Machine Learning

Lecture 4

Machine Learning Libraries

Pytorch

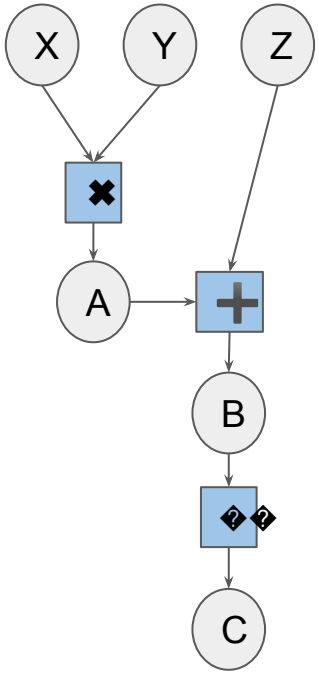
- Open source machine learning library
- Developed by Facebook's AI Research lab
- It leverages the power of GPUs
- Automatic computation of gradients
- Makes it easier to test and develop new ideas.

Why PyTorch ?

- It is pythonic - concise, close to Python conventions
- Strong GPU support Autograd - automatic differentiation
- Many algorithms and components are already implemented
- Similar to NumPy

Why PyTorch ?

Computation Graph



NumPy

```
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
```

PyTorch

```
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()
```

Getting Started With PyTorch

Installation using Anaconda :

```
>>> conda install pytorch torchvision torchaudio pytorch-cuda=11.7 -c pytorch -c nvidia
```

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)

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()`

```
# 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
```

Optimizers and Loss Functions

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 `nn.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
```