PyTorch Tutorial

TA: Binbin Chen

Email: chenbb@shanghaitech.edu.cn

What is PyTorch?

- a popular machine learning framework in Python
- 2 main features:
  - N-dimensional Tendor computation on GPUs
  - Automatic differentiation for training

Tensors:

- **multidimensional arrays,** check with `.shape` (1D-vector, 2D-matrix)
- common operations
- device:
  - `device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')`
  - `.to(device)`
- gradient calculation:
  - `x = torch.tensor(..,..,requires_grad=True); z = func(x); z.backward(); x.grad`

Linear Regression:

1. Dataset Generator:

```
def synthetic_data(w, b, num_examples): #@save """ y = Xw+b+noise
""" X = torch.normal(0, 1, (num_examples, len(w))) y =
torch.matmul(X, w) + b y += torch.normal(0, 0.01, y.shape) return
X, y.reshape((-1, 1)) true_w = torch.tensor([2, -3.4]) true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```

2. Load Data

```
def data_iter(batch_size, features, labels): num_examples = len(fea
tures) indices = list(range(num_examples)) # 这些样本是随机读取的，没有
特定的顺序 random.shuffle(indices) for i in range(0, num_examples, ba
tch_size): batch_indices = torch.tensor( indices[i: min(i + batch_s
ize, num_examples)]) yield features[batch_indices], labels[batch_in
dices]
```

```
batch_size = 10 for X, y in data_iter(batch_size, features,
labels): print(X, '\n', y) break ''' tensor([[ 0.1649, -1.1651],
[-2.0755, -1.0165], [-0.2189, 0.7607], [ 0.6833, 0.3537], [-0.2736,
-2.0485], [-0.3026, 0.9771], [ 2.4795, 0.6881], [-0.2045, -0.8509],
[-0.1353, 0.5476], [ 0.3371, -0.0479]]) tensor([[ 8.4901], [
3.5015], [ 1.1779], [ 4.3752], [10.6125], [ 0.2845], [ 6.8094], [
6.6776], [ 2.0598], [ 5.0189]]) '''
```

3. Weight Initialization

```
w = torch.normal(0, 0.01, size=(2,1), requires_grad=True) b = torc
h.zeros(1, requires_grad=True
```

4. Model Definition

```
def linreg(X, w, b): #@save """线性回归模型""" return torch.matmul(X,
w) + b def squared_loss(y_hat, y): #@save """均方损失""" return
(y_hat - y.reshape(y_hat.shape)) ** 2 / 2 def sgd(params, lr,
batch_size): #@save """小批量随机梯度下降""" with torch.no_grad(): for
param in params: param -= lr * param.grad / batch_size
param.grad.zero_()
```

5. Model Training

each epoch:

> load data → model prediction → get loss → backpropagation for grads → SGD optimize

```
lr = 0.03 num_epochs = 3 net = linreg loss = squared_loss for epoch
in range(num_epochs): for X, y in data_iter(batch_size, features,
labels): l = loss(net(X, w, b), y) # X和y的小批量损失 # 因为l形状是
(batch_size,1)，而不是一个标量。l中的所有元素被加到一起， # 并以此计算关于
[w,b]的梯度 l.sum().backward() sgd([w, b], lr, batch_size) # 使用参数
的梯度更新参数 with torch.no_grad(): train_l = loss(net(features, w,
b), labels) print(f'epoch {epoch + 1}, loss
{float(train_l.mean()):f}')
```

6. Test

```
print(f'w的估计误差: {true_w - w.reshape(true_w.shape)}') print(f'b的
估计误差: {true_b - b}')
```

Linear Regression - simple version:

1. Dataset Generator

2. Dataloader

```
def load_array(data_arrays, batch_size, is_train=True): #@save
"""构造一个PyTorch数据迭代器""" dataset =
data.TensorDataset(*data_arrays) return data.DataLoader(dataset,
batch_size, shuffle=is_train) batch_size = 10 data_iter =
load_array((features, labels), batch_size)
```

3. Model Definition

```
from torch import nn net = nn.Sequential(nn.Linear(2, 1))
net[0].weight.data.normal_(0, 0.01) net[0].bias.data.fill_(0) loss
= nn.MSELoss() trainer = torch.optim.SGD(net.parameters(), lr=0.03)
```

4. Model Training

each epoch:

load data → model prediction → get loss → backpropagation for grads → SGD optimize

```
num_epochs = 3 for epoch in range(num_epochs): for X, y in
data_iter: l = loss(net(X) ,y) trainer.zero_grad() l.backward()
trainer.step() l = loss(net(features), labels) print(f'epoch {epoch
+ 1}, loss {l:f}')
```

MLP - simple version:

0. Pre-function:

```
def train_epoch_ch3(net, train_iter, loss, updater): #@save # 将模型
设置为训练模式 if isinstance(net, torch.nn.Module): net.train() # 训练
损失总和、训练准确度总和、样本数 metric = Accumulator(3) for X, y in
train_iter: # 计算梯度并更新参数 y_hat = net(X) l = loss(y_hat, y) #
使用PyTorch内置的优化器和损失函数 updater.zero_grad()
l.mean().backward() updater.step() metric.add(float(l.sum()),
accuracy(y_hat, y), y.numel()) # 返回训练损失和训练精度 return
metric[0] / metric[2], metric[1] / metric[2]
```

```
def train_ch3(net, train_iter, test_iter, loss, num_epochs,
updater): #@save for epoch in range(num_epochs): train_metrics =
train_epoch_ch3(net, train_iter, loss, updater) test_acc =
evaluate_accuracy(net, test_iter) train_loss, train_acc =
train_metrics
```

1. Model Definition and Training

```
net = nn.Sequential(nn.Flatten(), nn.Linear(784, 256), nn.ReLU(),
nn.Linear(256, 10)) def init_weights(m): if type(m) == nn.Linear:
nn.init.normal_(m.weight, std=0.01) net.apply(init_weights)
```

```
batch_size, lr, num_epochs = 256, 0.1, 10 loss =
nn.CrossEntropyLoss(reduction='none') optimizer =
torch.optim.SGD(net.parameters(), lr=lr) train_iter, test_iter =
d2l.load_data_fashion_mnist(batch_size) d2l.train_ch3(net,
train_iter, test_iter, loss, num_epochs, optimizer)
```

Neural Networks

1. Custom Layers

```
class MLP(nn.Module): # 用模型参数声明层。这里，我们声明两个全连接的层 def
__init__(self): # 调用MLP的父类Module的构造函数来执行必要的初始化。 # 这
样，在类实例化时也可以指定其他函数参数，例如模型参数params（稍后将介绍）
super().__init__() self.hidden = nn.Linear(20, 256) # 隐藏层
self.out = nn.Linear(256, 10) # 输出层 # 定义模型的前向传播，即如何根据输
入X返回所需的模型输出 def forward(self, X): # 注意，这里我们使用ReLU的函数
版本，其在nn.functional模块中定义。 return
self.out(F.relu(self.hidden(X)))
```

```
class MySequential(nn.Module): def __init__(self, *args):
super().__init__() for idx, module in enumerate(args): # 这里,
module是Module子类的一个实例。我们把它保存在'Module'类的成员 # 变量
_modules中。_module的类型是OrderedDict self._modules[str(idx)] =
module def forward(self, X): # OrderedDict保证了按照成员添加的顺序遍历它
们 for block in self._modules.values(): X = block(X) return X net =
MySequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))
```

2. Parameters

```
# Access Parameters print(net[2].state_dict()) '''
OrderedDict([('weight', tensor([[ 0.3016, -0.1901, -0.1991,
-0.1220, 0.1121, -0.1424, -0.3060, 0.3400]])), ('bias',
tensor([-0.0291]))]) ''' print(*[(name, param.shape) for name,
param in net.named_parameters()]) ''' ('0.weight', torch.Size([8,
4])) ('0.bias', torch.Size([8])) ('2.weight', torch.Size([1, 8]))
('2.bias', torch.Size([1])) '''
```

```
# Parameter Initialization def init_normal(m): if type(m) ==
nn.Linear: nn.init.normal_(m.weight, mean=0, std=0.01)
nn.init.zeros_(m.bias) net.apply(init_normal)
net[0].weight.data[0], net[0].bias.data[0] def init_xavier(m): if
type(m) == nn.Linear: nn.init.xavier_uniform_(m.weight) def
init_42(m): if type(m) == nn.Linear: nn.init.constant_(m.weight,
42) net[0].apply(init_xavier) net[2].apply(init_42)
print(net[0].weight.data[0]) print(net[2].weight.data)
```

Save & Load Models:

```
torch.save(model.state_dict(), path) ckpt = torch.load(path)
model.load_state_dict(ckpt)
```