Machine Learning

Chapter 8.1: PyTorch

Fall 2022

Instructor: Xiaodong Gu

About PYTORCH

PyTorch is a python package that provides two high-level features:

- Tensor computation (like numpy) with strong GPU acceleration
- Neural Networks built on a tape-based autograd system

Why PYTORCH

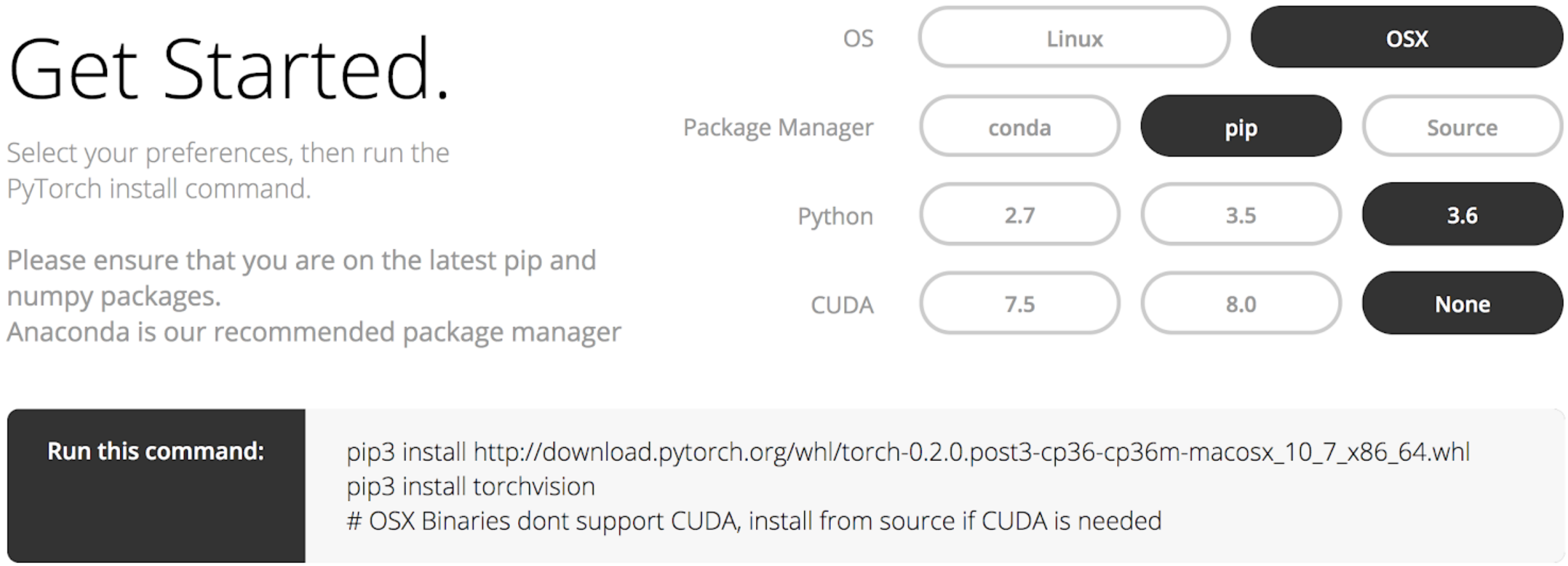
- More Pythonic (imperative)
  - flexible
  - intuitive and cleaner code
  - easy to debug
- More Neural Networks
  - write code as the network works
  - forward/backward

PYTORCH vs Tensorflow

	PyTorch	Tensorflow
Model Definition	Dynamic computational graph	Static graph definition
Debugging	easy for debugging since computation graph is defined at runtime	you won't be able to debug any python code with it
Deployment	use Flask or another alternative to code up a REST API on top of the model.	*TensorFlow Serving* may be a better option if performance is a concern.
Data Parallelism	use `torch.nn.DataParallel` to wrap any module for parallelism over batch.	fine tune every operation to be run on specific device
A framework or a library	provides useful abstractions in certain domain and a convenient way to use them	all operations are pretty low-level

Install PYTORCH

https://pytorch.org



Hello PyTorch

```
09:40 $ python
Python 3.6.2 (v3.6.2:5fd33b5926,Jul 16 2017, 20:11:06
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> import torch
>>> print(torch.__version__)
1.2.0_3
>>> # Happy!!
```

PYTORCH Rhythm

1. **Design your model using class**

$$x \rightarrow \boxed{\text{Model}} \rightarrow \hat{y}$$

2. **Construct loss and optimizer**
   (select from PyTorch API)

3. **Training cycle**
   (forward, backward, update)

How to define data?

Tensors - the basic data structure in Torch.

```
import torch
x = torch.tensor([0.1, 0.2, 0.3], dtype=torch.float)
W = torch.tensor([[0.1, 0.2, 0.3],
                  [0.4, 0.1, 0.7],
                  [0.8, 0.1, 0.2],
                 ])
r = torch.zeros([1,3], dtype=torch.long)
```

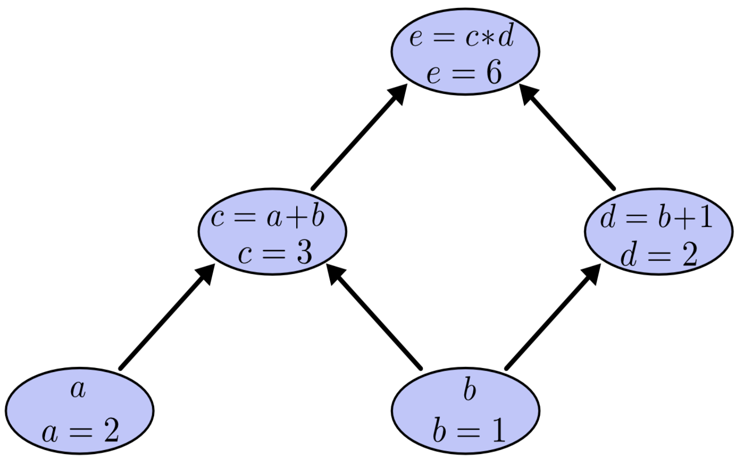
Tensors can also be defined from numpy array.

```
import numpy as np
import torch
x_np = np.array([0.1,0.2,0.3])
x_torch = torch.from_numpy(x_np)
```

Computational Graphs

- PyTorch's **tensor** object implicitly creates a **computation graph** in the background.

$$e = (a+b)\times(b+1)$$




```python
a = torch.tensor(2.0,
requires_grad=True)
# we set requires_grad=True to let
PyTorch know to keep the graph
b = torch.tensor(1.0,
requires_grad=True)
c = a + b
d = b + 1
e = c * d
print(c)
print(d)
print(e)
```

```
c tensor(3., grad_fn=<AddBackward0>)
d tensor(2., grad_fn=<AddBackward0>)
e tensor(6., grad_fn=<MulBackward0>)
```

PyTorch as an auto grad framework

- Now that we have seen that PyTorch keeps the graph around for us, let's use it to compute some **gradients** for us.
- Given $f(x)=(x-2)^2$, we want to compute $\frac{d}{dx}f(x)$ and then compute $f'(1)$.

We make a backward() call on the leaf variable (y) in the computation, computing all the gradients of y at once.

```python
def f(x):
        return (x-2)**2
def fp(x):
        return 2*(x-2)
x = torch.tensor([1.0], requires_grad=True)
y = f(x)
y.backward()
print('Analytical f\'(x):', fp(x))
print('PyTorch\'s f\'(x):', x.grad)
```

```
Analytical f'(x): tensor([-2.], grad_fn=<MulBackward0>)
PyTorch's f'(x): tensor([-2.])
```

PyTorch as an auto grad framework

- It can also find `gradients` of functions.

Let $w=[w_1,w_2]^T$

Consider $g(w)=2w_1w_2+w_2\cos(w_1)$

Q: Compute $\nabla_w g(w)$ and verify $\nabla_w g([\pi,1])=[2,\pi-1]^T$

```
def g(w):
        return 2*w[0]*w[1]+w[1]*torch.cos(w[0])
def grad_g(w):
        return torch.tensor([2*w[1]-w[1]*torch.sin(w[0]),2*w[0]+
torch.cos(w[0])])
w = torch.tensor([np.pi, 1], requires_grad=True)
z = g(w)
z.backward()
print('Analytical grad g(w)', grad_g(w))
print('PyTorch\'s grad g(w)', w.grad)
```

```
Analytical grad g(w) tensor([2.0000, 5.2832])
PyTorch's grad g(w) tensor([2.0000, 5.2832])
```

Define a Machine Learning Model

- A machine Learning model can be defined by inheriting the `torch.nn.Module`.
- You should customize the `forward()` function which defines the computational graph for the model.

```
class Model(torch.nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(1, 1) # One in and one out

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        y_pred = self.linear(x)
        return y_pred

# our model
model = Model()
```

Define a Machine Learning Model

Building the model may require submodules provided by PyTorch such as:

- Basic Modules(e.g., Linear,)
- Functions (e.g., activations,)
- Containers (e.g., sequential, concat,)
- Other Developed Models

Define Loss functions

- PyTorch implements many common loss functions including the `MSELoss` and the `CrossEntropyLoss`.

```
mse_loss_fn = nn.MSELoss()
input = torch.tensor([[0., 0, 0]])
target = torch.tensor([[1., 0, -1]])
loss = mse_loss_fn(input, target)
print(loss)
```

```
tensor(0.6667)
```

Choose an Optimizer

- PyTorch implements a number of gradient-based optimization methods in torch.optim, including Gradient Descent. At the minimum, it takes in the model parameters and a learning rate.
- Optimizers do not compute the gradients for you, so you must call backward() yourself.
- You also must call the optim.zero_grad() function before calling backward() since by default PyTorch does and inplace add to the .grad member variable rather than overwriting it.

Optimizers

```
# create a simple model
model = nn.Linear(1, 1)
# create a simple dataset
X_simple = torch.tensor([[1.]])
y_simple = torch.tensor([[2.]])
# create our optimizer
optim = torch.optim.SGD(model.parameters(), lr=1e-2)
loss_fn = nn.MSELoss()
y_hat = model(X_simple)
print('model params before:', model.weight)
loss = loss_fn(y_hat, y_simple)
optim.zero_grad()
loss.backward()
optim.step()
print('model params after:', model.weight)
```

Dataset

`torch.utils.data.Dataset` is an abstract class representing a dataset. Your custom dataset should inherit Dataset and override the following methods:

- __len__ so that len(dataset) returns the size of the dataset.
- __getitem__ to support the indexing such that dataset[i] can be used to get *i*-th sample

```
class MyDataset(Dataset):
    def __init__(self, csv_file):
        self.sentences = pd.read_csv(csv_file)
    def __len__(self):
        return len(self.sentences)
    def __getitem__(self, idx):
        sample = self.sentences[idx]
        sample = sample.lower()
        return sample
```

Data Loader

To efficiently load data into model (instead of a simple for loop), the torch.utils.data.DataLoader is an iterator which provides:

- batching the data
- shuffling the data
- load the data in parallel using multiprocessing workers.

```
dataloader = DataLoader(my_dataset, batch_size=4,
                        shuffle=True, num_workers=4)
for i_batch, sample_batched in enumerate(dataloader):
    print(i_batch, sample_batched['image'].size())

    ......
```

Training: forward, loss, backward, step

```python
# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```python
for x_val, y_val in zip(x_data, y_data):
        ...
        w.data = w.data - 0.01 * w.grad.data
```

PyTorch forward/backward

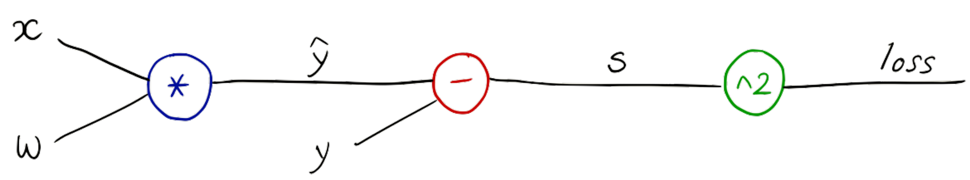
```python
w = torch.Tensor([1.0]) # Any random value

# our model forward pass
def forward(x):
   return x * w

# Loss function
def loss(x, y):
   y_pred = forward(x)
   return (y_pred - y) * (y_pred - y)

# Training loop
for epoch in range(10):
   for x_val, y_val in zip(x_data, y_data):
       l = loss(x_val, y_val)
       l.backward()
       print("\tgrad: ", x_val, y_val, w.grad.data[0])
       w.data = w.data - 0.01 * w.grad.data

       # Manually zero the gradients after updating weights
       w.grad.data.zero_()
   print("progress:", epoch, l.data[0])
```



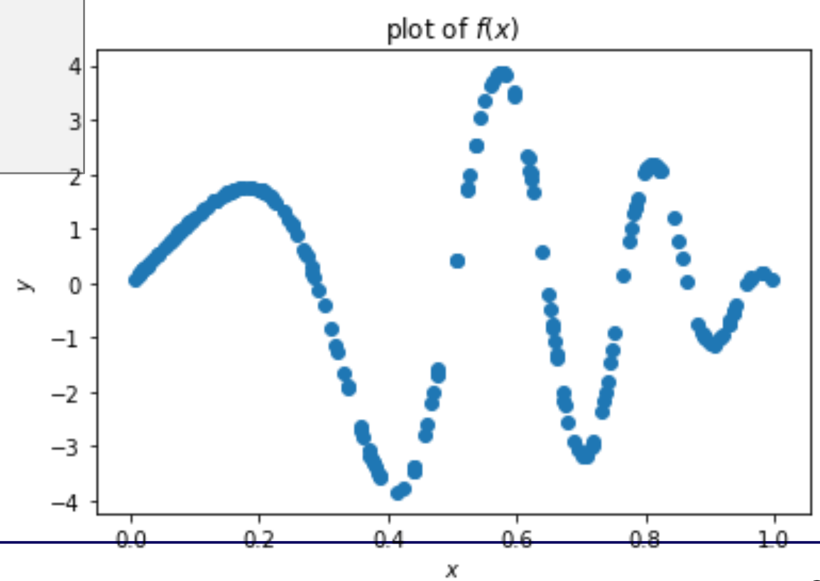
Hello World

Implementing an MLP using PyTorch

MLP using PyTorch

- **Create a simple dataset**

```
d = 1
n = 200
X = torch.rand(n,d)
y = 4 * torch.sin(np.pi * X) *
torch.cos(6*np.pi*X**2)
plt.scatter(X.numpy(), y.numpy())
plt.title('plot of $f(x)$’)
plt.xlabel('$x$’)
plt.ylabel('$y$’)
plt.show()
```



MLP using PyTorch

- **define the model**

```
n_hidden_1 = 32
n_hidden_2 = 32
d_out = 1

neural_network = nn.Sequential(
        nn.Linear(d, n_hidden_1),
        nn.Tanh(),
        nn.Linear(n_hidden_1, n_hidden_2),
        nn.Tanh(),
        nn.Linear(n_hidden_2, d_out)
)
```

MLP using PyTorch

- **Training**

```
step_size = 0.05
n_epochs = 6000
loss_func = nn.MSELoss()
optim = torch.optim.SGD(
   neural_network.parameters(), lr=step_size
)
print(‘iter, \tloss’)
for i in range(n_epochs):
    y_hat = neural_network(X)
    loss = loss_func(y_hat, y)
    optim.zero_grad()
    loss.backward()
    optim.step()
    if i % (n_epochs // 10) == 0:
         print(f’{i},{loss.item()}'))
```

```
iter, loss
   0, 3.96
 600, 3.69
1200, 2.58
1800, 1.10
2400, 0.85
3000, 0.60
3600, 0.14
4200, 0.08
4800, 0.06
5400, 0.24
```

MLP using PyTorch

- **Prediction and Visualization**

```
X_grid =
torch.from_numpy(np.linspace(0,1,50)).float().view(-1, d)
y_hat = neural_network(X_grid)
plt.scatter(X_numpy(), y_numpy())
plt.plot(X_gr                                   ).numpy(),
'r’)
plt.title('pl
plt.xlabel('$
plt.ylabel('$
plt.show()
```