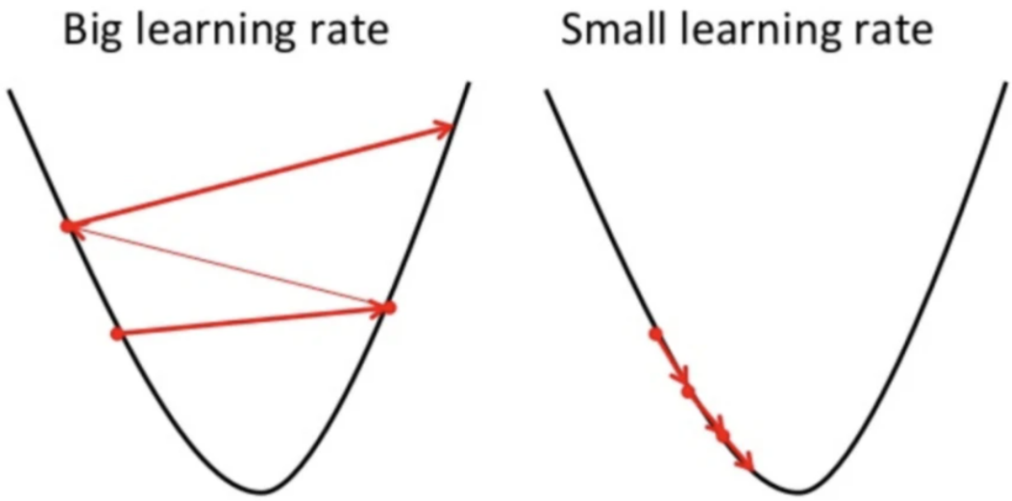

图 1.1: 步长选择策略

2 网络结构

2.1 全连接层

首先给出全连接层的数学模型，设输入该层的数据为 $\boldsymbol{X} \in \mathbb{R}^{n\times d}$，权重矩阵为 $\boldsymbol{W} \in \mathbb{R}^{d\times p}$,，偏置向量为 $\boldsymbol{b} \in \mathbb{R}^{1\times p}$，输出为 $\boldsymbol{Y} \in \mathbb{R}^{n\times p}$。则：

$$\boldsymbol{Y}_{n\times p} = \boldsymbol{X}_{n\times d}\boldsymbol{W}_{d\times p} + \mathbf{1}_{n\times 1}\boldsymbol{b}_{1\times p} \tag{2.1}$$

这实际完成了特征从 d 维空间到 p 维空间的映射。

2.1.1 前向传播

因此，我们可以写出前向传播对应的函数：

```
def forward(self, x):
    self.x = x
    sample_dim = x.shape[0]
    self.tensor_one = torch.ones([sample_dim, 1], dtype=torch.
                                 float64)
    x = torch.mm(x, self.w) + torch.mm(self.tensor_one, self.b)
    return x
```

2.1.2 反向传播

根据 1.1，容易给出 L 对 (W, b, X) 的偏导：

$$\begin{aligned}\nabla_{\boldsymbol{W}} L &= \boldsymbol{X}^T \frac{\partial L}{\partial \boldsymbol{Y}} \\ \nabla_{\boldsymbol{b}} L &= \mathbf{1}_{n\times 1}{}^T \frac{\partial L}{\partial \boldsymbol{Y}} \\ \nabla_{\boldsymbol{X}} L &= \frac{\partial L}{\partial \boldsymbol{Y}} \boldsymbol{W}^T \end{aligned} \tag{2.2}$$

因此我们可以写出反向传播对应的函数：

```
def backward(self, top_gard):
    self.dw = torch.mm(self.x.transpose(0, 1), top_gard)
    self.db = torch.mm(self.tensor_one.transpose(0, 1), top_gard
                                 )
    bottom_gard = torch.mm(top_gard, self.w.transpose(0, 1))
    return bottom_gard
```

2.2 Relu 层

给出 Relu 表达式:

$$\mathrm{Relu}(\boldsymbol{X}) = \max(\boldsymbol{X}, 0) = \begin{cases} \boldsymbol{X} & \boldsymbol{X} \geq 0 \\ 0 & otherwise \end{cases} \tag{2.3}$$

给出 Relu 的图像，如下图所示：

可以看到这是一个非线性的函数，可以作为全连接层的激活函数。

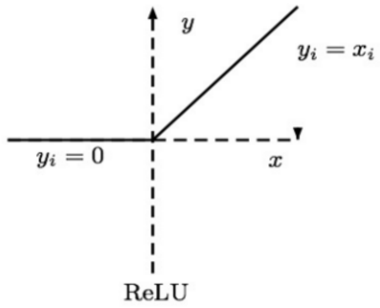

图 2.1: Relu 函数示意图

2.2.1 前向传播

因此，我们可以写出前向传播对应的函数：

```
def forward(self, x):
    self.x = x
    x = torch.maximum(x, torch.tensor([0.0]))
    return x
```

2.2.2 反向传播

根据式(2.3)，求出 Relu(x) 的导数：

$$\mathrm{Relu}'(x) = \begin{cases} 1 & x \geq 0 \\ 0 & otherwise \end{cases} \tag{2.4}$$

接着写出反向传播对应的公式：

$$\nabla_{\boldsymbol{X}} L = \nabla_{\boldsymbol{Y}} L \circ \mathrm{Relu}'(\boldsymbol{X}) = \begin{cases} \nabla_{\boldsymbol{Y}} L & \boldsymbol{X} \geq 0 \\ 0 & otherwise \end{cases} \tag{2.5}$$

其中 $\circ$ 为哈达玛积。
因此我们可以写出反向传播对应的函数：

```
def backward(self, top_grad):
    bottom_grad = top_grad
    bottom_grad[self.x < 0] = 0
    return bottom_grad
```

2.3 softmax 层

给出多分类情况下大家常用的 softmax 表达式：

$$\mathrm{softmax}(\boldsymbol{x}) = \frac{\exp(\boldsymbol{x})}{\sum_{i=1}^{d}\exp(\boldsymbol{x})} = \frac{\exp(\boldsymbol{x})}{\boldsymbol{1}^T\exp(\boldsymbol{x})} \tag{2.6}$$

其中 $x \in \mathbb{R}^d$，可以看到本质上就是一个将向量中每个元素映射到区间 $[0,1]$ 的函数，即 $f:\mathbb{R}^n \to \mathbb{R}_{[0,1]}$。若 $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ 为矩阵，则公式变为：

$$\begin{aligned}\mathrm{softmax}(\boldsymbol{X}) &= (\mathrm{softmax}(\boldsymbol{x_1}),\ldots,\mathrm{softmax}(\boldsymbol{x_n}))^T \\ &= \mathrm{diag}^{-1}\{\exp(\boldsymbol{X})\boldsymbol{1}\}\exp(\boldsymbol{X})\end{aligned} \tag{2.7}$$

其中 $\boldsymbol{X} = (\boldsymbol{x_1},\ldots,\boldsymbol{x_n})^T$。
读者看到这里可能会感觉疑惑，$\mathrm{softmax}(\boldsymbol{X})$ 与 $\boldsymbol{X}$ 都是矩阵的形式，而我们从未提及 $\frac{\partial\mathrm{softmax}(\boldsymbol{X})}{\partial\boldsymbol{X}}$ 这样矩阵对矩阵的求导形式，该怎么办？
事实上，在神经网络中，softmax 与交叉熵函数是成双成对出现的。为了便于后续说明，在此先给出交叉熵函数的数学表达式：

$$L(\hat{\boldsymbol{y}},\boldsymbol{y}) = \sum_{i=1}^{d} -\hat{y}_i\log(y_i) = -\hat{\boldsymbol{Y}}^T\log(\boldsymbol{Y}) \tag{2.8}$$

其中 $\boldsymbol{y} = (y_1,\ldots,y_d)^T$，$\hat{\boldsymbol{y}} = (\hat{y}_1,\ldots,\hat{y}_d)^T$，$\hat{\boldsymbol{Y}} = (\hat{\boldsymbol{y}}_{\boldsymbol{1}},\ldots,\hat{\boldsymbol{y}}_{\boldsymbol{n}})^T$，$\hat{y}_i = \begin{cases}1 & i=k\\ 0 & i\neq k\end{cases}$，表明样本属于第 k 类。
考察式(2.8)，容易发现其为一个标量。于是尝试把式(2.7)与式(2.8)结合起来以实现标量 L 对矩阵 $\boldsymbol{X}$ 的求导。至此，给出重新定义的 softmax 层的公式：

$$\begin{aligned}L(\hat{\boldsymbol{Y}},h(\boldsymbol{X})) &= \sum_{i=1}^{n} -\hat{\boldsymbol{y}}_{\boldsymbol{i}}\log(\mathrm{softmax}(\boldsymbol{x_i})) = \mathrm{tr}(-\hat{\boldsymbol{Y}}\log\mathrm{softmax}(\boldsymbol{X})) \\ &= -\mathrm{tr}(\hat{\boldsymbol{Y}}\log(\mathrm{diag}^{-1}\{\exp(\boldsymbol{X})\boldsymbol{1}\}))\end{aligned} \tag{2.9}$$

2.3.1 前向传播公式

因此，我们可以写出前向传播对应的函数（这包含了损失函数）：

```
def softmax(x):
    x = torch.exp(x)
    x = x / torch.sum(x, dim=1, keepdim=True)
    return x

def forward(self, x):
    self.x = x
    self.outputs = softmax(x)
    return self.outputs
    def get_loss(self, true_targets):
        assert true_targets.shape[0] == self.x.shape[0]
        self.true_targets_encode = self.encode_targets(
                                        true_targets)
        total_loss = 0.0
        for i, true_target in enumerate(true_targets):
            loss = self.sample_loss(i)
            total_loss += loss
        total_loss = total_loss / true_targets.shape[0]
        return total_loss

def sample_loss(self, index):
    return -torch.dot(torch.log(self.outputs[index]), self.
                                    true_targets_encode[index])
```

2.3.2 反向传播公式

同样的，给出反向传播对应的公式：

$$\nabla_{\boldsymbol{X}} L = \frac{1}{m}(\text{softmax}(\boldsymbol{X}) - \hat{\boldsymbol{Y}}) = \frac{1}{m}(\boldsymbol{Y} - \hat{\boldsymbol{Y}}) \tag{2.10}$$

因此我们可以写出反向传播对应的函数：

```
def backward(self):
    sample_count = self.outputs.shape[0]
    bottom_grad = (self.outputs - self.true_targets_encode) /
                                    sample_count
    return bottom_grad
```