cs391R - Intorduction to Pytorch

Yifeng Zhu

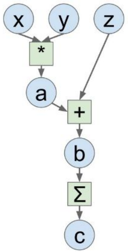
Department of Computer Science
The University of Texas at Austin

September 28, 2020

Disclaimer: Adopted from gatech tutorial: **Link**

Why PyTorch?

Computation Graph



Numpy

```python
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

Tensorflow

```python
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```python
import torch

N, D = 3, 4

x = torch.rand((N, D),requires_grad=True)
y = torch.rand((N, D),requires_grad=True)
z = torch.rand((N, D),requires_grad=True)

a =x * y
b =a + z
c=torch.sum(b)

c.backward()
```

Tensors

Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

Common operations for creation and manipulation of these Tensors are similar to those for ndarrays in NumPy. (rand, ones, zeros, indexing, slicing, reshape, transpose, cross product, matrix product, element wise multiplication)

Tensors

Attributes of a tensor 't':

- `t= torch.randn(1)`

`requires_grad`- making a trainable parameter

- By default False
- Turn on:
    - `t.requires_grad_()` or
    - `t = torch.randn(1, requires_grad=True)`
  - Accessing tensor value:
    - t.data
  - Accessingtensor gradient
    - t.grad

`grad_fn`- history of operations for autograd

- t.grad_fn

```python
import torch

N, D = 3, 4

x = torch.rand((N, D),requires_grad=True)
y = torch.rand((N, D),requires_grad=True)
z = torch.rand((N, D),requires_grad=True)

a = x * y
b = a + z
c=torch.sum(b)

c.backward()

print(c.grad_fn)
print(x.data)
print(x.grad)
```

```
<SumBackward0 object at 0x7fd0cb970cc0>
tensor([[0.4118, 0.2576, 0.3470, 0.0240],
        [0.7797, 0.1519, 0.7513, 0.7269],
        [0.8572, 0.1165, 0.8596, 0.2636]])
tensor([[0.6855, 0.9696, 0.4295, 0.4961],
        [0.3849, 0.0825, 0.7400, 0.0036],
        [0.8104, 0.8741, 0.9729, 0.3821]])
```

Loading Data, Devices and CUDA

Numpy arrays to PyTorch tensors

- `torch.from_numpy(x_train)`
- `Returns a cpu tensor!`

PyTorchtensor to numpy

- `t.numpy()`

Using GPU acceleration

- `t.to()`
- Sends to whatever device (cudaor cpu)

Fallback to cpu if gpu is unavailable:

- `torch.cuda.is_available()`

Check cpu/gpu tensor OR numpyarray ?

- `type(t)` or `t.type()` returns
  - numpy.ndarray
  - torch.Tensor
    - CPU - torch.cpu.FloatTensor
    - GPU - torch.cuda.FloatTensor

Autograd

- Automatic Differentiation Package
- Don't need to worry about partial differentiation, chain rule etc.
  - `backward()` does that
- Gradients are accumulated for each step by default:
  - Need to zero out gradients after each update
  - `tensor.grad_zero()`

```python
# Create tensors.
x = torch.tensor(1., requires_grad=True)
w = torch.tensor(2., requires_grad=True)
b = torch.tensor(3., requires_grad=True)

# Build a computational graph.
y = w * x + b    # y = 2 * x + 3

# Compute gradients.
y.backward()

# Print out the gradients.
print(x.grad)    # x.grad = 2
print(w.grad)    # w.grad = 1
print(b.grad)    # b.grad = 1
```

Optimizer and Loss

Optimizer

- Adam, SGD etc.
- An optimizer takes the parameters we want to update, the learning rate we want to use along with other hyper-parameters and performs the updates

Loss

- Various predefined loss functions to choose from
- L1, MSE, Cross Entropy

```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)


# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    loss.backward()

    optimizer.step()

    optimizer.zero_grad()

print(a, b)
```

Model

In PyTorch, a model is represented by a regular Python class that inherits from the Module class.

- Two components
  - `__init__(self)` : it defines the parts that make up the model- in our case, two parameters, a and b
  - `forward(self, x)` : it performs the actual computation, that is, it outputs a prediction, given the inputx

```
class ManualLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # To make "a" and "b" real parameters of the model, we need to wrap them with nn.Parameter
        self.a = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))
        self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))

    def forward(self, x):
        # Computes the outputs / predictions
        return self.a + self.b * x
```

Model

Two-layer neural network with ReLU activation function, and Sigmoid activation for the output.:

```
class TwoLayerNetwork(nn.Module):
   def __init__(self, input_dim=2,
                hidden_dim=128, output_dim=1):
      self.input_dim = input_dim
      self.hidden_dim = hidden_dim
      self.output_dim = output_dim
      self.layers = nn.Sequential([nn.Linear(input_dim,
                                             hidden_dim),
                                  nn.ReLU(),
                                  nn.Linear(hidden_dim,
                                            output_dim),
                                  nn.Sigmoid()])
   def forward(self, x):
     return self.layers(x)
```

Create custom dataset I

```
class CustomDataset(torch.utils.data.Dataset):
    def __init__(self, file_path, root_dir, transform=None):
        self.data = LOAD_DATA_FUNC(file_path)
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()
```

Create custom dataset II

```
# Load image
img_name = os.path.join(self.root_dir,
                        self.data[idx, 0])

img = io.imread(img_name)
label = self.data[idx, 1]

sample = {'image': img, 'label': label}

if self.transform:
   sample = self.transform(sample)

return sample
```

Example of training I

Safely enable gpu

```
GPU_AVAILABLE = torch.cuda.is_available()
def enable_cuda(x):
   if GPU_AVAILABLE:
      return x.cuda()
   return x
```

Example of training II

Initialization before training

```
dataset = CustomDataset(dataset_path)
loader = CustomDataLoader(dataset, batch_size=32,
                          shuffle=True, num_workers=1)
network = CustomNetwork(input_dim, output_dim, hidden_dim, ...

optimizer = torch.optim.Adam(network.parameters(), lr=lr)
criterion = torch.nn.BCEWIthLogitsLoss()

network = enable_cuda(network)
criterion = enable_cuda(criterion)
```

Training for-loop

```
for epoch in range(n_epoch):
    for data in loader:

        # x, label in data are defined in the custom dataLoader
        predicted_y = network(enable_cuda(data.x.float()))
        target = enable_cuda(data.label.float())

        loss = criterion(predicted_y, target)
        optimizer.zero_grad()

        # Compute gradients for backpropagation
        loadd.backward()

        # Do backpropagation
        optimizer.step()

# Save the network
torch.save(network.state_dict(), path_to_save)
```