

# PyTorch Warm Up

Deep Learning and Practice 2018 spring  
Department of Computer Science, NCTU

Some slides are from Stanford CS231n

# Frameworks

- **C++**
  - Caffe
- **Python**
  - Caffe2, TensorFlow, PyTorch, Theano, Keras, MXNet
- **Lua**
  - Torch7
- **Matlab**
  - MatConvNet

↑  
It died.

**Google:**  
TensorFlow



*“One framework  
to rule them all”*

**Facebook:**  
PyTorch +Caffe2



Research

Production

# Future Homework

- Do your homework using **PyTorch**
- Other frameworks are available:
  - Keras (backend= tensorflow)
  - Tensorflow

# Install PyTorch

- <http://pytorch.org/>

## Get Started.

Select your preferences, then run the PyTorch install command.

Please ensure that you are on the latest pip and numpy packages.

Anaconda is our recommended package manager

|                 |  |                                      |                                      |                            |
|-----------------|--|--------------------------------------|--------------------------------------|----------------------------|
| OS              | <input checked="" type="radio"/> Linux | <input type="radio"/> OSX            |                                      |                            |
| Package Manager | <input type="radio"/> conda            | <input checked="" type="radio"/> pip | <input type="radio"/> Source         |                            |
| Python          | <input checked="" type="radio"/> 2.7   | <input type="radio"/> 3.5            | <input type="radio"/> 3.6            |                            |
| CUDA            | <input type="radio"/> 8                | <input type="radio"/> 9.0            | <input checked="" type="radio"/> 9.1 | <input type="radio"/> None |

### Run this command:

```
pip install http://download.pytorch.org/whl/cu91/torch-0.3.1-cp27-cp27mu-linux_x86_64.whl
pip install torchvision
```

# if the above command does not work, then you have python 2.7 UCS2, use this command  
pip install http://download.pytorch.org/whl/cu91/torch-0.3.1-cp27-cp27m-linux\_x86\_64.whl

# Computational Graphs

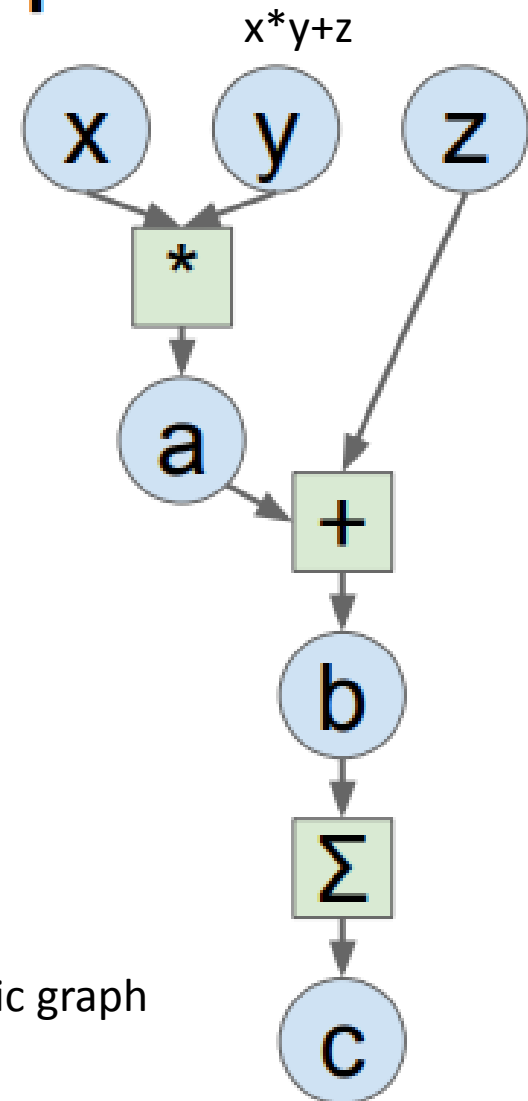
## Numpy

```
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)
```



Neural network can be denoted as a directed acyclic graph

# Computational Graphs

## Numpy

```
import numpy as np
np.random.seed(0)
```

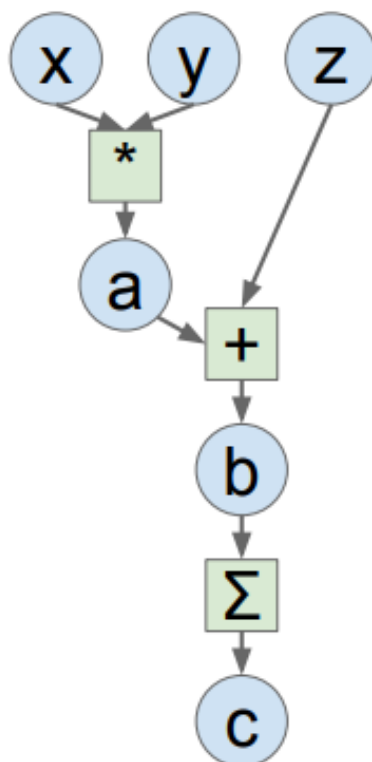
```
N, D = 3, 4
```

```
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
```

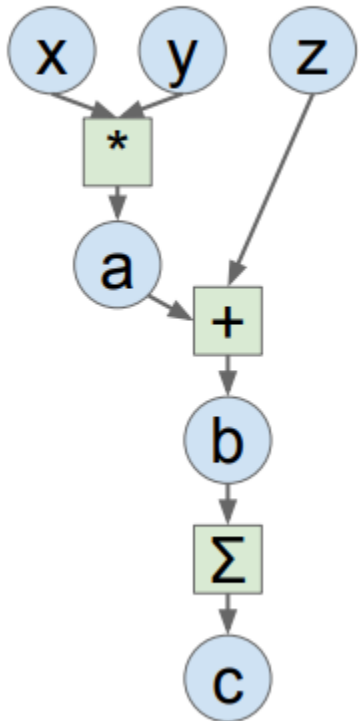
compute gradients



## Problems:

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

# Computational Graphs



Define Variable to start  
building graph

Forward Pass

Compute gradient

## PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4
```

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

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

```
c.backward()
```

```
print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```



# PyTorch: Three Levels of Abstraction

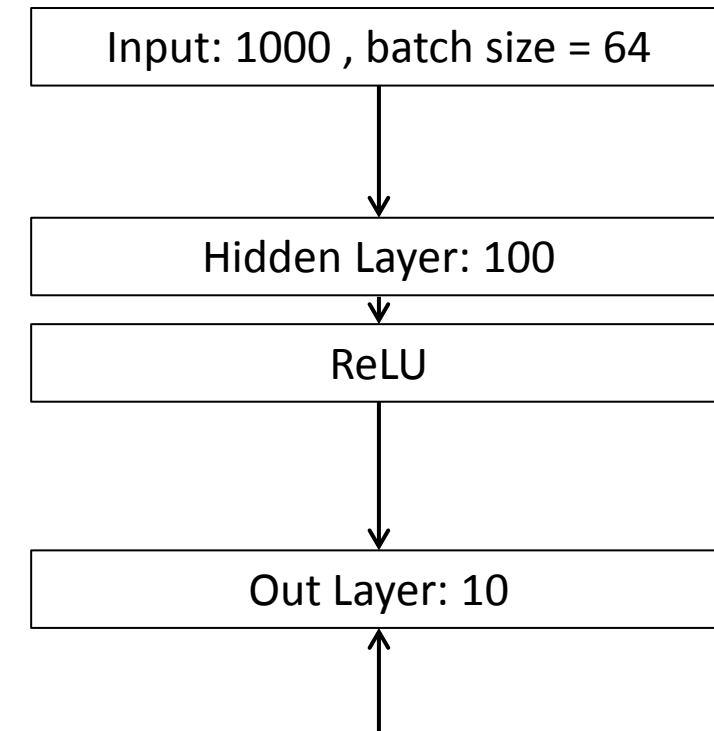
**Tensor:** Imperative ndarray,  
but runs on GPU

**Variable:** Node in a  
computational graph; stores  
data and gradient

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

# Example

- A 2-layer ReLU network



Loss: sum of square error  $\sum (y - \hat{y})^2$

$x: 64 \times 1000$

$w1: 1000 \times 100$

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

$w2: 100 \times 10$

$y: 64 \times 10$

# PyTorch: Autograd

A PyTorch **Variable** is a node in a computational graph

`x.data` is a Tensor

`x.grad` is a Variable of gradients  
(same shape as `x.data`)

`x.grad.data` is a Tensor of gradients

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()  # compute gradient

    w1.data -= learning_rate * w1.grad.data  # apply gradient
    w2.data -= learning_rate * w2.grad.data
```

# PyTorch: Autograd

We will not want gradients  
(of loss) with respect to data

Do want gradients with  
respect to weights

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

# PyTorch: Autograd

Forward pass looks exactly the same as the Tensor version, but everything is a variable now

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

# PyTorch: nn

Higher-level wrapper for  
working with neural nets

Similar to Keras and friends ...  
but only one, and it's good =)

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

# PyTorch: nn

Define our model as a  
sequence of layers

nn also defines common  
loss functions

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

# PyTorch: nn

Forward pass: feed data  
to model, and prediction  
to loss function

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```



# PyTorch: nn

Backward pass:  
compute all gradients

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

# PyTorch: optim

Use an **optimizer** for different update rules

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                              lr=learning_rate)

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
```

# PyTorch: optim

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                               lr=learning_rate)

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
```

Update all parameters  
after computing gradients



# PyTorch: nn

## Define new Modules

A PyTorch **Module** is a neural net layer; it inputs and outputs Variables

Modules can contain weights (as Variables) or other Modules

You can define your own Modules using autograd!

```
import torch
from torch.autograd import Variable

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = TwoLayerNet(D_in, H, D_out)


criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

# PyTorch: nn

## Define new Modules

Define our whole model  
as a single Module



```
import torch
from torch.autograd import Variable

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(1, D_in))
y = Variable(torch.randn(1, D_out), requires_grad=False)

model = TwoLayerNet(D_in, H, D_out)


criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

# PyTorch: nn

## Define new Modules

Initializer sets up two children (Modules can contain modules)



```
import torch
from torch.autograd import Variable

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(1, D_in))
y = Variable(torch.randn(1, D_out), requires_grad=False)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

# PyTorch: nn

## Define new Modules

Define forward pass using  
child modules and  
autograd ops on Variables

No need to define  
backward - autograd will  
handle it

```
import torch
from torch.autograd import Variable

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)


    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```



# PyTorch: nn

## Define new Modules

Construct and train an instance of our model



```
import torch
from torch.autograd import Variable

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```



# Real Application

- MNIST example for PyTorch



- git clone [https://github.com/JiaRenChang/DLcourse\\_NCTU.git](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

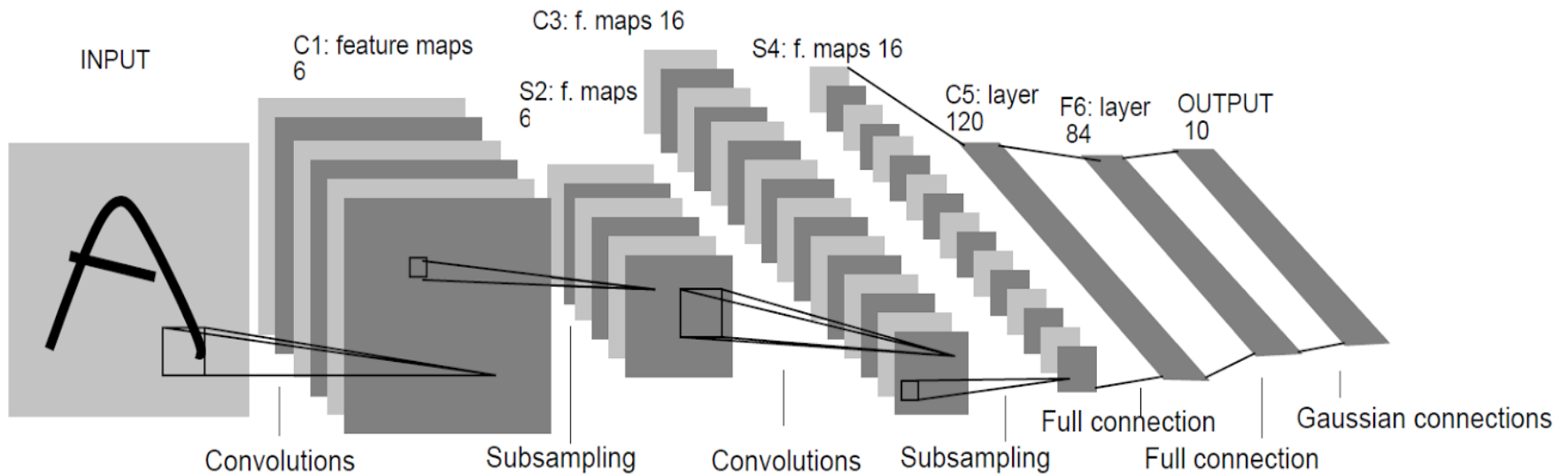
```
import torch
from torchvision import datasets, transforms

# 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

```
#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:
    model.cuda()
```

# 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.cuda(), target.cuda()
            data, target = Variable(data), Variable(target)
            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

```
#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.cuda(), target.cuda()
        data, target = Variable(data, volatile=True), Variable(target)
        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, ....