Deep Learning Software
PyTorch

Department of Computer Science, NYCU

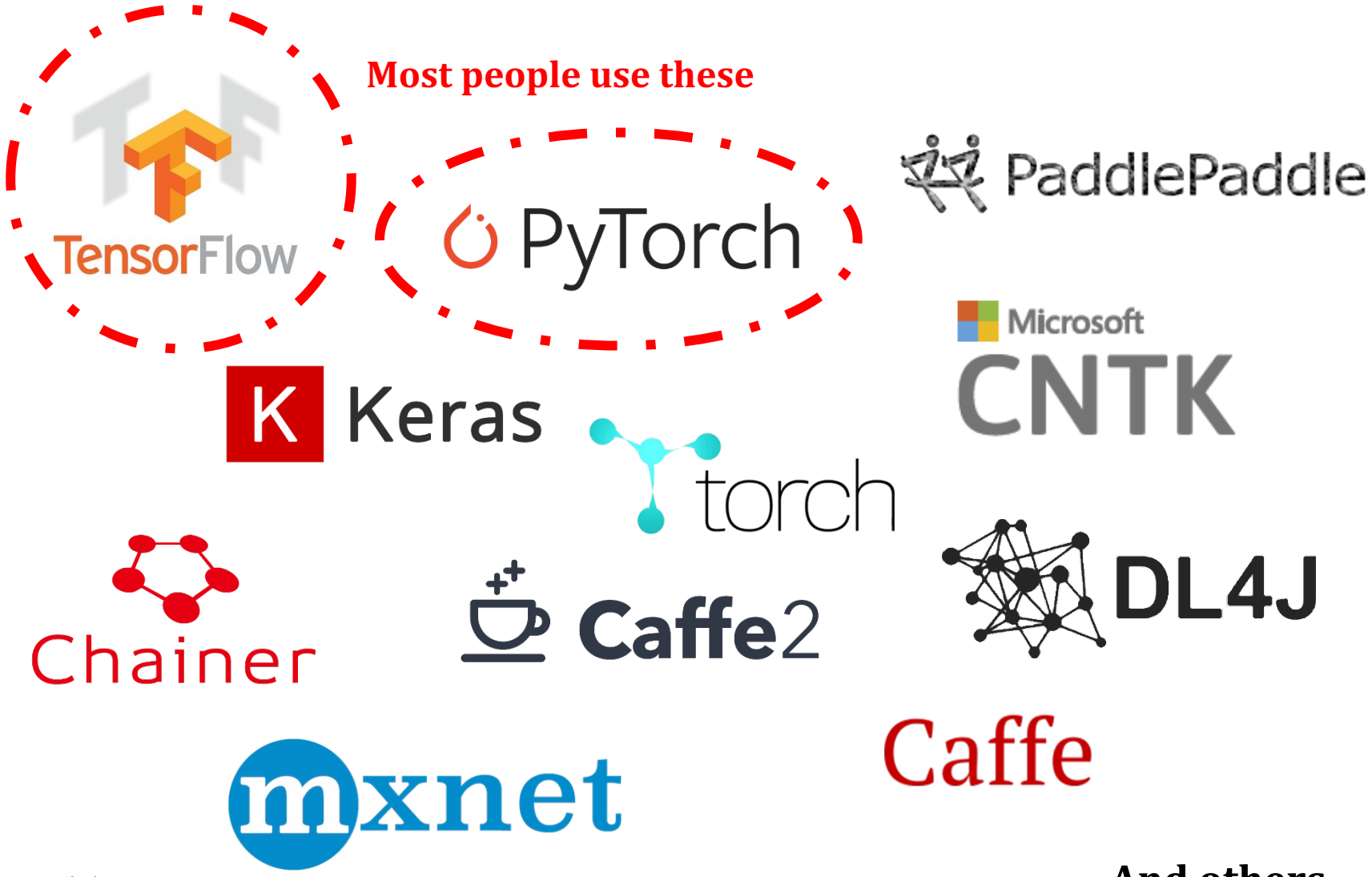
TA 劉子齊 Jonathan

Reference: Stanford CS231n

Frameworks

And others…

Frameworks


Most people use these
TensorFlow
PyTorch
PaddlePaddle
Microsoft
CNTK
Keras
torch
Chainer
Caffe2
DL4J
mxnet
Caffe
And others…

Frameworks

We will focus on this

And others…

Advantages of PyTorch

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

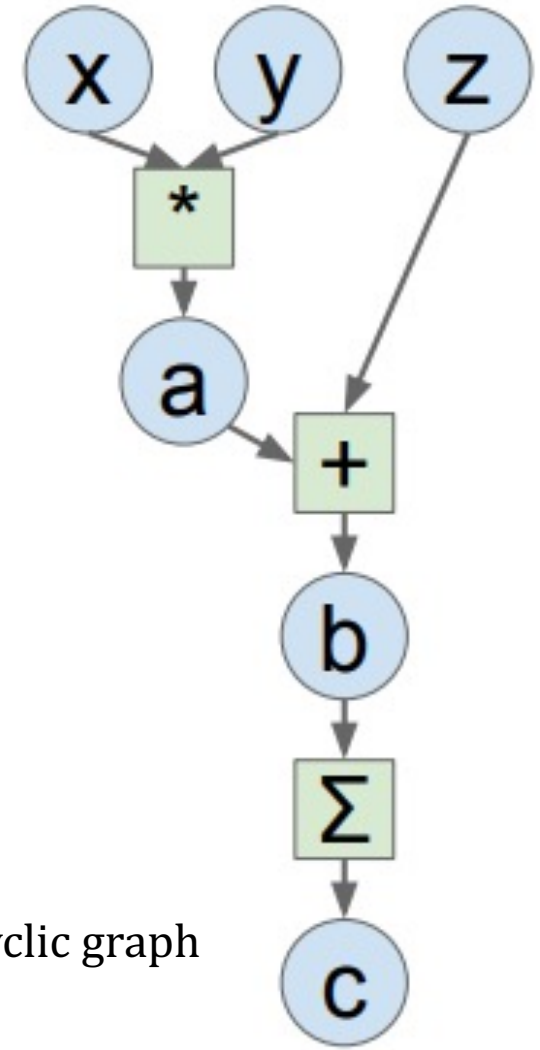
Please use PyTorch in this course !!

Computational Graphs

$x \times y + z$

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)
```



Neural network can be denoted as a directed acyclic graph

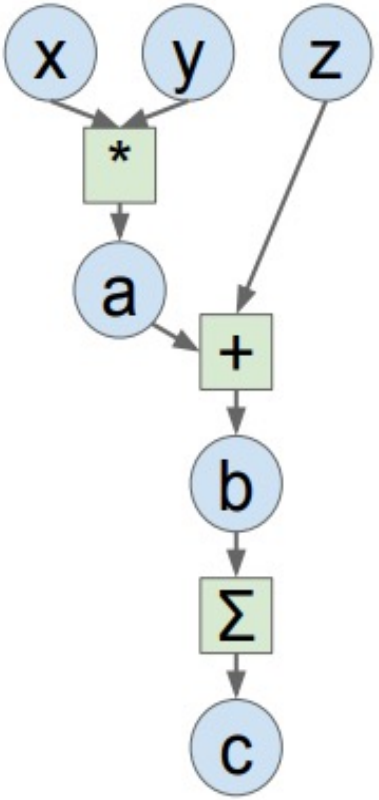
Computational Graphs

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
```

compute gradients



Problems:

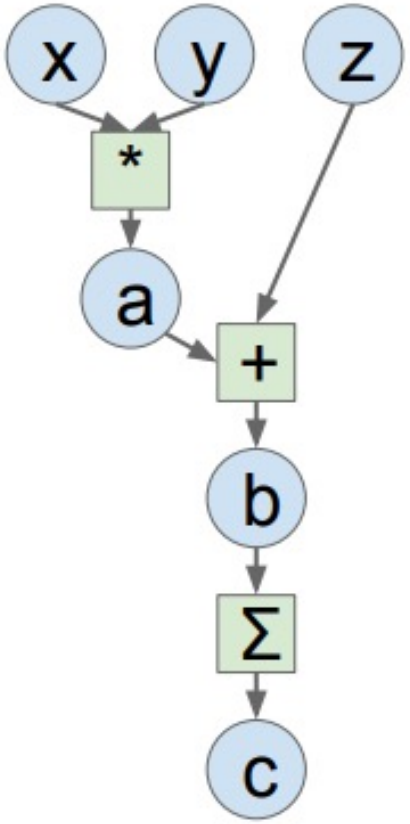
- Can't run on GPU
- Have to compute our own gradients

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

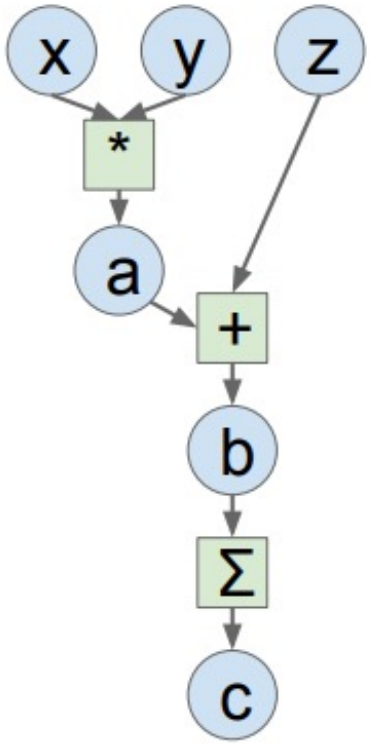
```
import torch

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)
```

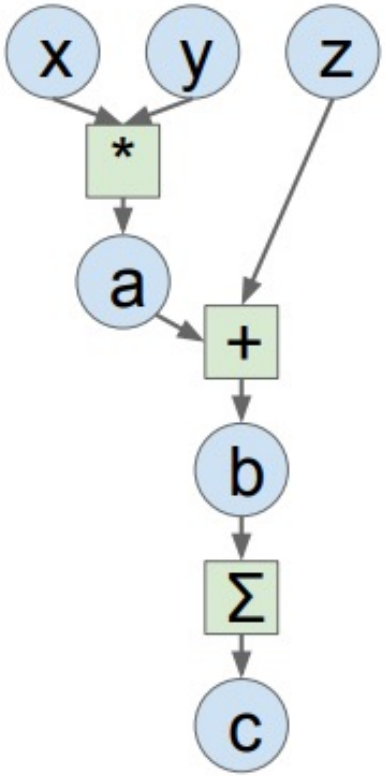
PyTorch handles gradients for us!

.backward() computes the 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
```



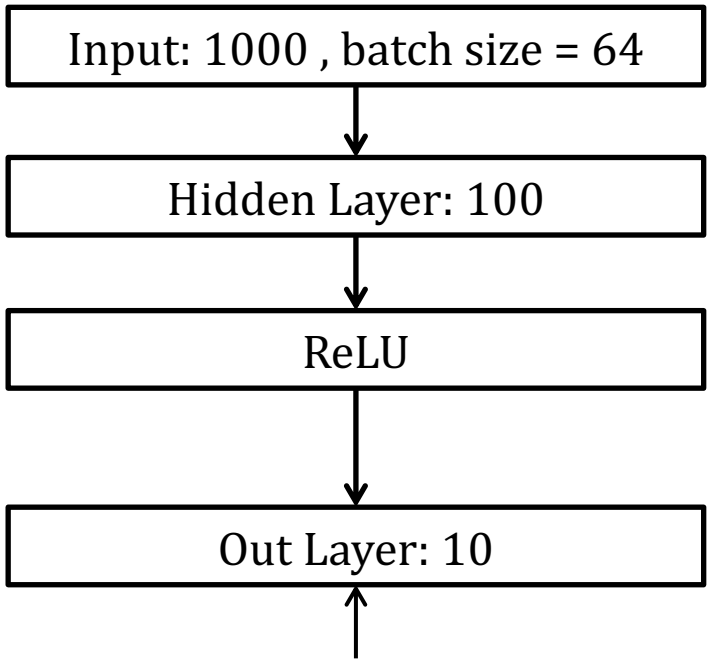
PyTorch

```
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

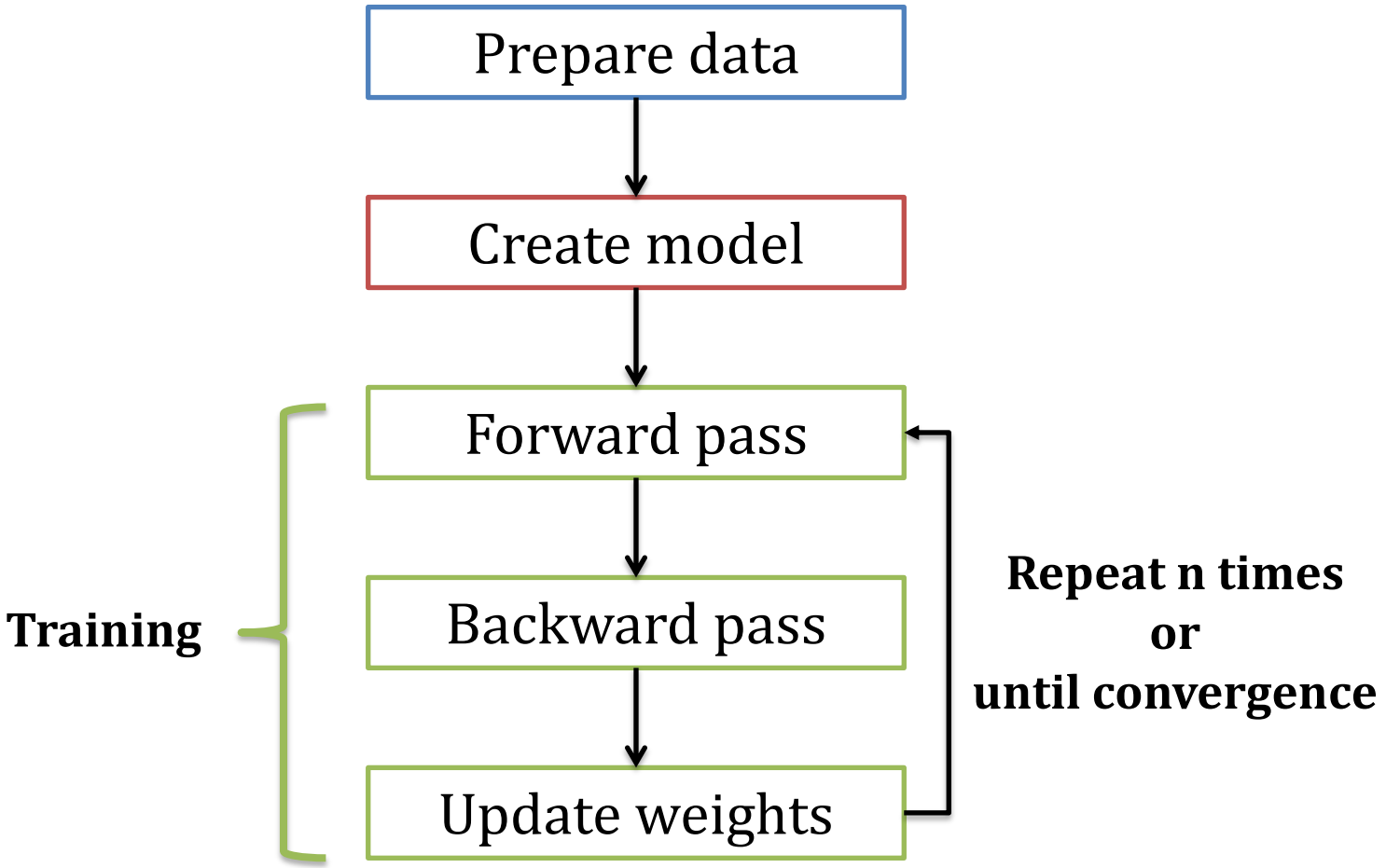
w1: 1000×100

$ReLU(x)=\begin{cases}x\ if\ x>0\\0\ if\ x\le0\end{cases}$

w2: 100×10

y：64×10

Flow Chart

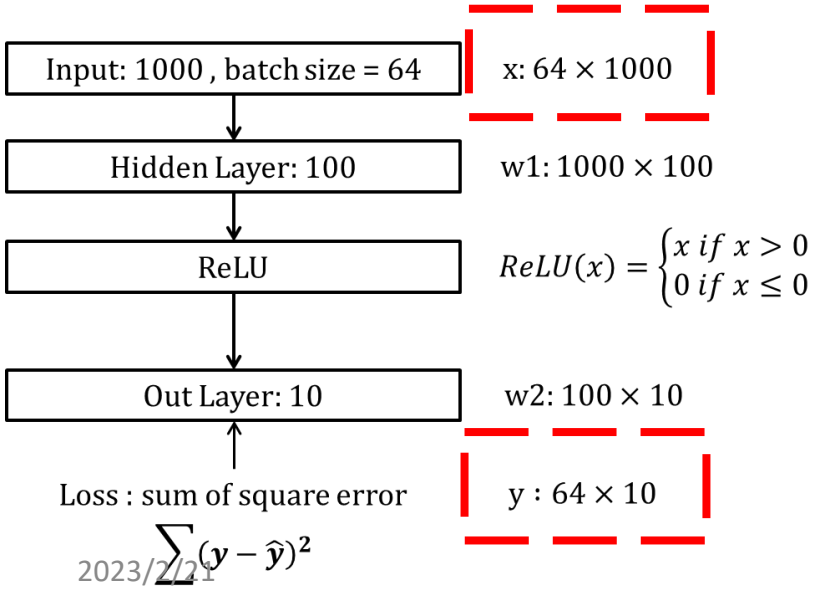

Prepare data
Create model
Forward pass
Backward pass
Update weights
Training
Repeat n times
or
until convergence

Step1. Prepare Data
PyTorch Tensors

Create random tensors as input and ground truth

To run on GPU, just use a different device, like a following:

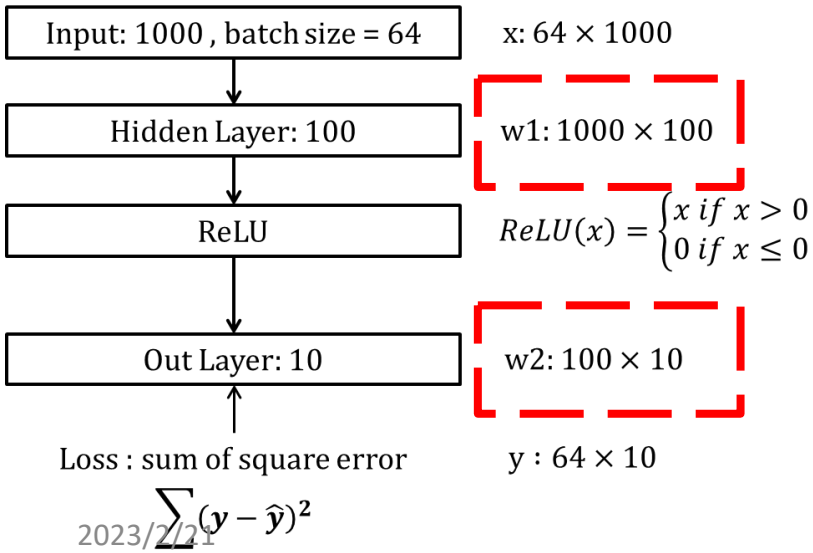
```
device = torch.device('cuda:0')
```




```
import torch

device = torch.device('cpu')
learning_rate = 1e-6

x = torch.randn(64, 1000, device=device)
y = torch.randn(64, 10, device=device)

w1 = torch.randn(1000, 100, device=device)
w2 = torch.randn(100, 10, device=device)

for t in range(300):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y)

    grad_y_pred = 2.0 * loss
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2

    print(loss.pow(2).sum())
```

Step2. Create Model
PyTorch Tensors

Create random tensors as layer weights

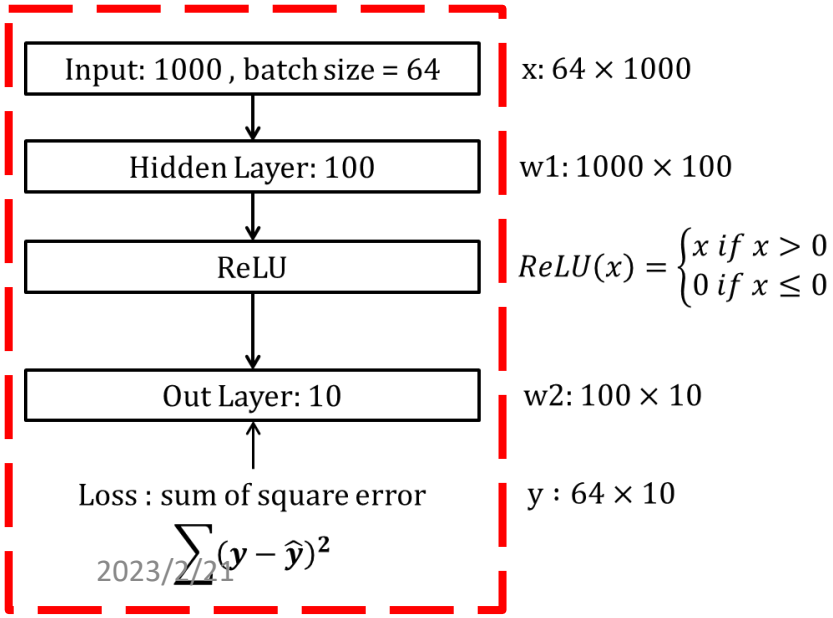



```python
import torch

device = torch.device('cpu')
learning_rate = 1e-6

x = torch.randn(64, 1000, device=device)
y = torch.randn(64, 10, device=device)

w1 = torch.randn(1000, 100, device=device)
w2 = torch.randn(100, 10, device=device)

for t in range(300):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y)

    grad_y_pred = 2.0 * loss
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2

    print(loss.pow(2).sum())
```

Step3. Forward pass

PyTorch Tensors

Compute predictions and loss

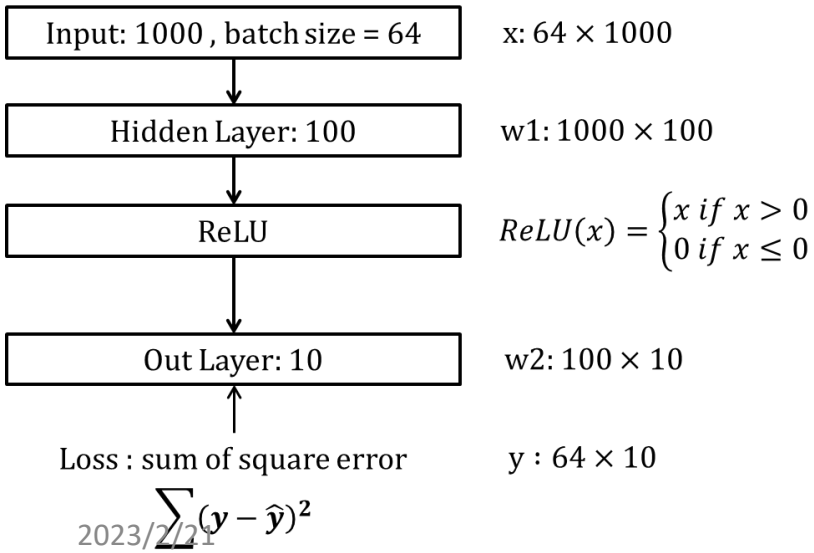



```python
import torch

device = torch.device('cpu')
learning_rate = 1e-6

x = torch.randn(64, 1000, device=device)
y = torch.randn(64, 10, device=device)

w1 = torch.randn(1000, 100, device=device)
w2 = torch.randn(100, 10, device=device)

for t in range(300):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y)

    grad_y_pred = 2.0 * loss
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2

    print(loss.pow(2).sum())
```

Step4. Backward pass

PyTorch Tensors

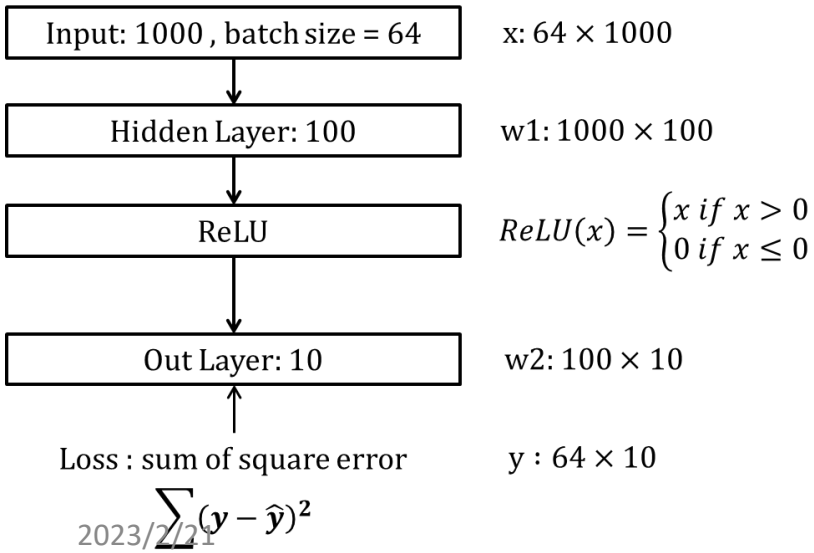
Manually compute gradients




```python
import torch

device = torch.device('cpu')
learning_rate = 1e-6

x = torch.randn(64, 1000, device=device)
y = torch.randn(64, 10, device=device)

w1 = torch.randn(1000, 100, device=device)
w2 = torch.randn(100, 10, device=device)

for t in range(300):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y)

    grad_y_pred = 2.0 * loss
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2

    print(loss.pow(2).sum())
```

Step5. Update Weights
PyTorch Tensors

Gradient descent step on weights




```python
import torch

device = torch.device('cpu')
learning_rate = 1e-6

x = torch.randn(64, 1000, device=device)
y = torch.randn(64, 10, device=device)

w1 = torch.randn(1000, 100, device=device)
w2 = torch.randn(100, 10, device=device)

for t in range(300):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y)

    grad_y_pred = 2.0 * loss
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2

    print(loss.pow(2).sum())
```

Implement your own DL 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

```
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

```
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

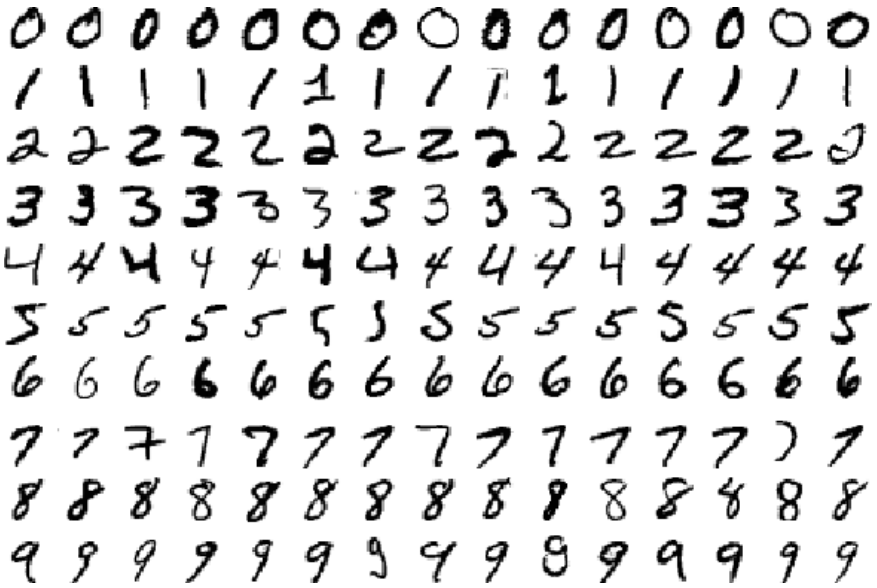
Use an **optimizer** for different update rules

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

```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()
```

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

```
74      # Training settings
75      parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
76      parser.add_argument('--batch-size', type=int, default=64, metavar='N',
77                          help='input batch size for training (default: 64)')
78      parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
79                          help='input batch size for testing (default: 1000)')
80      parser.add_argument('--epochs', type=int, default=14, metavar='N',
81                          help='number of epochs to train (default: 14)')
82      parser.add_argument('--lr', type=float, default=1.0, metavar='LR',
83                          help='learning rate (default: 1.0)')
84      parser.add_argument('--gamma', type=float, default=0.7, metavar='M',
85                          help='Learning rate step gamma (default: 0.7)')
86      parser.add_argument('--no-cuda', action='store_true', default=False,
87                          help='disables CUDA training')
88      parser.add_argument('--dry-run', action='store_true', default=False,
89                          help='quickly check a single pass')
90      parser.add_argument('--seed', type=int, default=1, metavar='S',
91                          help='random seed (default: 1)')
92      parser.add_argument('--log-interval', type=int, default=10, metavar='N',
93                          help='how many batches to wait before logging training status')
94      parser.add_argument('--save-model', action='store_true', default=False,
95                          help='For Saving the current Model')
96      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

```
112     transform=transforms.Compose([
113         transforms.ToTensor(),
114         transforms.Normalize((0.1307,), (0.3081,))
115         ])
116     dataset1 = datasets.MNIST('../data', train=True, download=True,
117                         transform=transform)
118     dataset2 = datasets.MNIST('../data', train=False,
119                         transform=transform)
120     train_loader = torch.utils.data.DataLoader(dataset1,**train_kwargs)
121     test_loader = torch.utils.data.DataLoader(dataset2, **test_kwargs)
```

Define Network

```
11  class Net(nn.Module):
12      def __init__(self):
13          super(Net, self).__init__()
14          self.conv1 = nn.Conv2d(1, 32, 3, 1)
15          self.conv2 = nn.Conv2d(32, 64, 3, 1)
16          self.dropout1 = nn.Dropout(0.25)
17          self.dropout2 = nn.Dropout(0.5)
18          self.fc1 = nn.Linear(9216, 128)
19          self.fc2 = nn.Linear(128, 10)
```

```
21      def forward(self, x):
22          x = self.conv1(x)
23          x = F.relu(x)
24          x = self.conv2(x)
25          x = F.relu(x)
26          x = F.max_pool2d(x, 2)
27          x = self.dropout1(x)
28          x = torch.flatten(x, 1)
29          x = self.fc1(x)
30          x = F.relu(x)
31          x = self.dropout2(x)
32          x = self.fc2(x)
33          output = F.log_softmax(x, dim=1)
34          return output
```

Define Optimizer/Loss function

- Negative log likelihood loss

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

- Adadelta

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

Learning rate scheduling

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

Training

```
37  def train(args, model, device, train_loader, optimizer, epoch):
38      model.train()
39      for batch_idx, (data, target) in enumerate(train_loader):
40          data, target = data.to(device), target.to(device)
41          optimizer.zero_grad()
42          output = model(data)
43          loss = F.nll_loss(output, target)
44          loss.backward()
45          optimizer.step()
46          if batch_idx % args.log_interval == 0:
47              print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
48                  epoch, batch_idx * len(data), len(train_loader.dataset),
49                  100. * batch_idx / len(train_loader), loss.item()))
50              if args.dry_run:
51                  break
```

Testing

```
54  def test(model, device, test_loader):
55      model.eval()
56      test_loss = 0
57      correct = 0
58      with torch.no_grad():
59          for data, target in test_loader:
60              data, target = data.to(device), target.to(device)
61              output = model(data)
62              test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch loss
63              pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
64              correct += pred.eq(target.view_as(pred)).sum().item()
65
66      test_loss /= len(test_loader.dataset)
67
68      print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
69          test_loss, correct, len(test_loader.dataset),
70          100. * correct / len(test_loader.dataset)))
```

Run and Save model

```
127     for epoch in range(1, args.epochs + 1):
128         train(args, model, device, train_loader, optimizer, epoch)
129         test(model, device, test_loader)
130         scheduler.step()
131 
132     if args.save_model:
133         torch.save(model.state_dict(), "mnist_cnn.pt")
```