Deep Learning Software
PyTorch

Department of Computer Science, NCTU

TA Yu-Chuan Chuang

Some slides are from Stanford CS231n

Frameworks

TensorFlow

PyTorch

PaddlePaddle

Microsoft CNTK

Keras

torch

Chainer

Caffe2

DL4J

Caffe

mxnet

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

TensorFlow

PyTorch

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 !!

Install PyTorch

- http://pytorch.org/
- https://pytorch.org/get-started/previous-versions/
- https://www.anaconda.com/

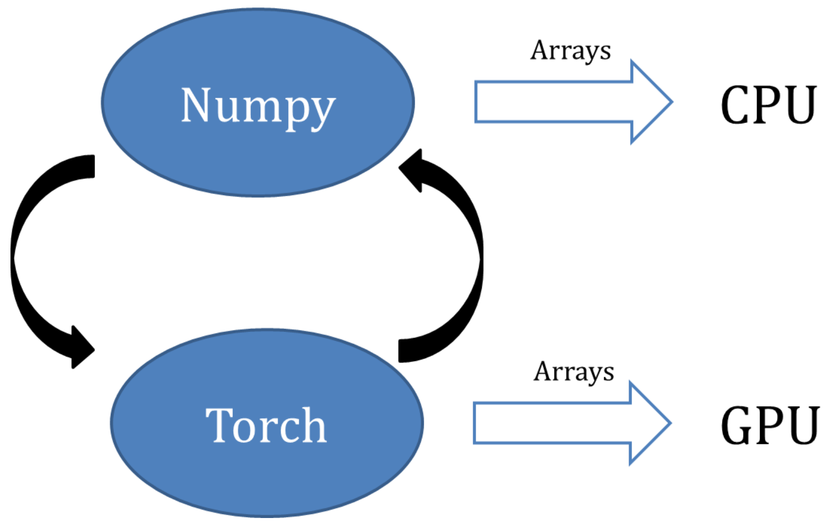
Get Started

Your OS	Linux	Mac	Windows		
Package	Conda	Pip	LibTorch	Source	
Language	Python 2.7	Python 3.5	Python 3.6	Python 3.7	C++
CUDA	8.0	9.0	10.0	None	

<u>TA's environment</u>

PyTorch Fundamental Concepts

- **Tensor** : Like a numpy array,
  but can run on GPU.



- **Autograd** : Package for building computational graphs out of Tensors and automatically computing gradients.

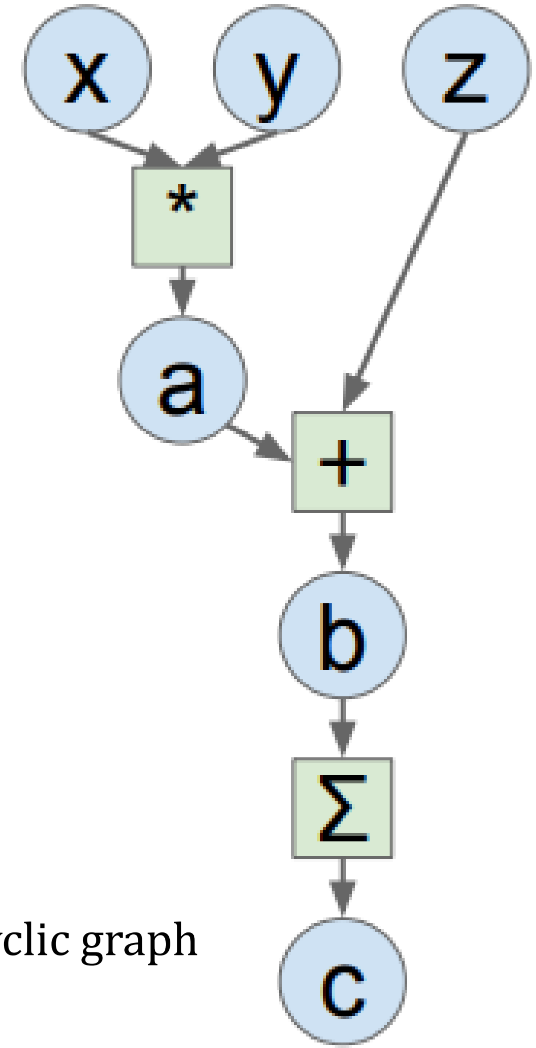
- **Module** : A neural network layer; may store state or learnable weights.

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)
```

$x \times y + z$



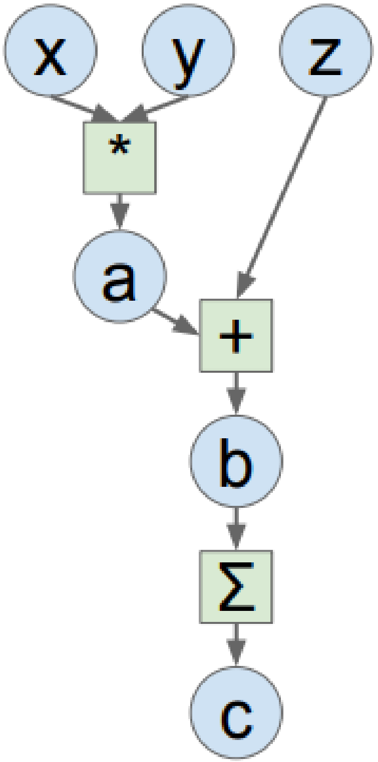
Neural network can be denoted as a directed acyclic graph

Computational Graphs

Numpy

```
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



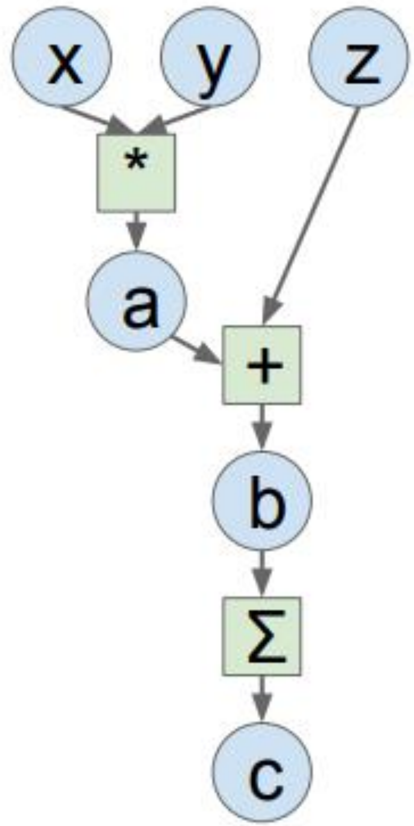
Problems:
- Can't run on GPU
- Have to compute our own gradients

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

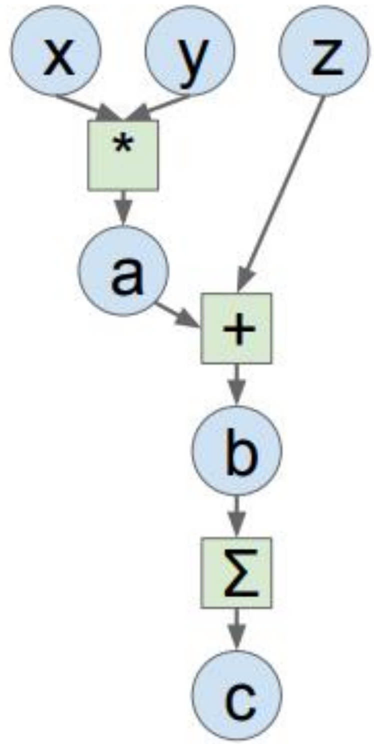
```
import torch

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)
```

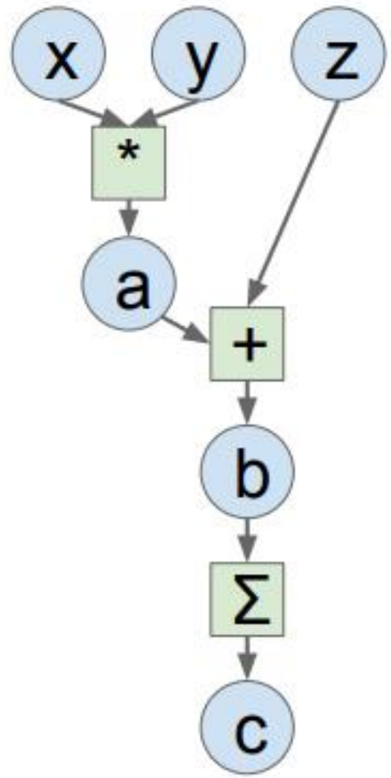
PyTorch handles gradients for us!

.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
```



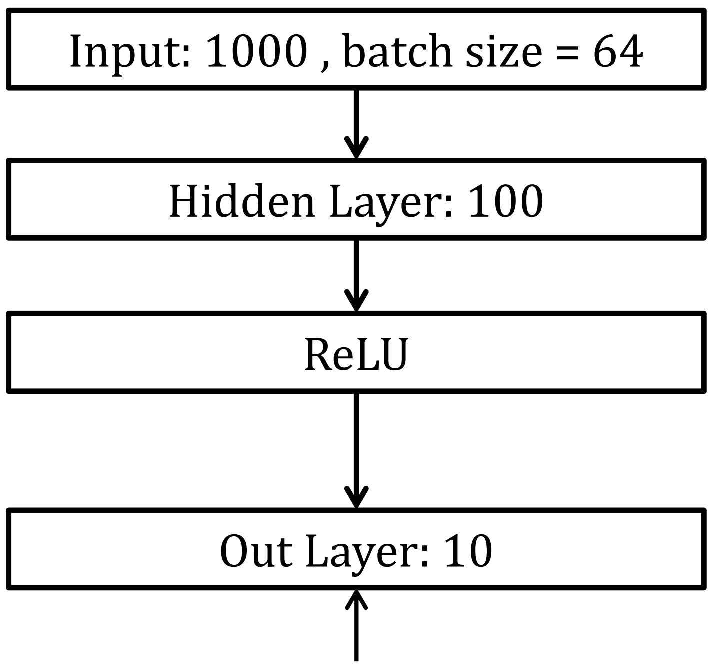
PyTorch

```
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



Loss : sum of square error

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

x: 64×1000

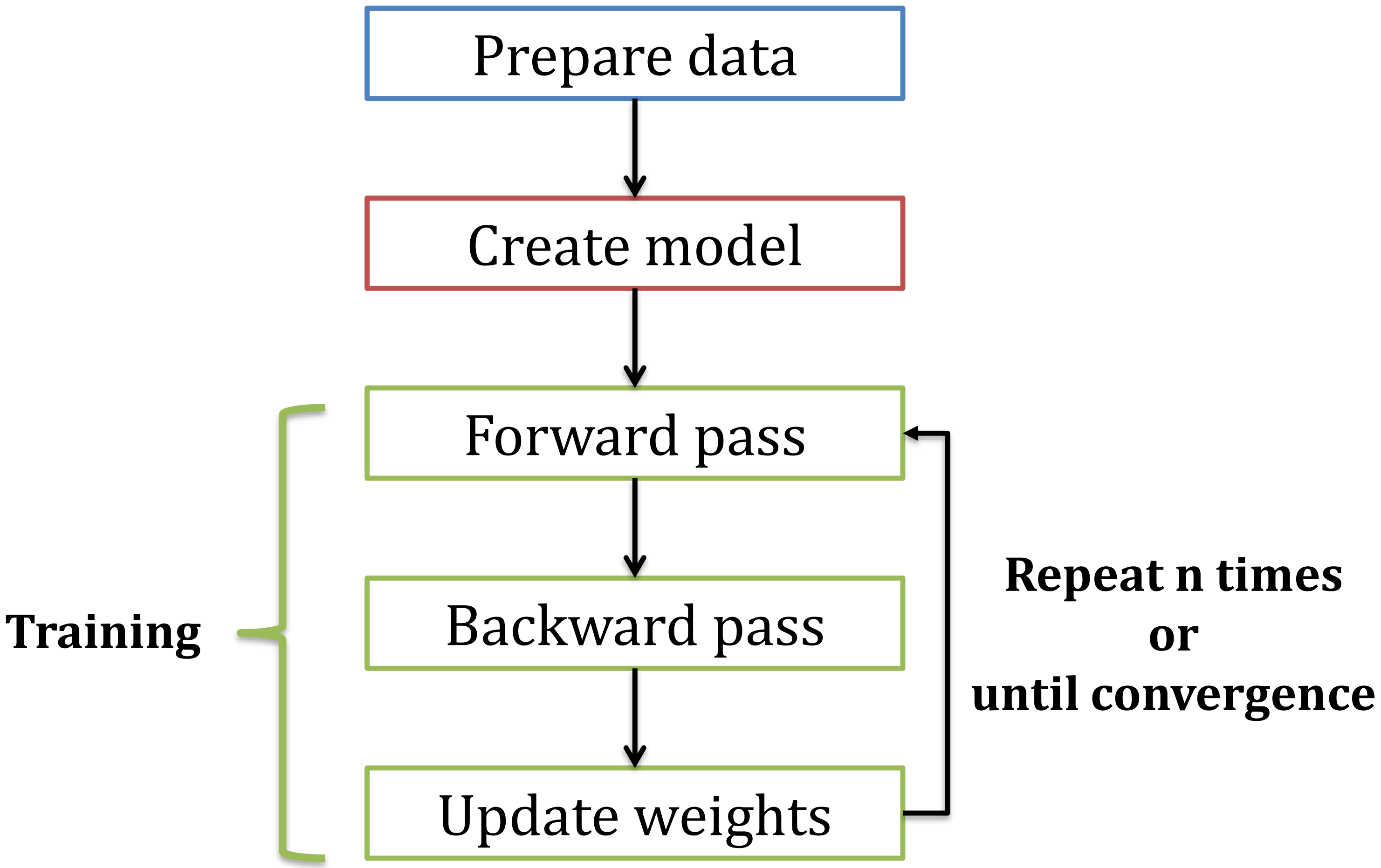
w1: 1000×100

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

w2: 100×10

y : 64×10

Flow Chart

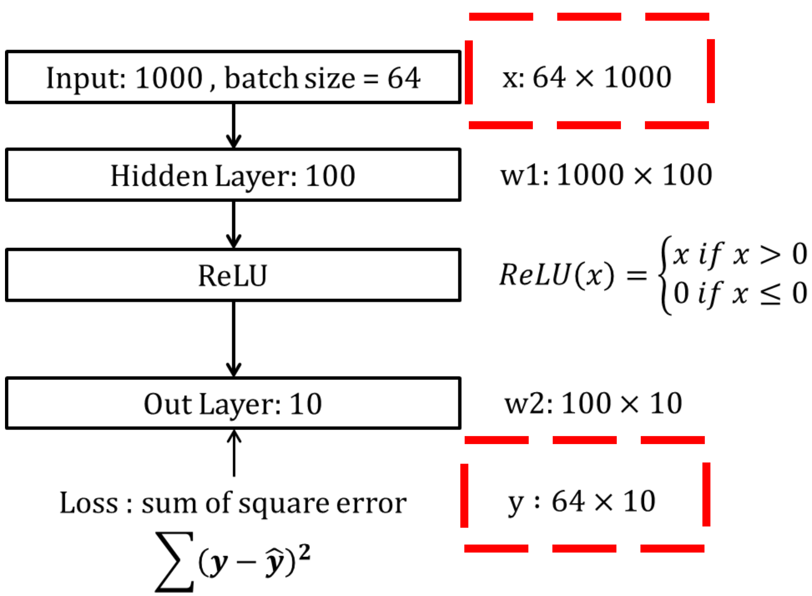

Prepare data
Create model
Forward pass
Backward pass
Update weights
Training
Repeat n times
or
until convergence

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:

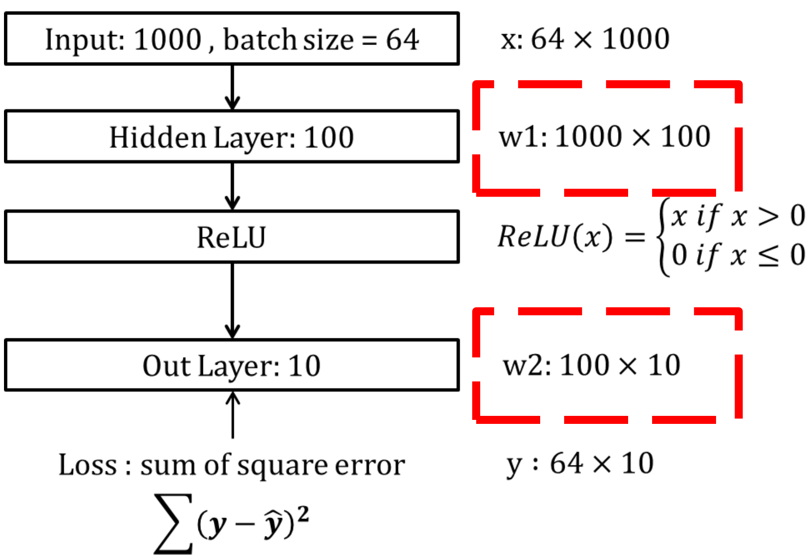
```
device = torch.device('cuda:0')
```




```
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 i in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0

    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_w1 = x.t().mm(grad_h)

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

Step2. Create Model
PyTorch Tensors

Create random tensors as layer weights

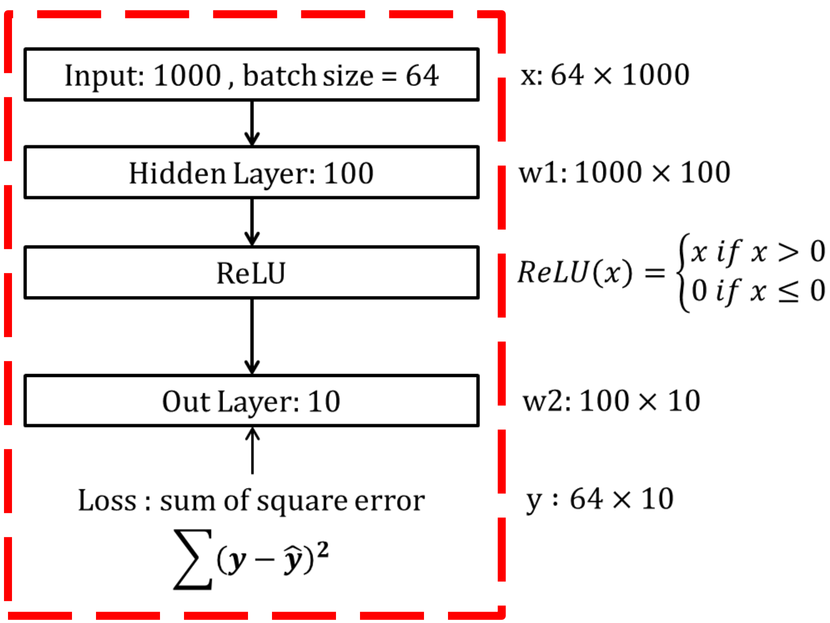



```
import torch
device = torch.device('cpu')
learning_rate = 1e-6

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 i in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0

    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_w1 = x.t().mm(grad_h)

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

Step3. Forward pass

PyTorch Tensors

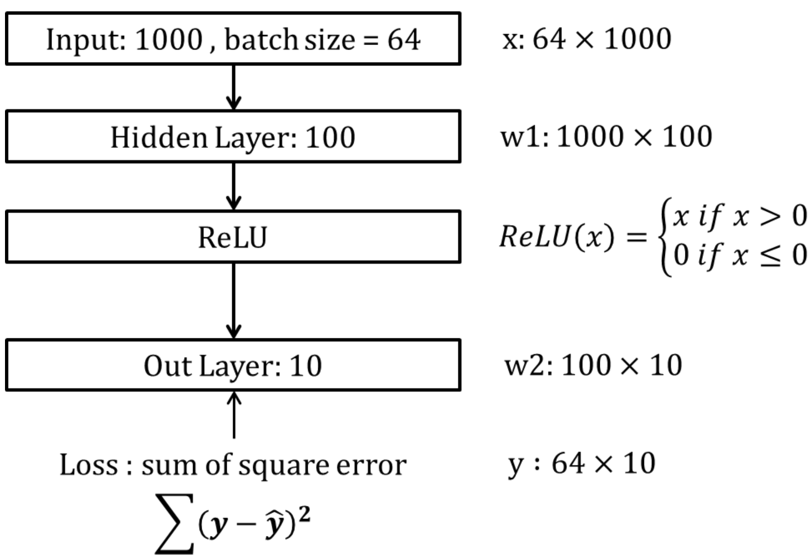
Compute predictions and loss




```
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 i in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0

    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_w1 = x.t().mm(grad_h)

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

Step4. Backward pass
PyTorch Tensors

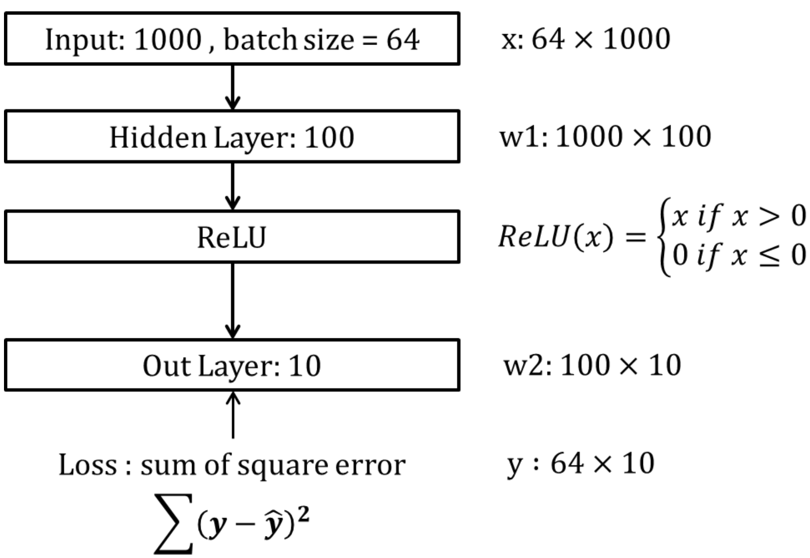
Manually compute gradients




```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 i in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0

    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_w1 = x.t().mm(grad_h)

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

Step5. Update Weights
PyTorch Tensors

Gradient descent step on weights




```
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 i in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0

    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_w1 = x.t().mm(grad_h)

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

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

Step1. Prepare Data
PyTorch.utils.data

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

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

Iterate over loader to form mini-batches

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

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

device = torch.device('cpu')
x = torch.randn(64, 1000)
y = torch.randn(64, 10)
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)
        self.relu = torch.nn.ReLU()

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


model = TwoLayerNet(D_in=1000, H=100, D_out=10)
optimizer = torch.optim.SGD(model.parameters(),
                            lr=1e-1)
for epochs in range(500):
    for x_batch, y_batch in loader:
        optimizer.zero_grad()
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)

        loss.backward()
        optimizer.step()
        print(loss.item())
```

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')
x = torch.randn(64, 1000)
y = torch.randn(64, 10)
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)
        self.relu = torch.nn.ReLU()

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

model = TwoLayerNet(D_in=1000, H=100, D_out=10)
optimizer = torch.optim.SGD(model.parameters(),
                            lr=1e-1)
for epochs in range(500):
    for x_batch, y_batch in loader:
        optimizer.zero_grad()
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)

        loss.backward()
        optimizer.step()
        print(loss.item())
```

Step2. Create Model
PyTorch.nn

Initializer sets up two children (Module can contain Modules)

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

device = torch.device('cpu')
x = torch.randn(64, 1000)
y = torch.randn(64, 10)
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)
        self.relu = torch.nn.ReLU()

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


model = TwoLayerNet(D_in=1000, H=100, D_out=10)
optimizer = torch.optim.SGD(model.parameters(),
                            lr=1e-1)
for epochs in range(500):
    for x_batch, y_batch in loader:
        optimizer.zero_grad()
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)

        loss.backward()
        optimizer.step()
        print(loss.item())
```

Step2. Create Model
PyTorch.nn

Define forward pass using child modules

No need to define backward – autograd will handle it.

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

device = torch.device('cpu')
x = torch.randn(64, 1000)
y = torch.randn(64, 10)
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)
        self.relu = torch.nn.ReLU()

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


model = TwoLayerNet(D_in=1000, H=100, D_out=10)
optimizer = torch.optim.SGD(model.parameters(),
                            lr=1e-1)
for epochs in range(500):
    for x_batch, y_batch in loader:
        optimizer.zero_grad()
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)

        loss.backward()
        optimizer.step()
        print(loss.item())
```

Step3. Forward pass
PyTorch.nn

Define forward pass using child modules

Feed data to model, and compute loss

nn.functional has a lot of useful helpers like loss functions

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

device = torch.device('cpu')
x = torch.randn(64, 1000)
y = torch.randn(64, 10)
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)
        self.relu = torch.nn.ReLU()

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


model = TwoLayerNet(D_in=1000, H=100, D_out=10)
optimizer = torch.optim.SGD(model.parameters(),
                            lr=1e-1)
for epochs in range(500):
    for x_batch, y_batch in loader:
        optimizer.zero_grad()
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)

        loss.backward()
        optimizer.step()
        print(loss.item())
```

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')
x = torch.randn(64, 1000)
y = torch.randn(64, 10)
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)
        self.relu = torch.nn.ReLU()

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


model = TwoLayerNet(D_in=1000, H=100, D_out=10)
optimizer = torch.optim.SGD(model.parameters(),
                            lr=1e-1)
for epochs in range(500):
    for x_batch, y_batch in loader:
        optimizer.zero_grad()
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)

        loss.backward()
        optimizer.step()
        print(loss.item())
```

Step5. Update Weights
PyTorch.optim

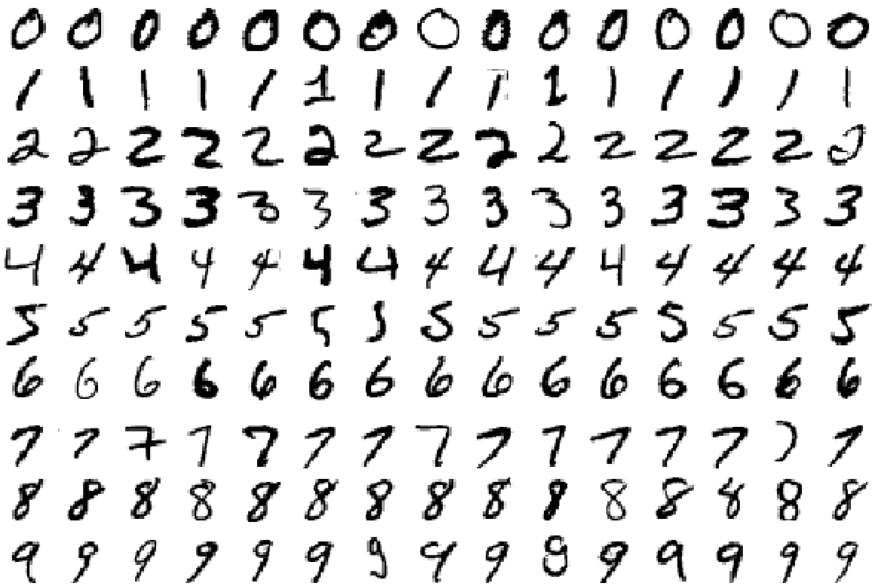
Use an **optimizer** for different update rules.

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')
x = torch.randn(64, 1000)
y = torch.randn(64, 10)
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)
        self.relu = torch.nn.ReLU()

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


model1 = TwoLayerNet(D_in=1000, H=100, D_out=10)
optimizer = torch.optim.SGD(model1.parameters(),
                            lr=1e-1)
for epochs in range(500):
    for x_batch, y_batch in loader:
        optimizer.zero_grad()
        y_pred = model1(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred,
                                            y_batch)

        loss.backward()
        optimizer.step()
        print(loss.item())
```

Real Application

- MNIST example for PyTorch



- git clone https://github.com/JiaRenChang/DLcourse_NCTU.git

CNN_MNIST_pytorch.py

Build and train a CNN classifier

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

Set hypermeters

```
# Training settings
parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
parser.add_argument('--batch-size', type=int, default=128, metavar='N',
                    help='input batch size for training (default: 128)')
parser.add_argument('--epochs', type=int, default=20, metavar='N',
                    help='number of epochs to train (default: 164)')
parser.add_argument('--lr', type=float, default=0.1, metavar='LR',
                    help='learning rate (default: 0.1)')
parser.add_argument('--momentum', type=float, default=0.9, metavar='M',
                    help='SGD momentum (default: 0.9)')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='disables CUDA training')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')

args = parser.parse_args()
```

Data Loader

- Pytorch offers data loaders for popular dataset

The following datasets are available:

Datasets

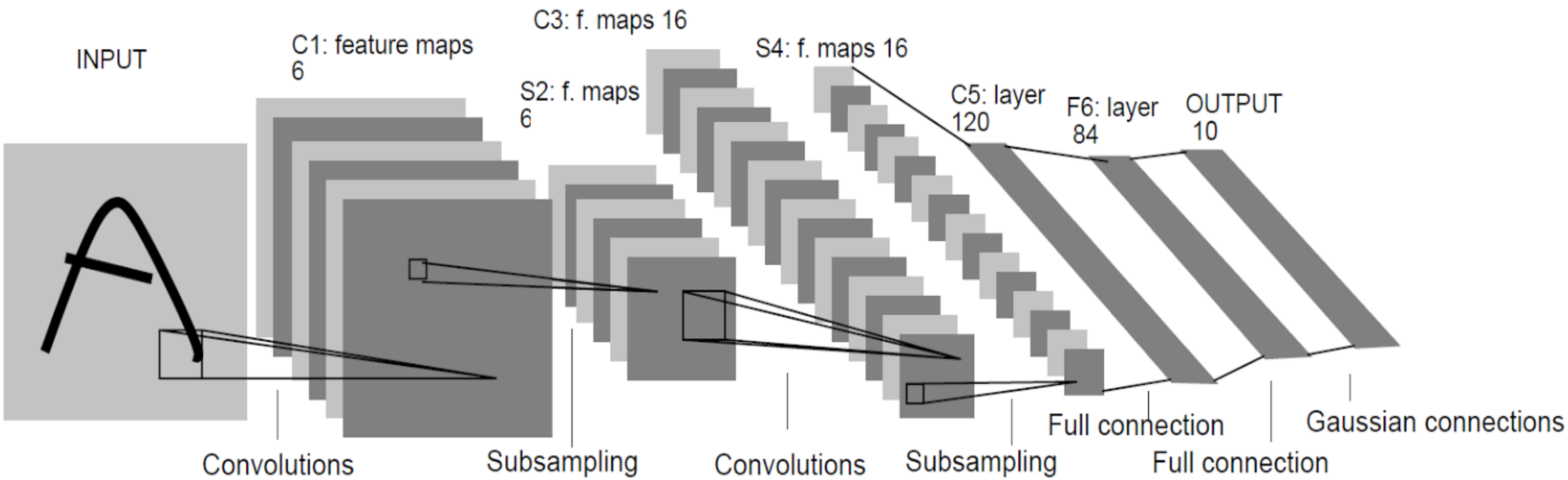
- MNIST
- COCO
  - Captions
  - Detection
- LSUN
- ImageFolder
- Imagenet-12
- CIFAR
- STL10
- SVHN
- PhotoTour

Data Loader

```python
import torch
from torchvision import datasets, transforms

# Dataloader
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])),
    batch_size=args.batch_size, shuffle=True,num_workers = 2)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False, transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])),
    batch_size=args.batch_size, shuffle=True,num_workers = 2)
```

Define Network

- LeNet



Define Network

```python
#Define Network, we implement LeNet here
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=(5,5),stride=1, padding=0)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=(5,5),stride=1, padding=0)
        self.fc1 = nn.Linear(16*4*4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = F.max_pool2d(out, 2)
        out = F.relu(self.conv2(out))
        out = F.max_pool2d(out, 2)
        out = out.view(out.size(0), -1) #flatten
        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out

model = Net()
if args.cuda:
        device = torch.device('cuda')
        model.to(device)
```

Define Optimizer/Loss function

- Cross Entropy Loss
- Stochastic Gradient Descent

```
#define optimizer/loss function
Loss = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=args.lr, momentum=args.momentum)
```

Learning rate scheduling

- 20 epochs
- LR decay at 10 and 15 epoch

```
#learning rate scheduling
def adjust_learning_rate(optimizer, epoch):

    if epoch < 10:
        lr = 0.01
    elif epoch < 15:
        lr = 0.001
    else:
        lr = 0.0001

    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
```

Training

```
#training function
def train(epoch):
    model.train()  Set model to training mode
    adjust_learning_rate(optimizer, epoch)

    for batch_idx, (data, target) in enumerate(train_loader):
        if args.cuda:
            data, target = data.to(device), target.to(device)

        optimizer.zero_grad()      Clean gradient
        output = model(data)
        loss = Loss(output, target)
        loss.backward()    Backward gradient
        optimizer.step()   Update weight
        print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.data[0]))
```

Testing

```python
#Testing function
def test(epoch):
    model.eval()
    test_loss = 0
    correct = 0
    for batch_idx, (data, target) in enumerate(test_loader):
        if args.cuda:
            data, target = data.to(device), target.to(device)
        with torch.no_grad():
                output = model(data)
        test_loss += Loss(output, target).data[0]
        pred = output.data.max(1)[1] # get the index of the max log-probability
        correct += pred.eq(target.data).cpu().sum()

    test_loss = test_loss
    test_loss /= len(test_loader) # loss function already averages over batch size
    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

```
#run and save model
for epoch in range(1, args.epochs + 1):
    train(epoch)
    test(epoch)
    savefilename = 'LeNet_'+str(epoch)+'.tar'
    torch.save({
            'epoch': epoch,
            'state_dict': model.state_dict(),
        }, savefilename)
```

You can achieve ~99.1% test accuracy.

Exercise

- Deeper: add more convolution layer
  - insert two 3x3 conv layer between conv1 and conv2 (stride=1,pad=1)
  - Hint: define new conv layer, and forward
  - Notice the *spatial* dimension
- Wider: add more neuron
  - Make your net 2x wider
  - Notice the *in/out* dimension
- Other Optimizer
  - Try Adam/RMSprop
- More epochs, New learning rate schedule, ….