Neural Network Implementation Based on Tensor

2333105 Ning Zhiheng

1 Theoretical Basis

1.1 Matrix Calculus

As is well known, for real-valued scalar functions $y = f(\boldsymbol{X})$ where $y \in \mathbb{R}$, $\boldsymbol{X} \in \mathbb{R}^{p\times q}$, $f : \mathbb{R}^{p\times q} \to \mathbb{R}$. We have the following formula for establishment:

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

Due to the lack of established matrix composite differentiation formulas, the chain rule for real number differentiation cannot be simply followed.Assuming $\frac{\partial y}{\partial \boldsymbol{X}} = \frac{\partial f}{\partial \boldsymbol{G}(\boldsymbol{X})} \frac{\partial \boldsymbol{G}(\boldsymbol{X})}{\partial \boldsymbol{X}}$ where $f : \mathbb{R}^{m\times n} \to \mathbb{R}, \boldsymbol{G} : \mathbb{R}^{p\times q} \to \mathbb{R}^{m\times n}$ is valid.
Let's derive the derivation rule for matrix composite functions from the essence of differentiation, given $y = f(\boldsymbol{M})$, $\boldsymbol{M} = \boldsymbol{A}\boldsymbol{X}\boldsymbol{B}$, find $\frac{\partial y}{\partial \boldsymbol{X}}$:

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

l.e. $\frac{\partial y}{\partial \boldsymbol{X}} = \boldsymbol{A}^T \frac{\partial f}{\partial \boldsymbol{M}} \boldsymbol{B}^T$.
In order to save space, the abstract functions given later in this article are considered to be real valued scalar functions.

1.2 Supervised Learning

Let's first recall the paradigm of supervised learning, defining the sample space $\boldsymbol{X} \in \mathcal{X}$, and the label space $\boldsymbol{Y} \in \mathcal{Y}$ And contains the hypothetical space $\mathcal{H} = \mathcal{X} \times \mathcal{Y}$ from $\mathcal{X}$ to $\mathcal{Y}$ Considering the loss function $\mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, where $\boldsymbol{P}$ is the measure of $\mathcal{H}$ in the sample space, what we need to do is minimize the loss function, which is

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

where $\hat{\boldsymbol{Y}}$ is true value , $h \in \mathcal{H}$.
Given sample dataset $\mathcal{D} = \{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_n}, y_n)\}$ where $\boldsymbol{x_i} \in \mathbb{R}^d$, $y_i \in \mathbb{R}$, $|\mathcal{D}| = n$. And each sample set in $\mathcal{D}$ is independently sampled from the sample space, we have $\lim_{n\to+\infty} \frac{1}{n} L(\hat{\boldsymbol{Y}}, h_{\boldsymbol{w},\boldsymbol{b}}(\boldsymbol{X})) = E_p(L(\hat{\boldsymbol{Y}}, h_{\boldsymbol{w},\boldsymbol{b}}(\boldsymbol{X})))$, this is given based on the Hoeffding inequality. where $\frac{1}{n} L(\hat{\boldsymbol{Y}}, h_{\boldsymbol{w},\boldsymbol{b}}(\boldsymbol{X}))$ is an empirical loss function.
Next, the problem is transformed into an optimization empirical loss function, which is:

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

Below, we will use the gradient descent method to solve this problem (the complete idea will not be repeated, please refer to it for yourself). The core of the gradient descent method is **Along the direction opposite to the gradient, the local descent of the function is the fastest**. By iteratively optimizing parameters to approach the minimum value, the expression is as follows:

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

Whereα is the step size ,$\boldsymbol{w}^{(k)}$,$\boldsymbol{b}^{(k)}$ are the parameters of the kth iteration.
Following the above process, if we continue to iterate, we can obtain the parameter $\boldsymbol{w}, \boldsymbol{b}$, when the function takes **global minimum**. Careful readers may find that since solving can be done with such a simple method, why is there so much "art" of tuning parameters in deep learning (neural networks)? There are three core points to explain here:

1) The function value will indeed decrease along the opposite direction of the gradient, but if the α is too large, it may have the opposite effect, as shown in Figure 1.1, So the correct parameter adjustment mode should be:

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

Since $\boldsymbol{w}, \boldsymbol{b}$ is a variable, the αthat obtains the minimum value in each iteration is not unique, and precise step size search methods can be used to determine α. For simple practice, this article adopts the simple batch random gradient descent method and sets $\alpha = 0.01$, which can approximately ensure that the value of the function shows a downward trend.

2) The simple gradient descent method is a greedy algorithm, so it cannot guarantee that the function will descend the fastest on **global**. If solving a solvable problem requires a lot of time, So the algorithm loses its meaning, and at this point we say that the problem is "unsolvable".

3) Generally speaking, we want the loss function to be convex, because once the gradient descent method converges, it is the global minimum point of the function; For non convex functions, due to the existence of multiple local minimum points, once the algorithm jumps into the local trap, it is difficult to jump out.

So in order to better and faster converge the function to the minimum value, neural network hyperparameter tuning is also a field worth studying.

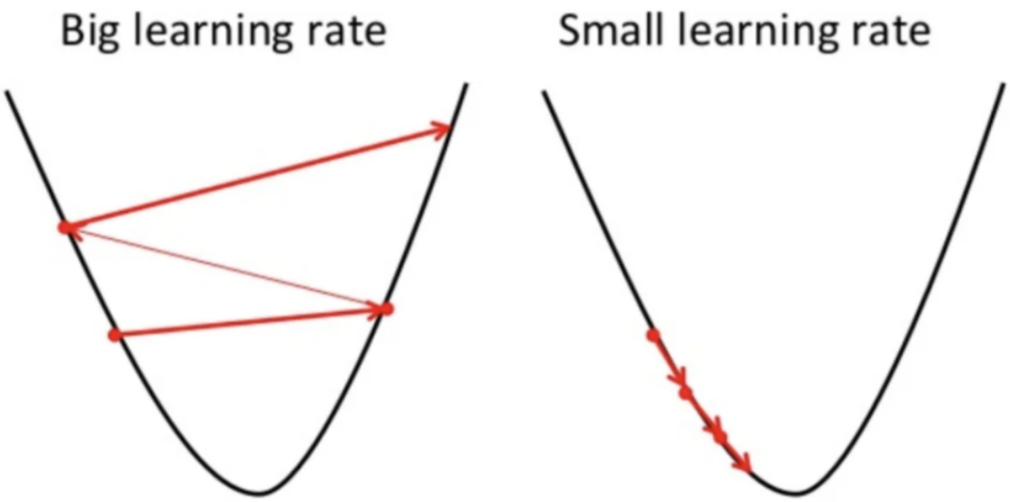

Figure 1.1: Step size selection strategy

2 Network Structure

2.1 Fully Connected Layer

Firstly, provide a mathematical model for the fully connected layer, and assume that the data input to this layer is $\boldsymbol{X} \in \mathbb{R}^{n\times d}$, the weight matrix is $\boldsymbol{W} \in \mathbb{R}^{d\times p}$, the bias vector is $\boldsymbol{b} \in \mathbb{R}^{1\times p}$, output is $\boldsymbol{Y} \in \mathbb{R}^{n\times p}$. i.e.:

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

This actually completes the mapping of features from the $d$dimensional space to the $p$dimensional space.

2.1.1 Forward Propagation

Therefore, we can write the function corresponding to forward propagation:

```
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 Back Propagation

According to 1.1, It is easy to give the partial derivative of L to $\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{X}$:

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

Therefore, we can write the function corresponding to backpropagation:

```
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 Layer

given the Relu expression :

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

given the image of Relu:
It can be seen that this is a nonlinear function that can serve as the activation function for fully connected layers.

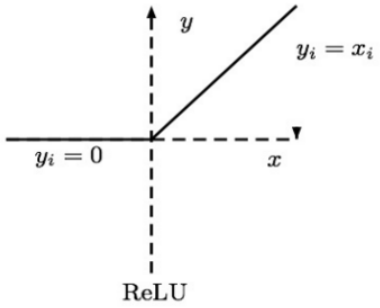

Figure 2.1: Relu function

2.2.1 Forward Propagation

Therefore, we can write the function corresponding to forward propagation:

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

2.2.2 Back Propagation

Calculate the derivative of Relu(x) based on equation (2.3):

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

Next, write the corresponding formula for backpropagation:

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

Where $\circ$ is the Hadamard product.
Therefore, we can write the function corresponding to backpropagation:

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

2.3 Softmax Layer

Given the commonly used softmax expression for multi classification scenarios:

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

Where $x \in \mathbb{R}^d$, It can be seen that it is essentially a function that maps each element in a vector to the interval $[0, 1]$, i.e $f : \mathbb{R}^n \to \mathbb{R}_{[0,1]}$ If $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ is a matrix, then the formula becomes

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

Where $\boldsymbol{X} = (\boldsymbol{x_1}, \ldots, \boldsymbol{x_n})^T$.
Readers may feel puzzled when they see this, as $\text{softmax}(\boldsymbol{X})$ and $\boldsymbol{X}$ are both in the form of matrices, and we have never mentioned $\frac{\partial \text{softmax}(\boldsymbol{X})}{\partial \boldsymbol{X}}$, which is the derivative form of a matrix over a matrix, what should I do?
In fact, in neural networks, softmaxand the cross entropy function appear in pairs. For the convenience of further explanation, the mathematical expression of the cross entropy function is given here first:

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

Where $\boldsymbol{y} = (y_1, \ldots, y_d)^T$, $\hat{\boldsymbol{y}} = (\hat{y_1}, \ldots, \hat{y_d})^T$, $\hat{\boldsymbol{Y}} = (\hat{\boldsymbol{y_1}}, \ldots, \hat{\boldsymbol{y_n}})^T$ $\hat{y}_i = \begin{cases} 1 & i = k \\ 0 & i \neq k \end{cases}$, indicates that the sample belongs to class k.
The examination (2.8) is easily found to be a scalar. So we tried to combine (2.7) with (2.8) to achieve the derivative of scalar L on matrix $\boldsymbol{X}$. At this point, the formula for the redefined softmaxlayer is given:

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

2.3.1 Forward Propagation

Therefore, we can write the function corresponding to forward propagation (which includes the loss function):

```
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 Back Propagation

Similarly, the corresponding formula for backpropagation is given:

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

Therefore, we can write the function corresponding to backpropagation:

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