第十八次作业讲评

郭明非

2022-05-16

神经网络实现-数据集加载

```python
class DataLoader:
    # 黄楠 1900012126
    def __init__(self, dataset: Dataset, batch_size: int, shuffle: bool = True):
        self.dataset = dataset
        self.batch_size = batch_size
        self.shuffle = shuffle

    def __iter__(self):
        if self.shuffle:
            self.batch_idx = np.random.permutation(len(self.dataset))
            return (
                self.dataset[self.batch_idx[i : i + self.batch_size]]
                for i in range(0, len(self.dataset), self.batch_size)
            )
        else:
            return (
                self.dataset[i : i + self.batch_size]
                for i in range(0, len(self.dataset), self.batch_size)
            )
```

- 利用迭代器实现PyTorch DataLoader，可以shuffle以及划分mini-batch。

神经网络实现-Numpy风格

```python
def forward_backward(x,y):
    #forward
    #项昱然 2000017477
    global W1,W2,B1,B2
    z1 = W1 @ x + B1
    a1 = sigmoid(z1)
    z2 = W2 @ a1 + B2
    #loss function调优: 使用softmax+CE 为最后一层的loss function
    a2 = softmax(z2)
    loss = crossentropy(a2.reshape(2),y)
    #backward

    if y == 0:
        grad_z2 = np.array([[-a2[1][0]],[a2[1][0]]])
    else:
        grad_z2 = np.array([[a2[0][0]],[-a2[0][0]]])
    grad_a1 = W2.T @ grad_z2
    grad_z1 = grad_a1 * sigmoid(z1) * (1-sigmoid(z1))
    B2 -= lr * grad_z2
    W2 -= lr * grad_z2 @ a1.T
    B1 -= lr * grad_z1
    W1 -= lr * grad_z1 @ x.T
    return loss
```

```python
def forward_backward(x,y):
    #forward
    #项昱然 2000017477
    global W1,W2,W3,B1,B2,B3
    z1 = W1 @ x + B1
    a1 = relu(z1)
    z2 = W2 @ a1 + B2
    a2 = relu(z2)
    z3 = W3 @ a2 + B3
    #loss function调优: 使用softmax+CE 为最后一层的loss function
    a3 = softmax(z3)
    loss = crossentropy(a3.reshape(2),y)
    #backward
    if y == 0:
        grad_z3 = np.array([[-a3[1][0]],[a3[1][0]]])
    else:
        grad_z3 = np.array([[a3[0][0]],[-a3[0][0]]])

    grad_a2 = W3.T @ grad_z3
    grad_z2 = grad_a2 * relu(np.sign(z2))
    grad_a1 = W2.T @ grad_z2
    grad_z1 = grad_a1 * relu(np.sign(z1))
    B3 -= lr * grad_z3
    W3 -= lr * grad_z3 @ a2.T
    B2 -= lr * grad_z2
    W2 -= lr * grad_z2 @ a1.T
    B1 -= lr * grad_z1
    W1 -= lr * grad_z1 @ x.T
    return loss
```

神经网络实现-Pytorch风格

```python
class Layer:
    """
    Abstract class for nn operations
    """
    # 黄楠 1900012126
    @abstractmethod
    def forward(self, x: np.ndarray, *args, **kwargs):
        ...

    @abstractmethod
    def backward(self, grad_output: np.ndarray):
        ...

    @abstractmethod
    def update(self, lr: float):
        ...

    def __call__(self, x: np.ndarray, *args, **kwargs):
        return self.forward(x, *args, **kwargs)
```

神经网络实现-Pytorch风格

```python
class Linear(Layer):
    def __init__(
        self,
        in_num: int,
        out_num: int,
        bias=True,
        init_std: float = (
        dtype=np.float64,
    ):
        """
        W: (out_num, in_nur
        b: (out_num, 1)
        """
        # 黄楠 1900012126
        self.in_num = in_nu
        self.out_num = out_
        self.bias = True
        # Initialize weight
        self.W = np.random.
        self.b = (
            np.random.normal(scale=init_std, size=(out_num,)).astype(dtype)
            if self.bias
            else 0
        )
        self.grad_W = None
        if self.bias:
            self.grad_b = None
        self.x = None
```

```python
class Sigmoid(Layer):
    # 黄楠 1900012126
    def __init__(self):
        self.x = None

    def forward(self, x: np.ndarray, *args, **kwargs):
        self.x = 1 / (1 + np.exp(-x))
        return self.x

    def backward(self, grad_output: np.ndarray):
        assert self.x is not None, "must call `forward()` before `backward()`"
        return (self.x - self.x**2) * grad_output

    def update(self, lr):
        # reset cache
        self.x = None

    def __repr__(self):
        return "Sigmoid()"
```

神经网络实现-Pytorch风格

```python
class Sequential(Layer):
    def __init__(self, layers: list):
        self.layers = layers

    def forward(self, x: np.ndarray, *args, **kwargs):
        for layer in self.layers:
            x = layer(x)
        return x

    def backward(self, grad_output: np.ndarray):
        for layer in reversed(self.layers):
            grad_output = layer.backward(grad_output)
        return grad_output

    def update(self, lr: float):
        for layer in self.layers:
            layer.update(lr)

    def __repr__(self):
        return (
            "Sequential(\n\t" + "\n\t".join([l.__repr__() for l in self.layers]) + "\n)"
        )
```

```python
# TODO
# 2层前馈网络，以sigmoid或tanh作为激活函数
# 黄楠 1900012126
model = Sequential(
    [
        Linear(784, 128),
        Tanh(),
        Linear(128, 2),
    ]
)
```

调优-数据预处理

```
#数据预处理 归一化防止梯度爆炸，避免溢出
#陈福康 1900013049
x_train = x_train.astype(np.float32)/255
x_test = x_test.astype(np.float32)/255
```

- 原始像素值为0-255，输入神经网络后算出的梯度可能会过大使训练不稳定，可以通过预处理归一化

```
# 数据增强：增加高斯噪声
# 李政 1900012146
mean=0
st=0.05
gauss = np.random.normal(mean,st,x_pre_train.shape)
x_noisy = x_pre_train + gauss
print(x_pre_train.shape,x_noisy.shape)
print(np.max(x_pre_train),np.min(x_pre_train))
```

- 各类数据增强

调优-网络初始化

```python
#调优：实验结果表明，这样比xavier初始化效果更好
#陈睿博 1900012203
self.params['w']=np.random.randn(in_features,out_features)*1e-1
# self.params['w']=np.random.uniform(-np.sqrt(6/(in_features+out_features)),
# np.sqrt(6/in_features+out_features),(in_features,out_features))

self.params['b']=np.random.randn(1,out_features)*1e-1
# self.params['b']=np.zeros((1,out_features))
```

$$U[-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}]$$

- Xavier Initialization (https://zhuanlan.zhihu.com/p/27919794)

```python
def __init__(self, input_size=784, lr=0.01):
    # He Initialization
    # 胡行健 1900017768
    self.linear_layer1 = Tensor(2 * np.random.randn(input_size, 1024) / np.sqrt(input_size))
    self.linear_layer1_bias = Tensor(np.zeros((1, 1024)))
    self.linear_layer2 = Tensor(2 * np.random.randn(1024, 1024) / np.sqrt(1024))
    self.linear_layer2_bias = Tensor(np.zeros((1, 1024)))
    self.output_layer = Tensor(np.random.randn(1024, 1) / np.sqrt(1024))
    self.output_layer_bias = Tensor(np.zeros((1, 1)))
    self.lr = lr
```

- He Initialization (for ReLU) $Var(W^i) = \frac{2}{n_i}$

调优-参数更新

```python
# 增加了moment
# 陈滨琪 2000013185
# 优化器模拟Momentum SGD, momentum=0.5
if len(self.last_bias_grad) == 0:
    for i in range(len(self.weights)):
        self.weights[i] -= self.lr * W_grad[i]
        self.bias[i] -= self.lr * b_grad[i]
    self.last_bias_grad = b_grad
    self.last_weights_grad = W_grad
else:
    for i in range(len(self.weights)):
        self.weights[i] -= self.lr * (W_grad[i] + self.momentum * self.last_weights_grad[i])
        self.bias[i] -= self.lr * (b_grad[i] + self.momentum * self.last_bias_grad[i])
    self.last_bias_grad = b_grad
    self.last_weights_grad = W_grad
```

计算梯度估计：$\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),\boldsymbol{y}^{(i)})$

计算速度更新：$\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\boldsymbol{g}$

- 模拟带动量的SGD——相当于用梯度的指数加权平均来更新参数

调优-损失函数

```python
# aug考虑了数据的不均衡,增大预测为9的部分的loss
# 李思哲 1900013061
def aug_loss(x1, x2):
    mask = np.ones_like(x2)
    mask[x2==1] = 3
    return (np.square(x1-x2)*mask).mean()

def aug_grad(x1, x2):
    mask = np.ones_like(x2)
    mask[x2==1] = 3
    return 2*(x1-x2)*mask
```

- 样本数不均衡（1:3），使用加权的loss

调优-学习率

```python
class ExponentialLR(Scheduler):
    # 黄楠 1900012126
    def __init__(self, optimizer: Optimizer, gamma=0.99):
        super().__init__(optimizer)
        self.gamma = gamma

    def step(self):
        self.target.lr *= self.gamma
```

```python
# ——调优部分——
# 学习率多项式型衰减
# 文天宇 1900017823
def learning_rate_decay_polynomial(init_learning_rate, end_learning_rate, current_epoch, epochs, power=1):
    learning_rate_range = init_learning_rate - end_learning_rate
    remaining = 1 - current_epoch / (epochs - current_epoch)
    current_learning_rate = learning_rate_range * remaining ** power + end_learning_rate

    return current_learning_rate
```

- 训练acc为固定值（0.5/0.75），调低学习率或者数据归一化
- 学习率调整策略：指数衰减，多项式衰减等