Deep Learning Software
PyTorch

Department of Computer Science, NYCU

TA 賴佑家

Some slides are from Stanford CS231n

Frameworks

TensorFlow

PyTorch

PaddlePaddle

Microsoft CNTK

Keras

torch

Chainer

Caffe2

DL4J

mxnet

Caffe

And others……

Frameworks

Most people use these

TensorFlow

PyTorch

PaddlePaddle

Microsoft CNTK

Keras

torch

Chainer

Caffe2

DL4J

Caffe

mxnet

And others……

Frameworks

We will focus on this

PyTorch

TensorFlow

PaddlePaddle

Microsoft CNTK

Keras

torch

Chainer

Caffe2

DL4J

mxnet

Caffe

And others……

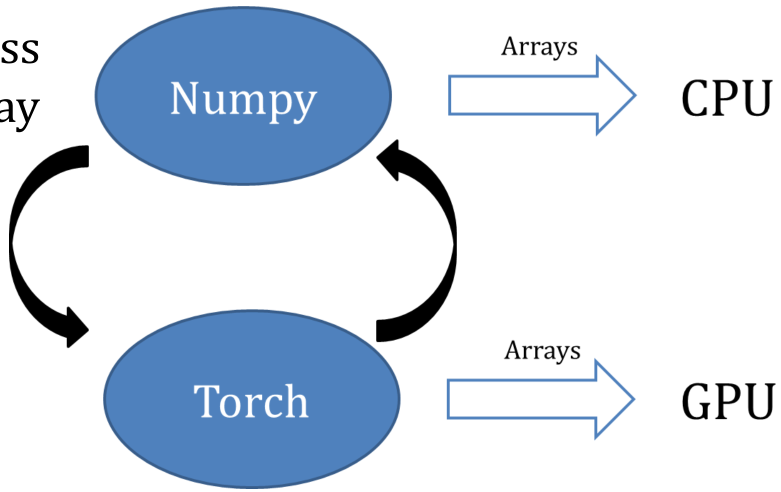
The advantages of deep learning frameworks

- Developing and testing new ideas are quickly
- Computing gradients automatically
- Running model structures on GPU is efficiently

Please use PyTorch to complete all your assignments !!

PyTorch Fundamental Concepts

- **Tensor** : torch.Tensor is the central class of the package. It is like a numpy array but can run on GPU.



- **Autograd** : The autograd package provides automatic differentiation for all operations on Tensors.

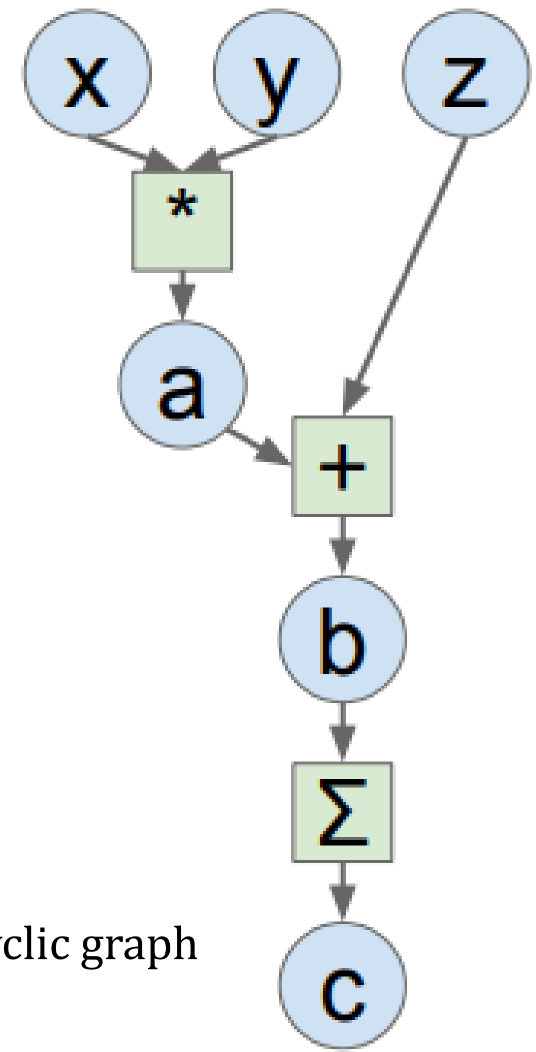
- **Module** : A neural network layer stores both state and learnable weight.

Computational Graphs

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)
```

$x \times y + z$



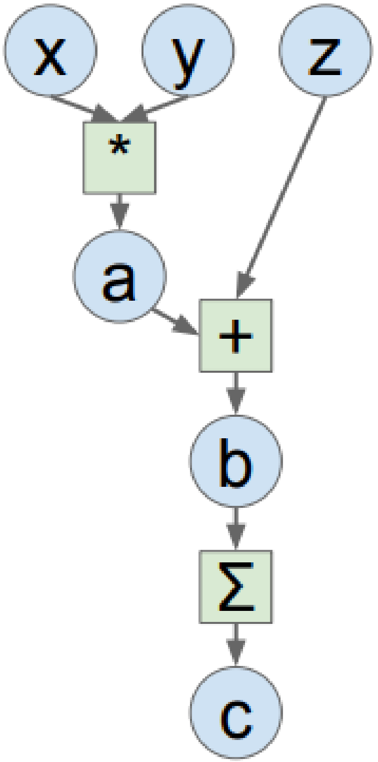
Neural network can be denoted as a directed acyclic graph

Computational Graphs

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
```

compute gradients



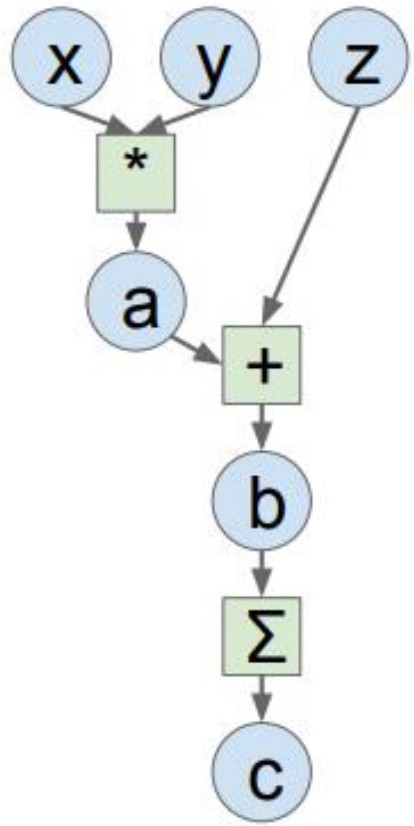
Problems:
- Can’t run on GPU
- Have to compute our own gradients

Computational Graphs

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

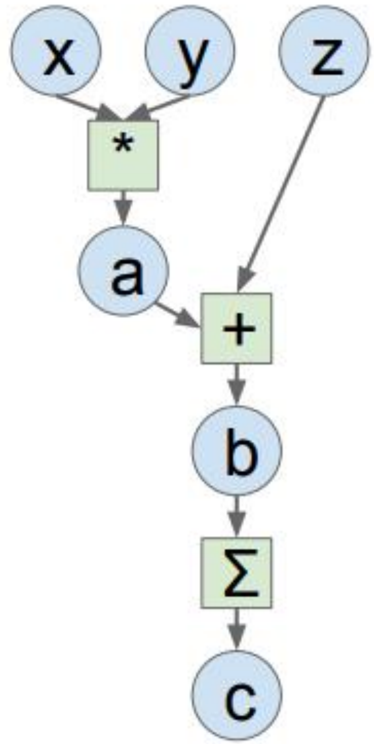
```
import torch

N, D = 3, 4
x = torch.randn(N, D)
y = torch.randn(N, D)
z = torch.randn(N, D)

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

Looks exactly like numpy!

Computational Graphs

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

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

c.backward()
print(x.grad)
```

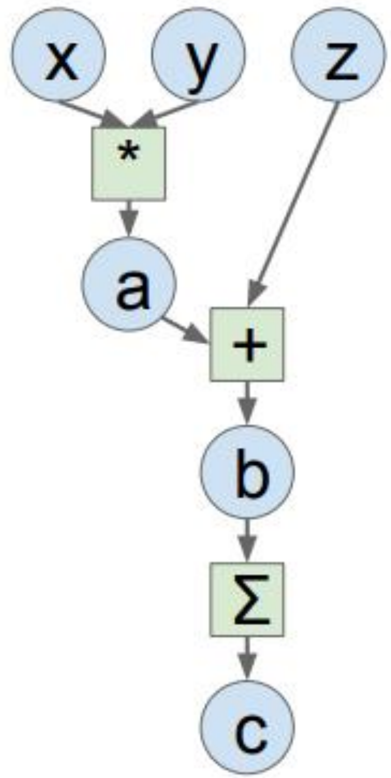
PyTorch handles gradients for us!

.backward()   compute gradient

Computational Graphs

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
```



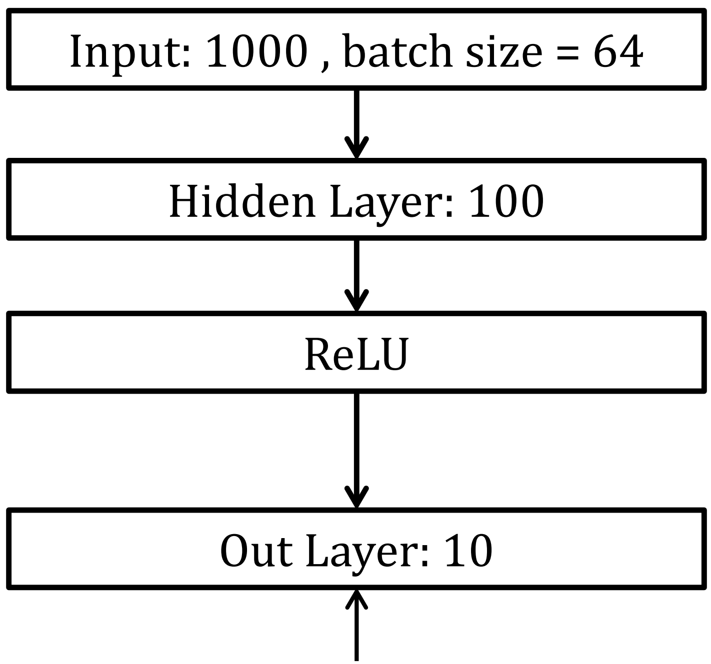
PyTorch

```
import torch

device = 'cuda:0'
N, D = 3, 4
x = torch.randn(N, D, requires_grad=True,
                device=device)
y = torch.randn(N, D, device=device)
z = torch.randn(N, D, device=device)

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

c.backward()
print(x.grad)
```

Trivial to run on GPU - just construct arrays on a different device!

Example

- 2-layer network



x: 64×1000

w1: 1000×100

$ReLU(x) = \begin{cases} x \ if \ x > 0 \\ 0 \ if \ x \le 0 \end{cases}$

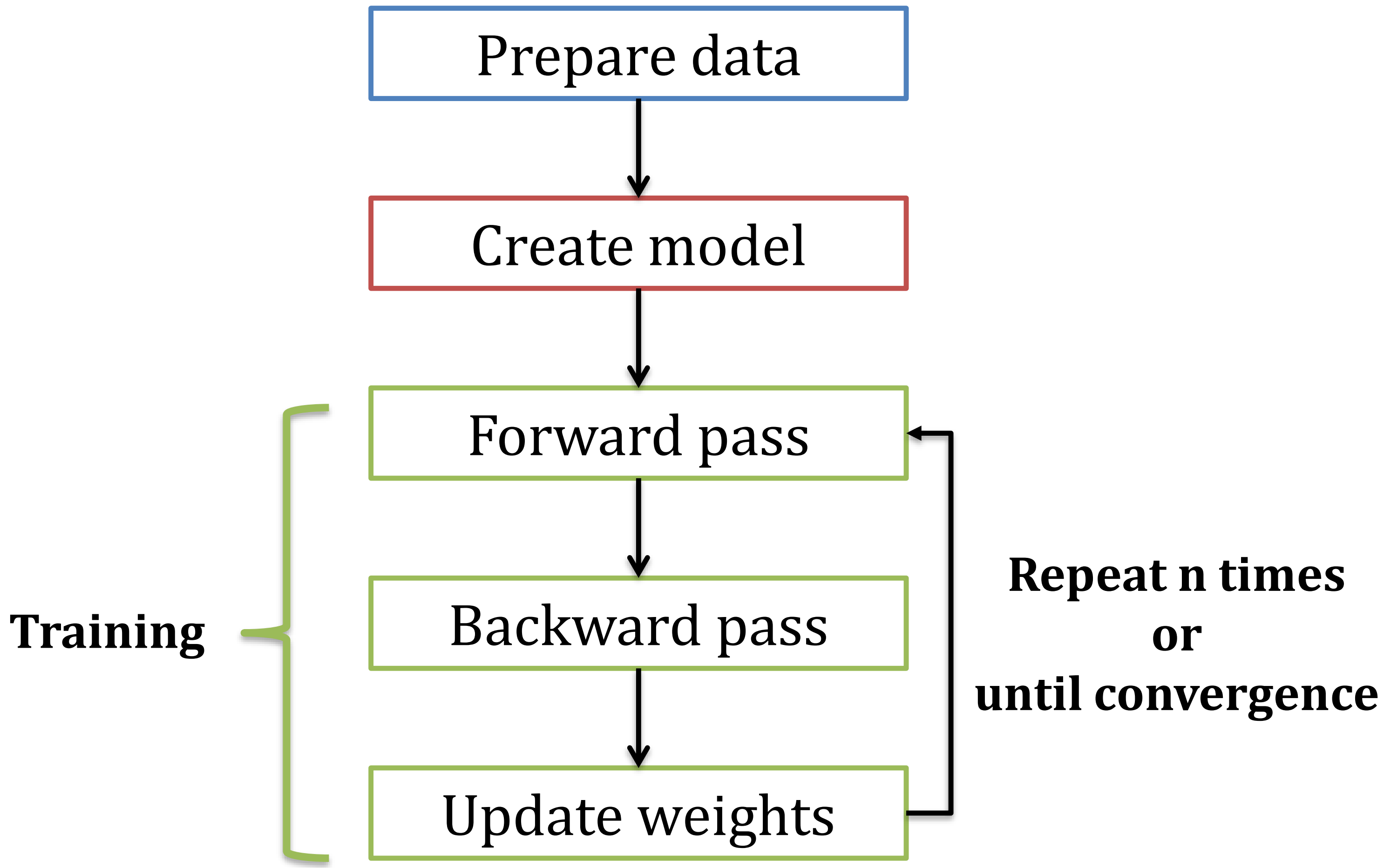
w2: 100×10

Loss : sum of square error

$$\sum (\boldsymbol{y} - \widehat{\boldsymbol{y}})^{\mathbf{2}}$$

y : 64×10

Flow Chart


Prepare data
Create model
Forward pass
Backward pass
Update weights
Training
Repeat n times
or
until convergence

Easily implement your own deep learning model by using **PyTorch**

Step1. Prepare Data
PyTorch.utils.data

DataLoader wraps a **Dataset** and provides minibatches, shuffling, multithreading, for you

When you need to load custom data, just write your own Dataset class

Iterate over loader to form mini-batchs

https://github.com/utkuozbulak/pytorch-custom-dataset-examples

```python
import torch
from torch.utils.data import TensorDataset, DataLoader

device = torch.device('cpu')
learning_rate = 1e-2

x = torch.randn(64, 1000, device=device)
y = torch.randn(64, 10, device=device)
loader = DataLoader(TensorDataset(x, y), batch_size=8)

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear_1 = torch.nn.Linear(D_in, H)
        self.linear_2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h = self.linear_1(x)
        h_relu = torch.nn.functional.relu(h)
        y_pred = self.linear_2(h_relu)
        return y_pred

model = TwoLayerNet(D_in=1000, H=100, D_out=10)
model = model.to(device)

optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for epochs in range(50):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)
        print(loss.item())

        loss.backward()

        optimizer.step()
        optimizer.zero_grad()
```

Step2. Create Model
PyTorch.nn

Higher-level wrapper for working with neural nets

Use this ! It will make your life easier

A PyTorch Module is a neural net layer, it can contain weights or other modules

Define your whole model as a single module

```python
import torch
from torch.utils.data import TensorDataset, DataLoader

device = torch.device('cpu')
learning_rate = 1e-2

x = torch.randn(64, 1000, device=device)
y = torch.randn(64, 10, device=device)
loader = DataLoader(TensorDataset(x, y), batch_size=8)

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear_1 = torch.nn.Linear(D_in, H)
        self.linear_2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h = self.linear_1(x)
        h_relu = torch.nn.functional.relu(h)
        y_pred = self.linear_2(h_relu)
        return y_pred

model = TwoLayerNet(D_in=1000, H=100, D_out=10)
model = model.to(device)

optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for epochs in range(50):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)
        print(loss.item())

        loss.backward()

        optimizer.step()
        optimizer.zero_grad()
```

Step2. Create Model
PyTorch.nn

Initializer sets up two children (Module can contain Modules)

```python
import torch
from torch.utils.data import TensorDataset, DataLoader

device = torch.device('cpu')
learning_rate = 1e-2

x = torch.randn(64, 1000, device=device)
y = torch.randn(64, 10, device=device)
loader = DataLoader(TensorDataset(x, y), batch_size=8)

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear_1 = torch.nn.Linear(D_in, H)
        self.linear_2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h = self.linear_1(x)
        h_relu = torch.nn.functional.relu(h)
        y_pred = self.linear_2(h_relu)
        return y_pred

model = TwoLayerNet(D_in=1000, H=100, D_out=10)
model = model.to(device)

optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for epochs in range(50):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)
        print(loss.item())

        loss.backward()

        optimizer.step()
        optimizer.zero_grad()
```

Step2. Create Model
PyTorch.nn

Define forward pass using child modules

No need to define backward – autograd will handle it

```
import torch
from torch.utils.data import TensorDataset, DataLoader

device = torch.device('cpu')
learning_rate = 1e-2

x = torch.randn(64, 1000, device=device)
y = torch.randn(64, 10, device=device)
loader = DataLoader(TensorDataset(x, y), batch_size=8)

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear_1 = torch.nn.Linear(D_in, H)
        self.linear_2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h = self.linear_1(x)
        h_relu = torch.nn.functional.relu(h)
        y_pred = self.linear_2(h_relu)
        return y_pred

model = TwoLayerNet(D_in=1000, H=100, D_out=10)
model = model.to(device)

optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for epochs in range(50):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)
        print(loss.item())

        loss.backward()

        optimizer.step()
        optimizer.zero_grad()
```

Step3. Forward pass
PyTorch.nn

Define forward pass using child modules

Feed data to model, and compute loss

nn.functional has useful helpers like loss functions

```python
import torch
from torch.utils.data import TensorDataset, DataLoader

device = torch.device('cpu')
learning_rate = 1e-2

x = torch.randn(64, 1000, device=device)
y = torch.randn(64, 10, device=device)
loader = DataLoader(TensorDataset(x, y), batch_size=8)

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear_1 = torch.nn.Linear(D_in, H)
        self.linear_2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h = self.linear_1(x)
        h_relu = torch.nn.functional.relu(h)
        y_pred = self.linear_2(h_relu)
        return y_pred

model = TwoLayerNet(D_in=1000, H=100, D_out=10)
model = model.to(device)

optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for epochs in range(50):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)
        print(loss.item())

        loss.backward()

        optimizer.step()
        optimizer.zero_grad()
```

Step4. Backward pass
PyTorch.autograd

Forward pass looks exactly the same as before, but we don't need to track intermediate values

PyTorch keeps track of them for us in the computational graph

Compute gradient of loss with respect to all model weights (they have requires_grad=True)

```
import torch
from torch.utils.data import TensorDataset, DataLoader

device = torch.device('cpu')
learning_rate = 1e-2

x = torch.randn(64, 1000, device=device)
y = torch.randn(64, 10, device=device)
loader = DataLoader(TensorDataset(x, y), batch_size=8)

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear_1 = torch.nn.Linear(D_in, H)
        self.linear_2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h = self.linear_1(x)
        h_relu = torch.nn.functional.relu(h)
        y_pred = self.linear_2(h_relu)
        return y_pred

model = TwoLayerNet(D_in=1000, H=100, D_out=10)
model = model.to(device)

optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for epochs in range(50):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)
        print(loss.item())

        loss.backward()

        optimizer.step()
        optimizer.zero_grad()
```

Step5. Update Weights

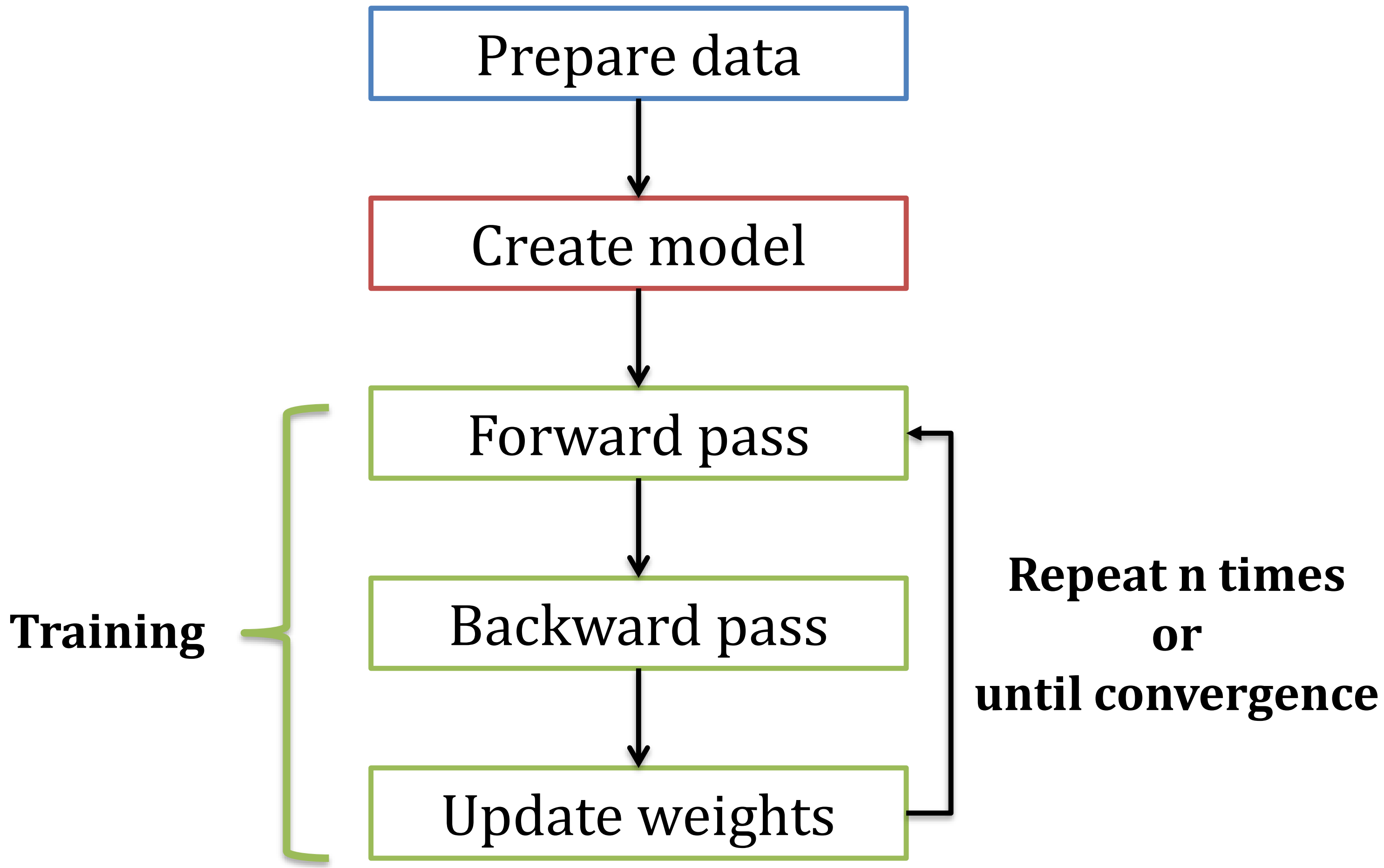
PyTorch.optim

Use an **optimizer** for different update rules

After computing gradients, use optimizer to update each model parameters and reset gradients

```
import torch
from torch.utils.data import TensorDataset, DataLoader

device = torch.device('cpu')
learning_rate = 1e-2

x = torch.randn(64, 1000, device=device)
y = torch.randn(64, 10, device=device)
loader = DataLoader(TensorDataset(x, y), batch_size=8)

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear_1 = torch.nn.Linear(D_in, H)
        self.linear_2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h = self.linear_1(x)
        h_relu = torch.nn.functional.relu(h)
        y_pred = self.linear_2(h_relu)
        return y_pred

model = TwoLayerNet(D_in=1000, H=100, D_out=10)
model = model.to(device)

optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for epochs in range(50):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)
        print(loss.item())

        loss.backward()

        optimizer.step()
        optimizer.zero_grad()
```

Flow Chart


Prepare data
Create model
Forward pass
Backward pass
Update weights
Training
Repeat n times
or
until convergence

Print model

```
class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear_1 = torch.nn.Sequential(
                        torch.nn.Linear(D_in, H),
                        torch.nn.ReLU()
                        )
        self.linear_2 = torch.nn.Linear(H, D_out)
    def forward(self, x):
        h = self.linear_1(x)
        y_pred = self.linear_2(h_relu)
        return y_pred

model = TwoLayerNet(D_in=1000, H=100, D_out=10)
print (model)
```

```
TwoLayerNet(
  (linear_1): Sequential(
    (0): Linear(in_features=1000, out_features=100, bias=True)
    (1): ReLU()
  )
  (linear_2): Linear(in_features=100, out_features=10, bias=True)
)
```

Save and Load model

Saving & Loading Model for Inference

Save:

```
torch.save(model.state_dict(), PATH)
```

Load:

```
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
model.eval()
```

Save/Load Entire Model

Save:

```
torch.save(model, PATH)
```

Load:

```
# Model class must be defined somewhere
model = torch.load(PATH)
model.eval()
```

Real Application

- MNIST example for PyTorch

- https://github.com/pytorch/examples/tree/master/mnist

Build and train a CNN classifier

- Data Loader
- Define Network
- Define Optimizer/Loss function
- Learning rate scheduling
- Training
- Testing
- Run and Save model

Set hyperparameters

```
# Training settings
parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                    help='input batch size for training (default: 64)')
parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                    help='input batch size for testing (default: 1000)')
parser.add_argument('--epochs', type=int, default=14, metavar='N',
                    help='number of epochs to train (default: 14)')
parser.add_argument('--lr', type=float, default=1.0, metavar='LR',
                    help='learning rate (default: 1.0)')
parser.add_argument('--gamma', type=float, default=0.7, metavar='M',
                    help='Learning rate step gamma (default: 0.7)')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='disables CUDA training')
parser.add_argument('--dry-run', action='store_true', default=False,
                    help='quickly check a single pass')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')
parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                    help='how many batches to wait before logging training status')
parser.add_argument('--save-model', action='store_true', default=False,
                    help='For Saving the current Model')
args = parser.parse_args()
```

Data Loader

- Pytorch offers data loaders for popular dataset

The following datasets are available:

Datasets

- MNIST
- COCO
  - Captions
  - Detection
- LSUN
- ImageFolder
- Imagenet-12
- CIFAR
- STL10
- SVHN
- PhotoTour

Data Loader

```
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
        ])
    dataset1 = datasets.MNIST('../data', train=True, download=True,
                       transform=transform)
    dataset2 = datasets.MNIST('../data', train=False,
                       transform=transform)
    train_loader = torch.utils.data.DataLoader(dataset1,**train_kwargs)
    test_loader = torch.utils.data.DataLoader(dataset2, **test_kwargs)
```

Define Network

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

Define Optimizer/Loss function

- **Negative log likelihood loss**
- **Adadelta**

```
        loss = F.nll_loss(output, target)
```

```
    optimizer = optim.Adadelta(model.parameters(), lr=args.lr)
```

Learning rate scheduling

```
scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
```

Training

```
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
            if args.dry_run:
                break
```

Testing

```
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

Run and Save model

```
    for epoch in range(1, args.epochs + 1):
        train(args, model, device, train_loader, optimizer, epoch)
        test(model, device, test_loader)
        scheduler.step()

    if args.save_model:
        torch.save(model.state_dict(), "mnist_cnn.pt")
```

Q&A