PyTorch Tutorial
-NTU Machine Learning Course-

Lyman Lin 林裕訓
Nov. 03, 2017
lymanblue[at]gmail.com

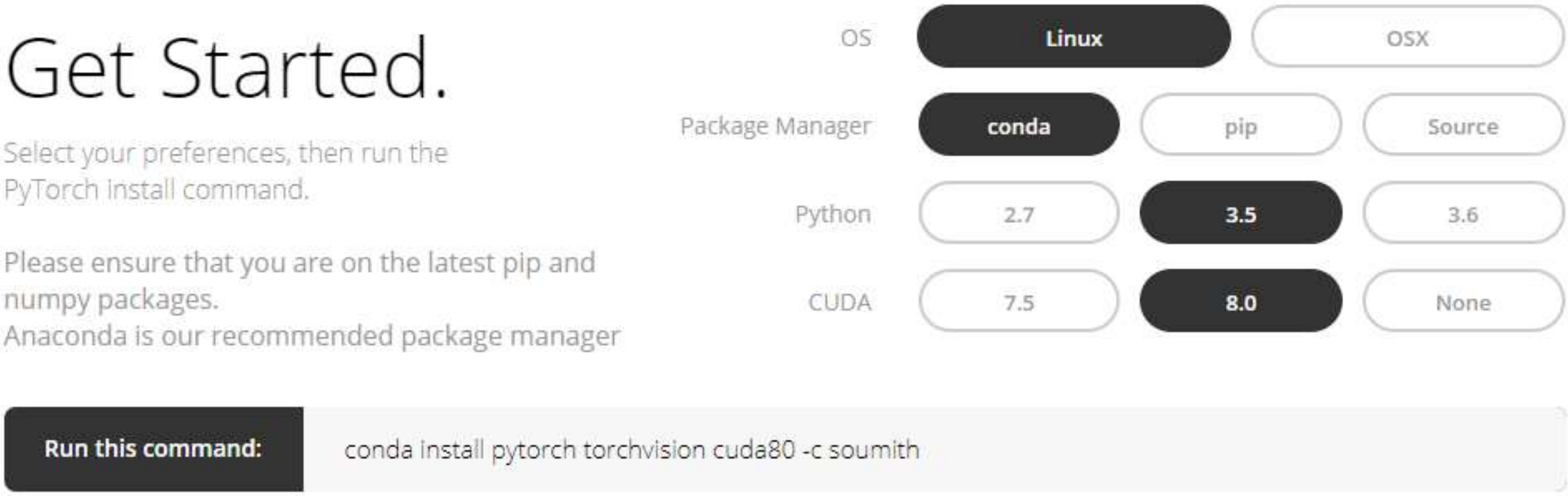
What is PyTorch?

- Developed by Facebook
  - Python first
  - Dynamic Neural Network
  - This tutorial is for PyTorch 0.2.0

- Endorsed by Director of AI at Tesla

Installation

- PyTorch Web: http://pytorch.org/



Packages of PyTorch

This Tutorial

Package	Description
torch	a Tensor library like Numpy, with strong GPU support
torch.autograd	a tape based automatic differentiation library that supports all differentiable Tensor operations in torch
torch.nn	a neural networks library deeply integrated with autograd designed for maximum flexibility
torch.optim	an optimization package to be used with torch.nn with standard optimization methods such as SGD, RMSProp, LBFGS, Adam etc.
torch.multiprocessing	python multiprocessing, but with magical memory sharing of torch Tensors across processes. Useful for data loading and hogwild training.
torch.utils	DataLoader, Trainer and other utility functions for convenience
torch.legacy(.nn/.optim)	legacy code that has been ported over from torch for backward compatibility reasons

Outline

- Neural Network in Brief
- Concepts of PyTorch
- Multi-GPU Processing
- RNN
- Transfer Learning
- Comparison with TensorFlow

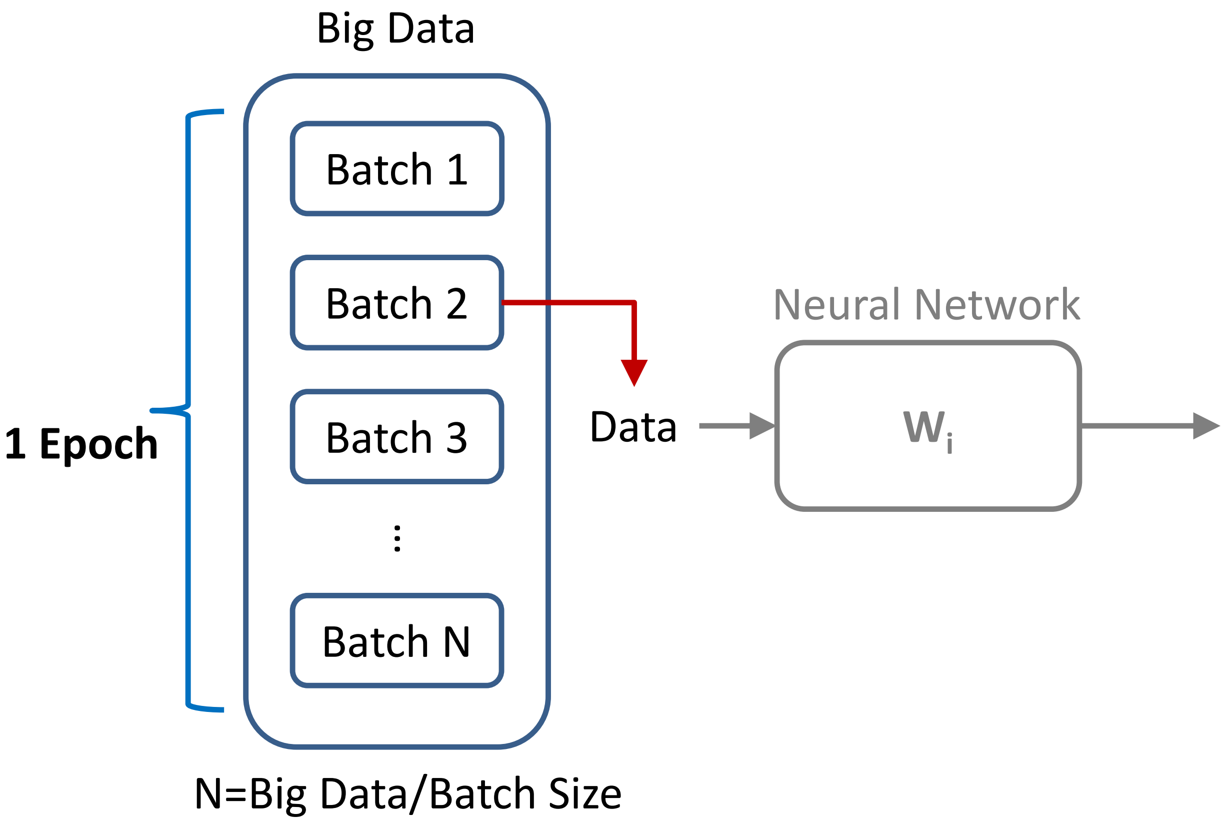
Neural Network in Brief

- Supervised Learning
  - Learning a function f, that f(x)=y

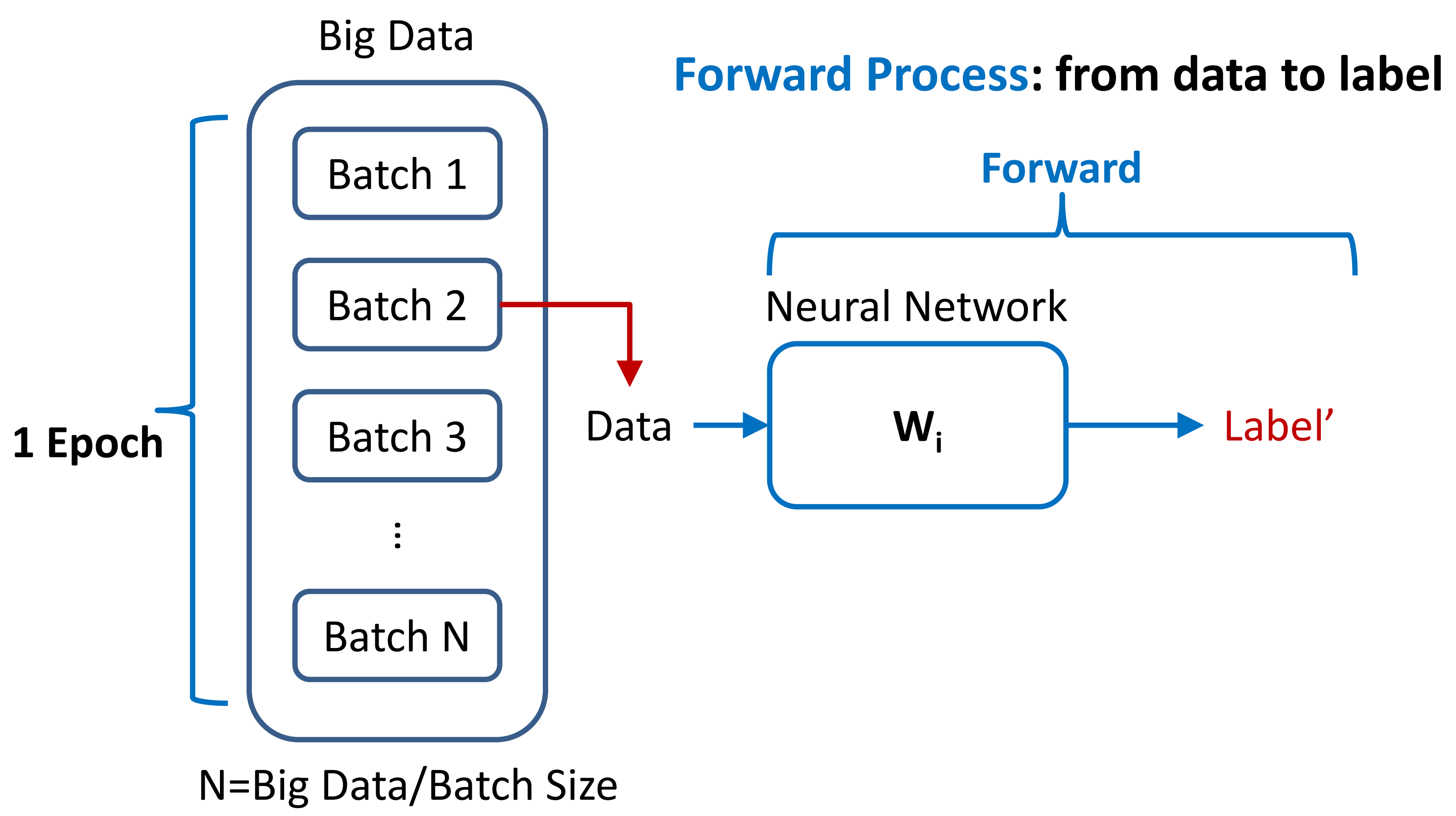
Trying to learn f(.), that f(x)=y

Data	Label
X1	Y1
X2	Y2
…	…

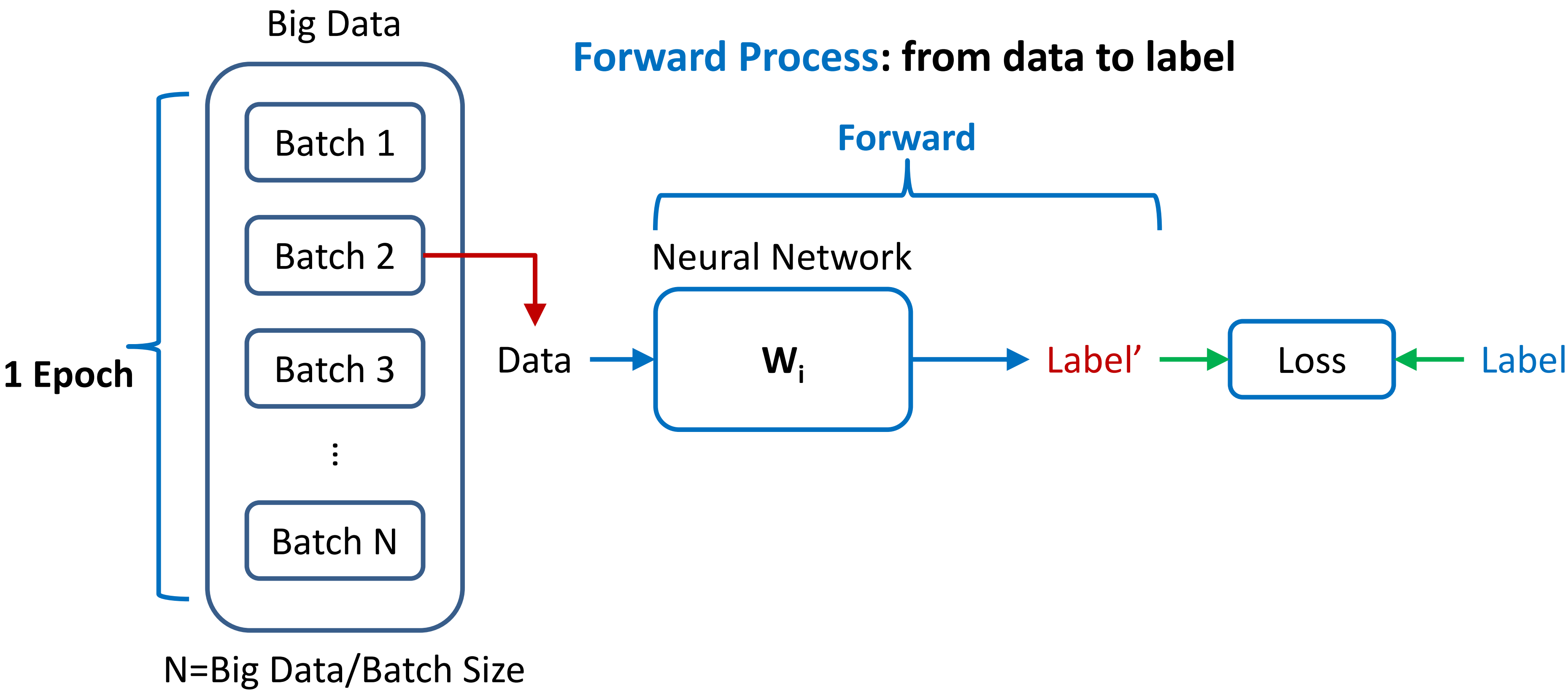
Neural Network in Brief



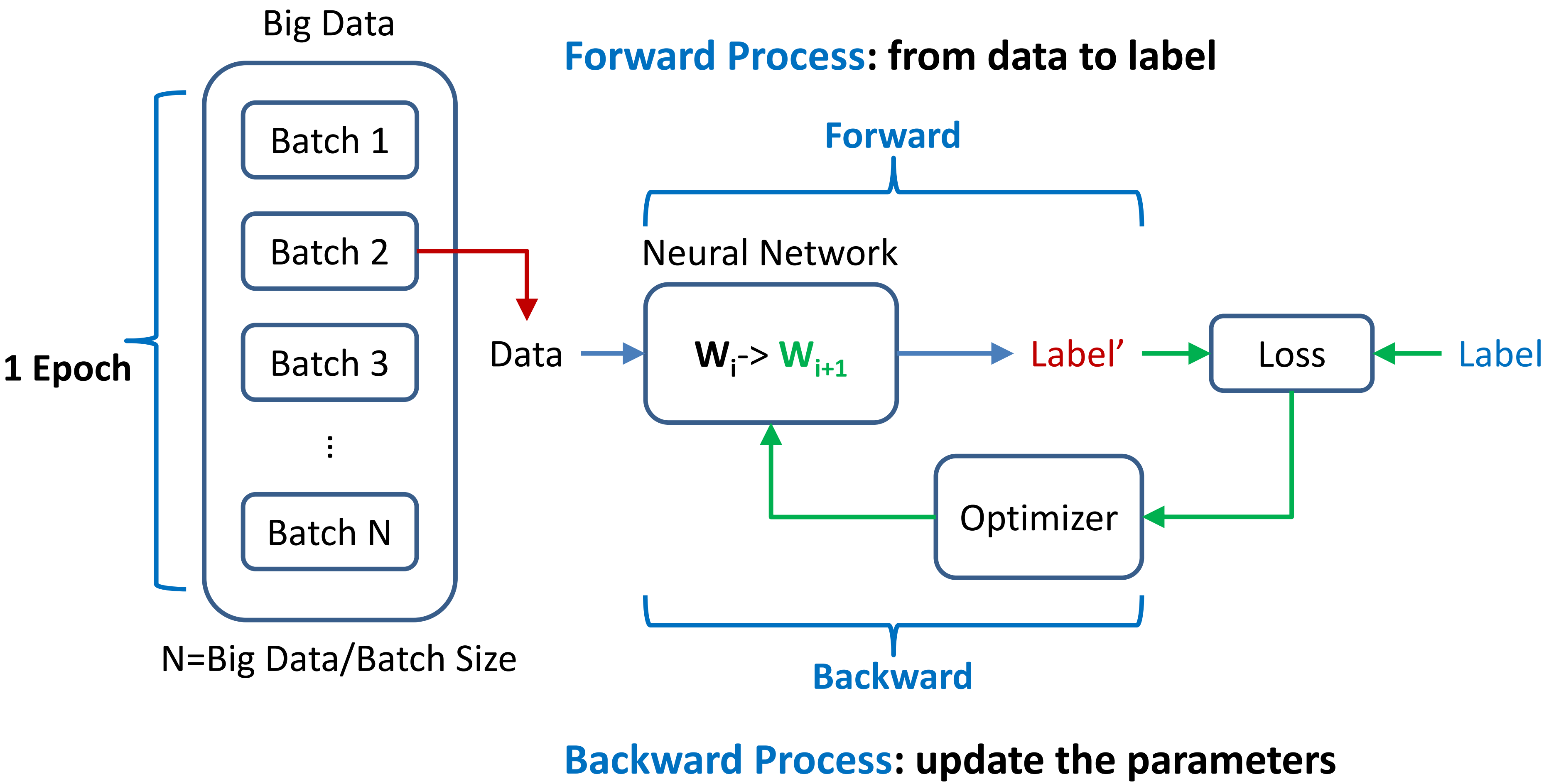
Neural Network in Brief



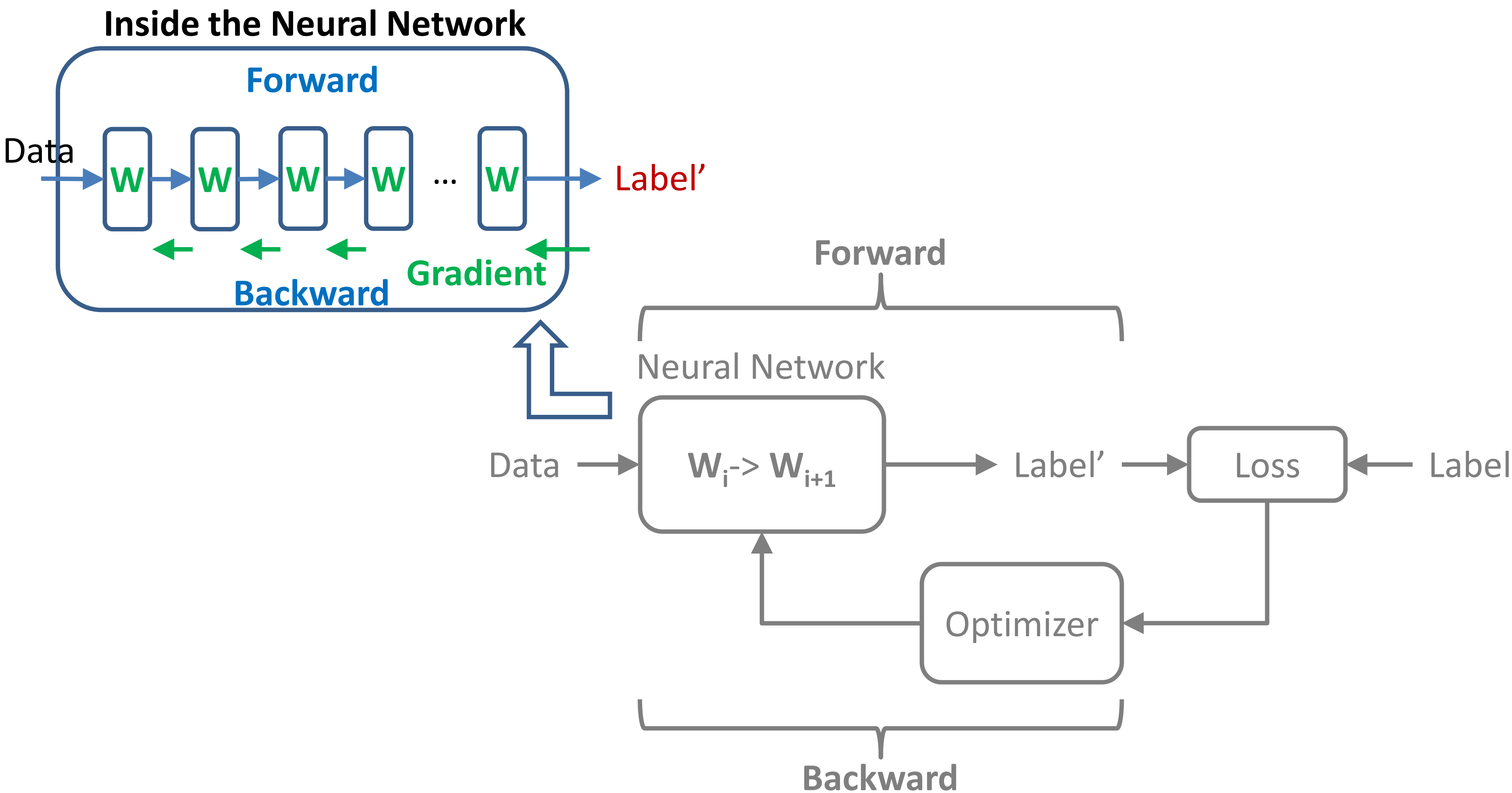
Neural Network in Brief



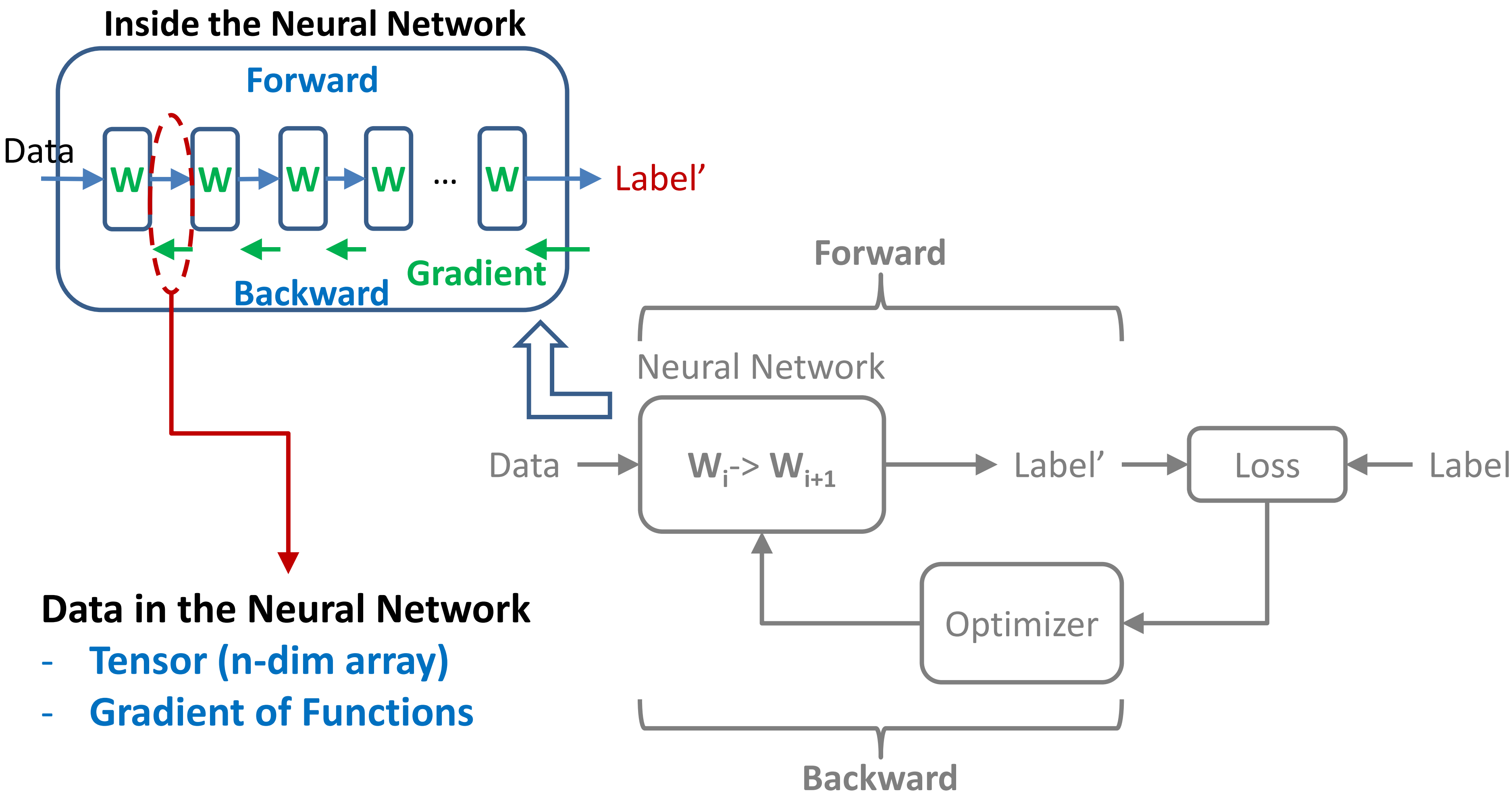
Neural Network in Brief



Neural Network in Brief



Neural Network in Brief



Data in the Neural Network

- **Tensor (n-dim array)**
- **Gradient of Functions**

Concepts of PyTorch

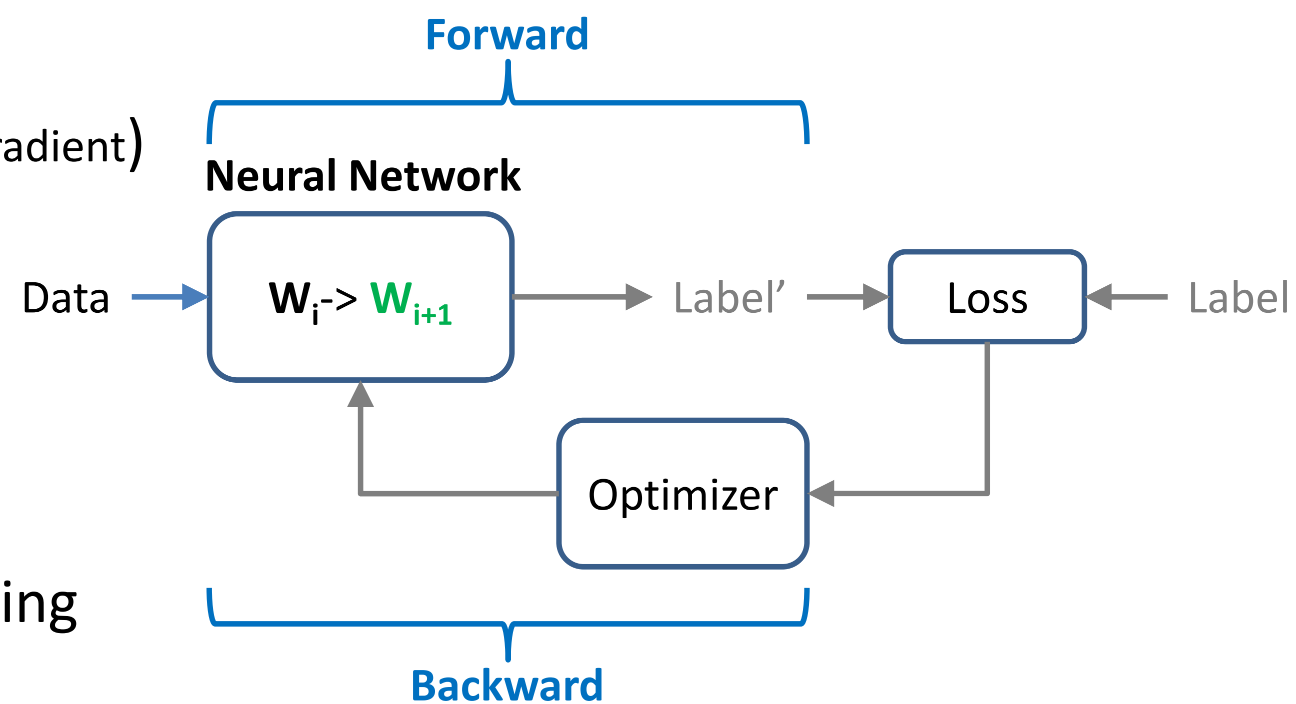
- Modules of PyTorch

Data:

- Tensor
- Variable (for Gradient)

Function:

- NN Modules
- Optimizer
- Loss Function
- Multi-Processing



Concepts of PyTorch

- Modules of PyTorch

Data:
- **Tensor**
- Variable (for Gradient)

Function:
- NN Modules
- Optimizer
- Loss Function
- Multi-Processing

- Similar to Numpy

```
x = torch.Tensor(5, 3)
print(x)
```

Out:

```
1.00000e-36 *
  0.0228  0.0000  1.3490
  0.0000  0.0958  0.0000
  0.0958  0.0000  0.0958
  0.0000  0.0958  0.0000
  0.0958  0.0000  0.0958
[torch.FloatTensor of size 5x3]
```

```
x = torch.rand(5, 3)
print(x)
```

Out:

```
0.2285  0.2843  0.1978
 0.0092  0.8238  0.2703
 0.1266  0.9613  0.2472
 0.0918  0.2827  0.9803
 0.9237  0.1946  0.0104
[torch.FloatTensor of size 5x3]
```

Concepts of PyTorch

- Modules of PyTorch

  Data:
  - **Tensor**
  - Variable (for Gradient)

  Function:
  - NN Modules
  - Optimizer
  - Loss Function
  - Multi-Processing

- Operations
  – z=x+y
  – torch.add(x,y, out=z)
  – y.add_(x) # in-place

Concepts of PyTorch

- Modules of PyTorch

  Data:
  - **Tensor**
  - Variable (for Gradient)

  Function:
  - NN Modules
  - Optimizer
  - Loss Function
  - Multi-Processing

- Numpy Bridge

- To Numpy
  - a = torch.ones(5)
  - b = a.numpy()

- To Tensor
  - a = numpy.ones(5)
  - b = torch.from_numpy(a)

Concepts of PyTorch

- Modules of PyTorch

  Data:
  - **Tensor**
  - Variable (for Gradient)

  Function:
  - NN Modules
  - Optimizer
  - Loss Function
  - Multi-Processing

- CUDA Tensors

- Move to GPU
  - x = x.cuda()
  - y = y.cuda()
  - x+y

Concepts of PyTorch

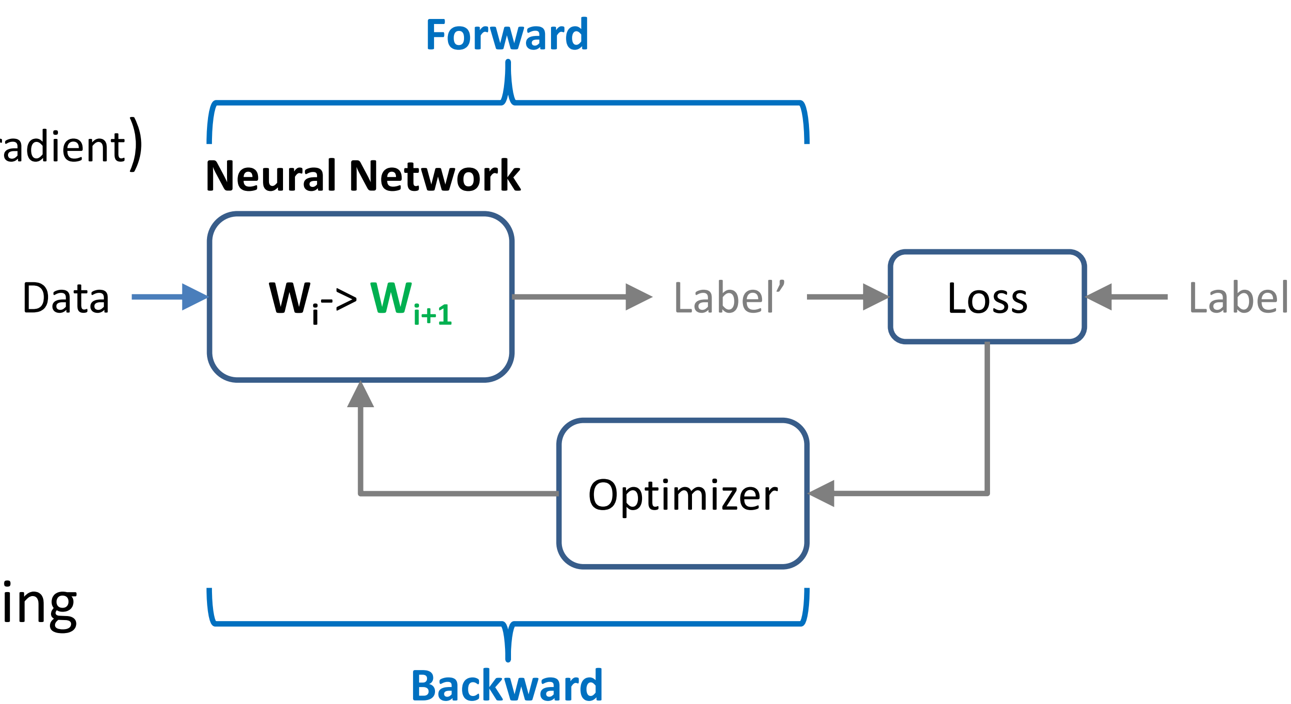
- Modules of PyTorch

Data:

- Tensor
- Variable (for Gradient)

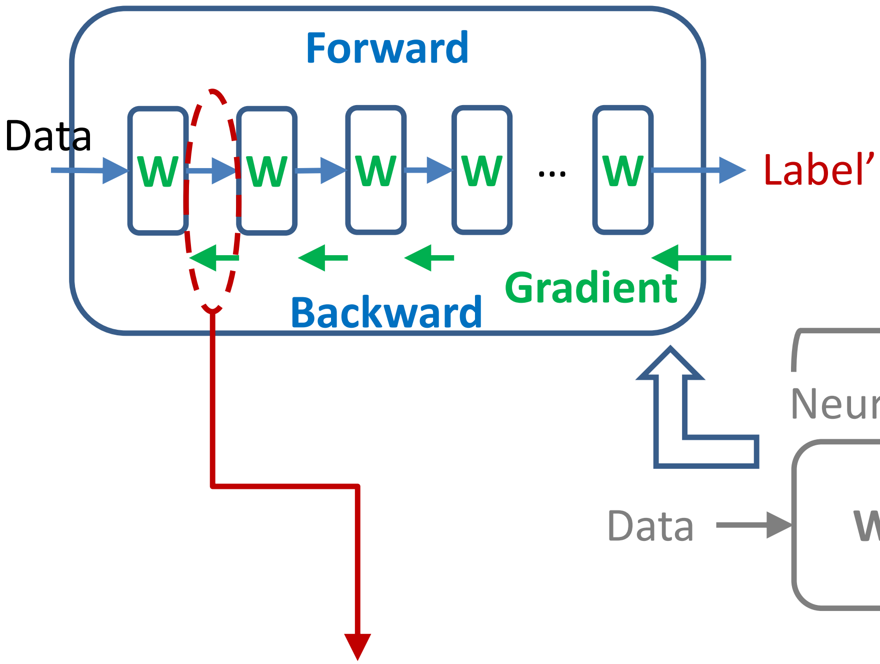
Function:

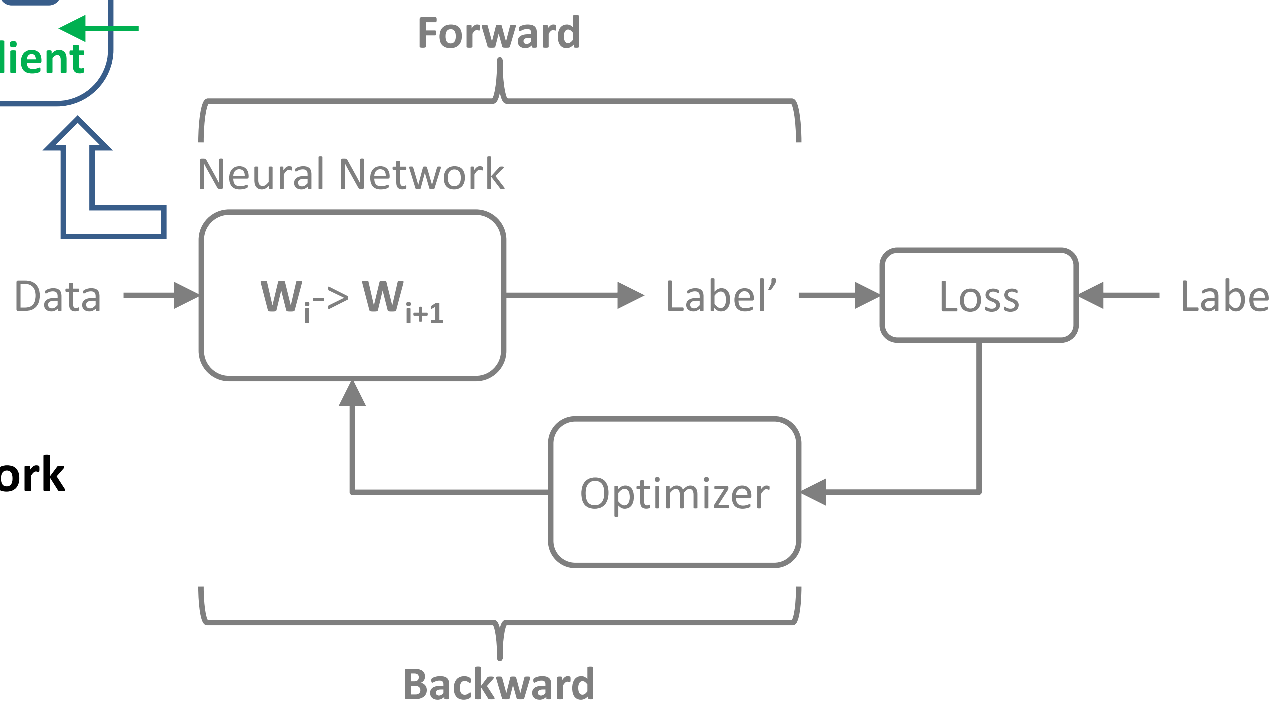
- NN Modules
- Optimizer
- Loss Function
- Multi-Processing



Neural Network in Brief

Inside the Neural Network





Data in the Neural Network

- **Tensor (n-dim array)**
- **Gradient of Functions**

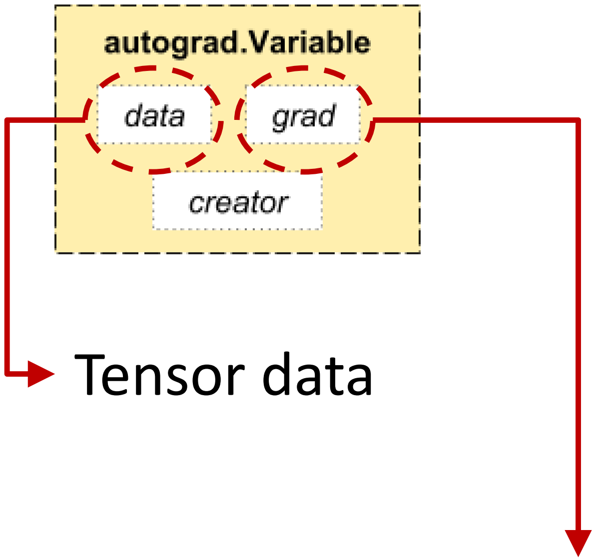
Concepts of PyTorch

- Modules of PyTorch

Data:
- Tensor
- **Variable (for Gradient)**

Function:
- NN Modules
- Optimizer
- Loss Function
- Multi-Processing

- Variable



Tensor data

For Current Backward Process

Handled by PyTorch Automatically

Concepts of PyTorch

- Modules of PyTorch

  Data:
  - Tensor
  - **Variable (for Gradient)**

  Function:
  - NN Modules
  - Optimizer
  - Loss Function
  - Multi-Processing

- Variable
  - x = Variable(torch.ones(2, 2), requires_grad=True)
  - print(x)

Out:
```
Variable containing:
 1  1
 1  1
[torch.FloatTensor of size 2x2]
```

  - y = x + 2
  - z = y * y * 3
  - out = z.mean()
  - out.backward()
  - print(x.grad)

Out:
```
Variable containing:
 4.5000  4.5000
 4.5000  4.5000
[torch.FloatTensor of size 2x2]
```

$$\text{out} = \frac{1}{4}\sum z_i$$

$$z_i = 3y_i^2 = 3(x_i + 2)^2$$

$$\frac{\partial \text{out}}{\partial x_i} = \frac{3}{2}(x_i + 2) = \frac{9}{2}$$

```python
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules
(must have)

Build network
(must have)

```python
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

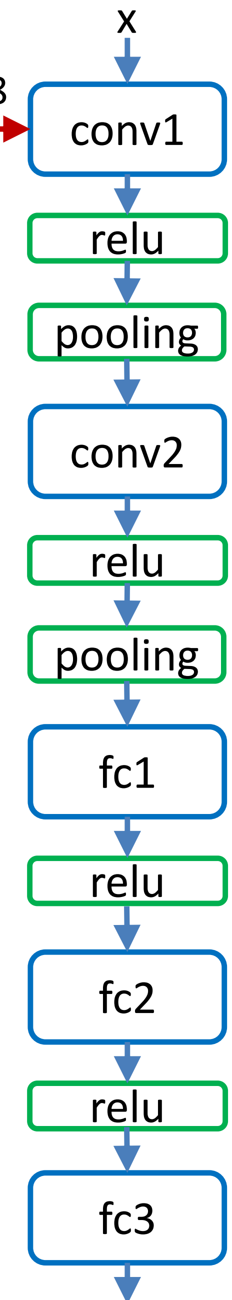
- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules
(must have)

Build network
(must have)

```python
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

[Channel, H, W]: 1x32x32->6x28x28



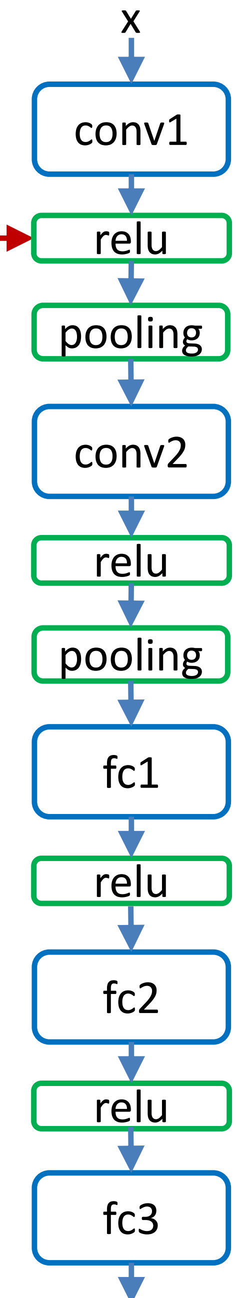
- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

```python
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

Define modules (must have)

Build network (must have)

[Channel, H, W]: 6x28x28



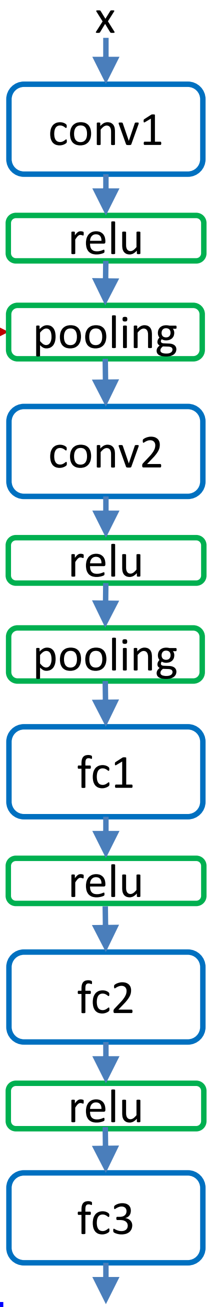
- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

[Channel, H, W]: 6x28x28 -> 6x14x14

```python
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```



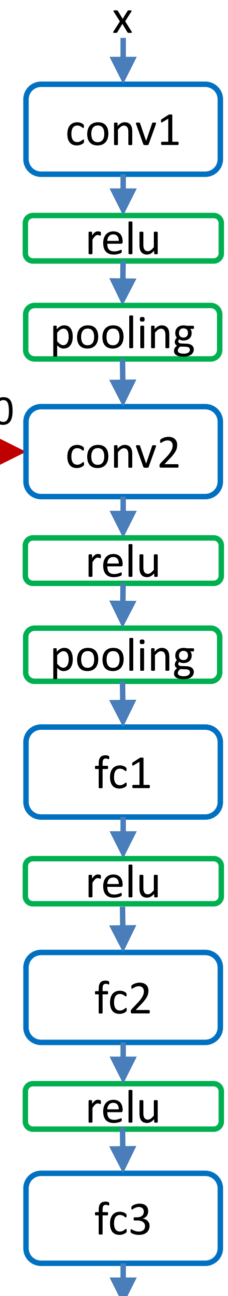
- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

```python
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

Define modules (must have)

Build network (must have)

[Channel, H, W]: 6x14x14 -> 16x10x10



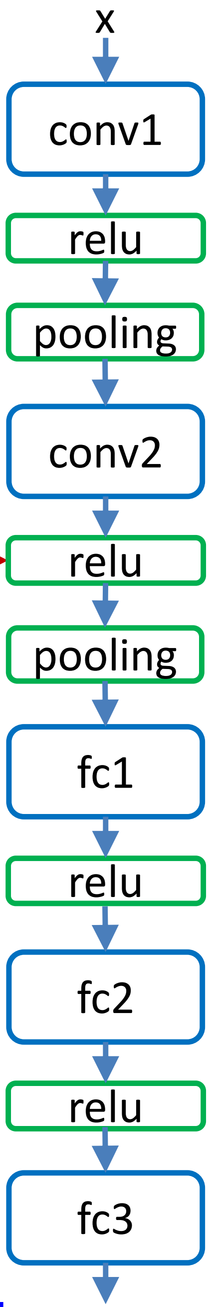
- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

```python
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

Define modules (must have)

Build network (must have)

[Channel, H, W]: 16x10x10



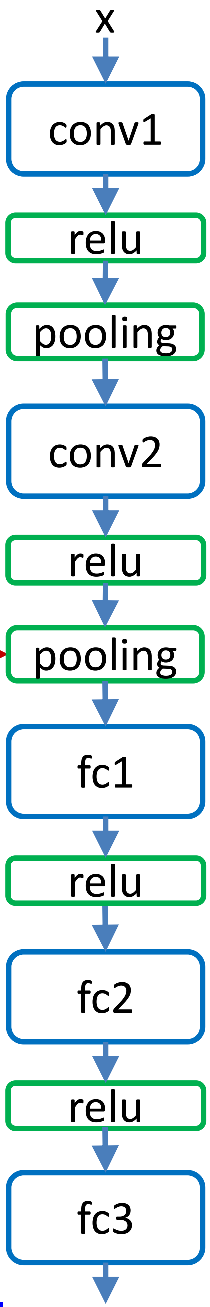
- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules
(must have)

Build network
(must have)

```python
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

[Channel, H, W]: 16x10x10 -> 16x5x5



- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

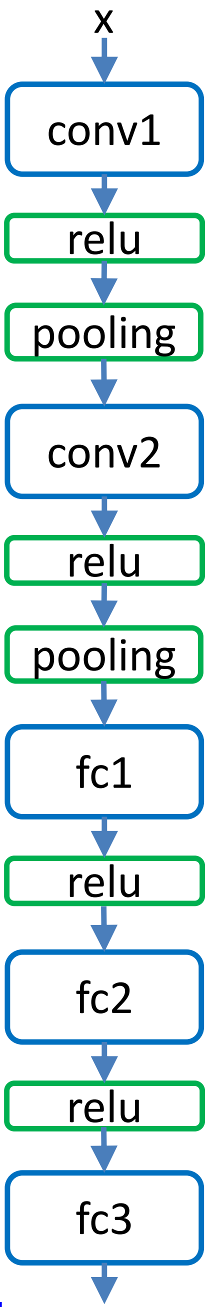
```python
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

Define modules (must have)

Build network (must have)

Flatten the Tensor
16x5x5

Tensor: [Batch N, Channel, H, W]

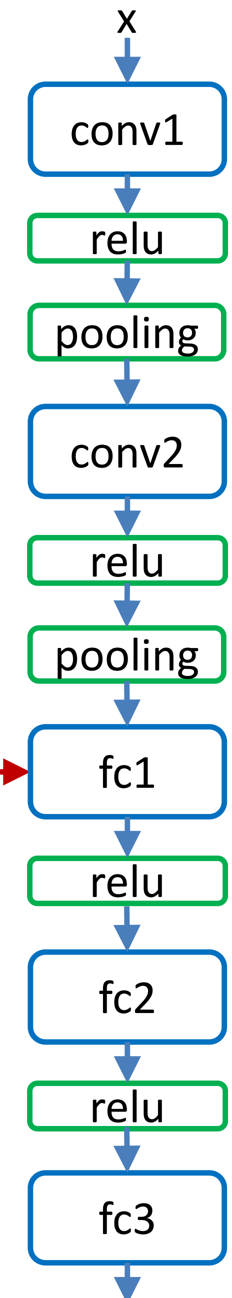


- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

```python
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

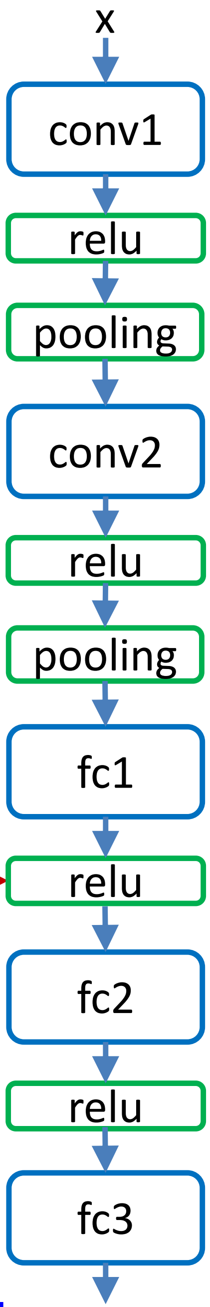


- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

```python
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

Define modules (must have)

Build network (must have)

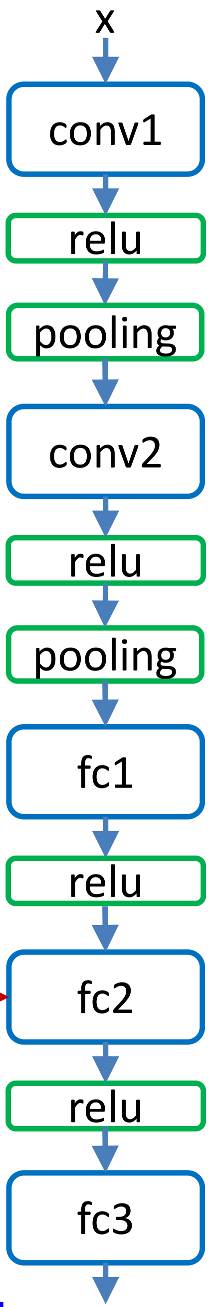


- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

```
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

Define modules (must have)

Build network (must have)

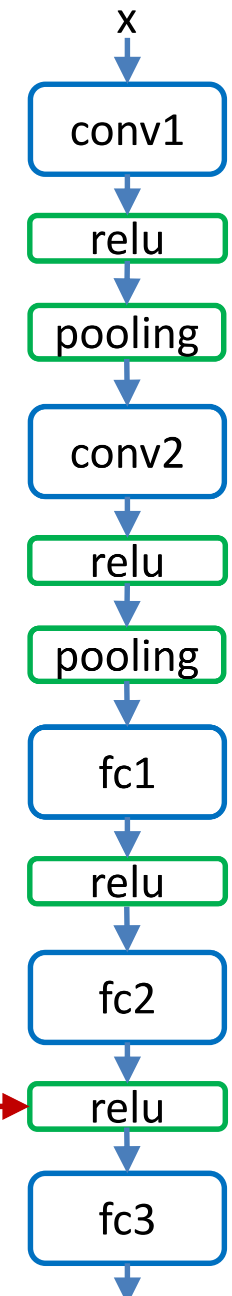


- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

```python
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

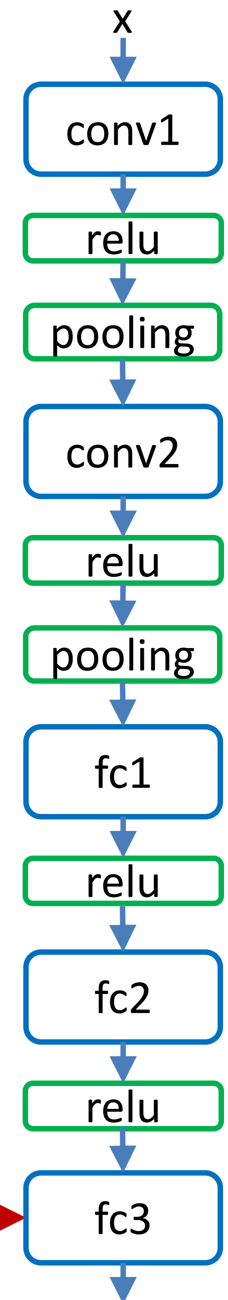


- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Define modules (must have)

Build network (must have)

```
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```



- http://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#define-the-network

Concepts of PyTorch

- Modules of PyTorch

  Data:
  - Tensor
  - Variable (for Gradient)

  Function:
  - **NN Modules**
  - Optimizer
  - Loss Function
  - Multi-Processing

- NN Modules (torch.nn)
  - Modules built on Variable
  - Gradient handled by PyTorch

- Common Modules
  - Convolution layers
  - Linear layers
  - Pooling layers
  - Dropout layers
  - Etc…

NN Modules

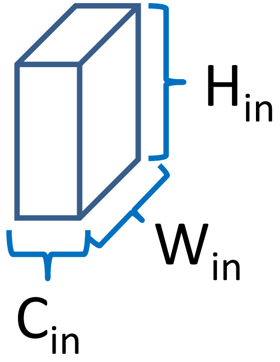
- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D

  - Example:
  - torch.nn.conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)

NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D

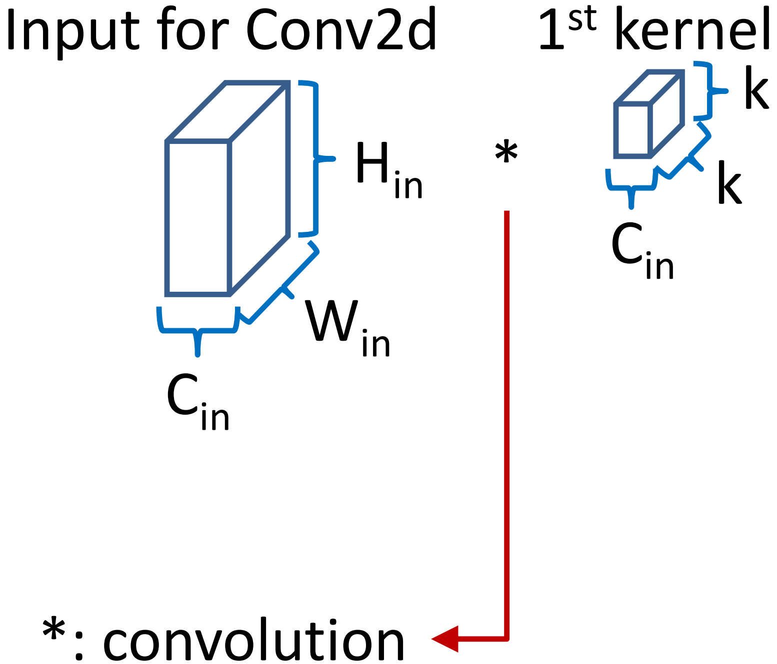
Input for Conv2d



*: convolution

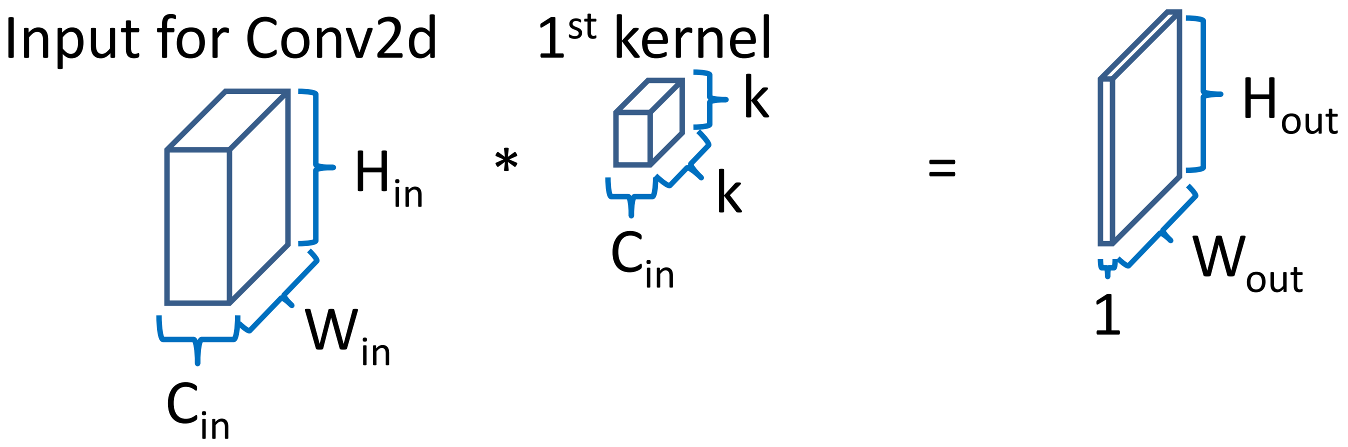
NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D



NN Modules

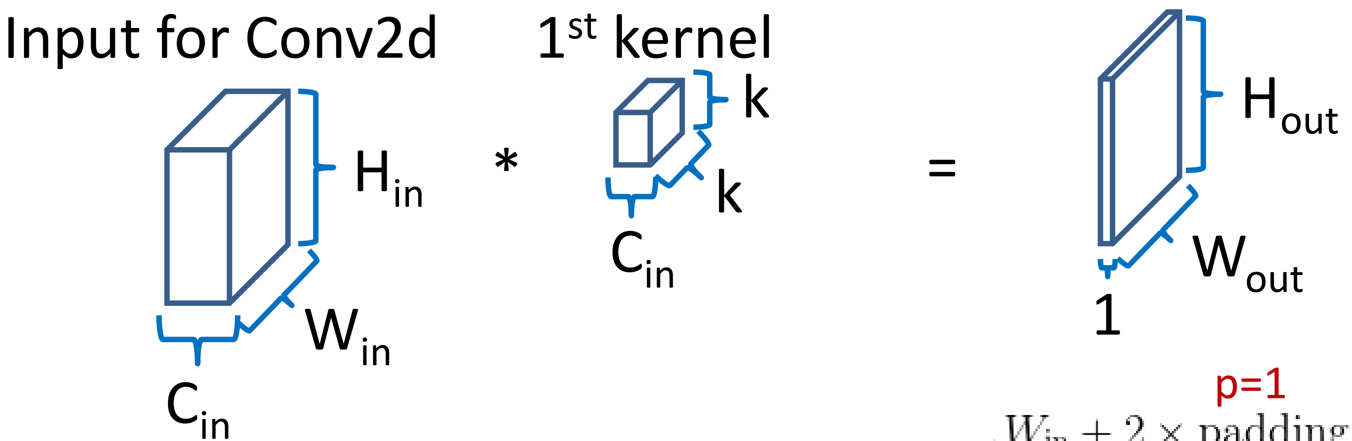
- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D



*: convolution

NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D



p=1 d=1 k=3

$$W_{\text{out}} = \text{floor}(\frac{W_{\text{in}} + 2 \times \text{padding} - \text{dilation} \times (k-1) - 1}{\text{stride}} + 1)$$

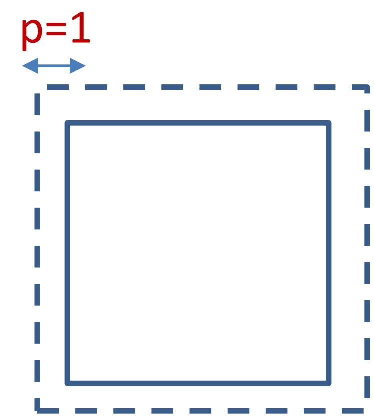
s=1, moving step size

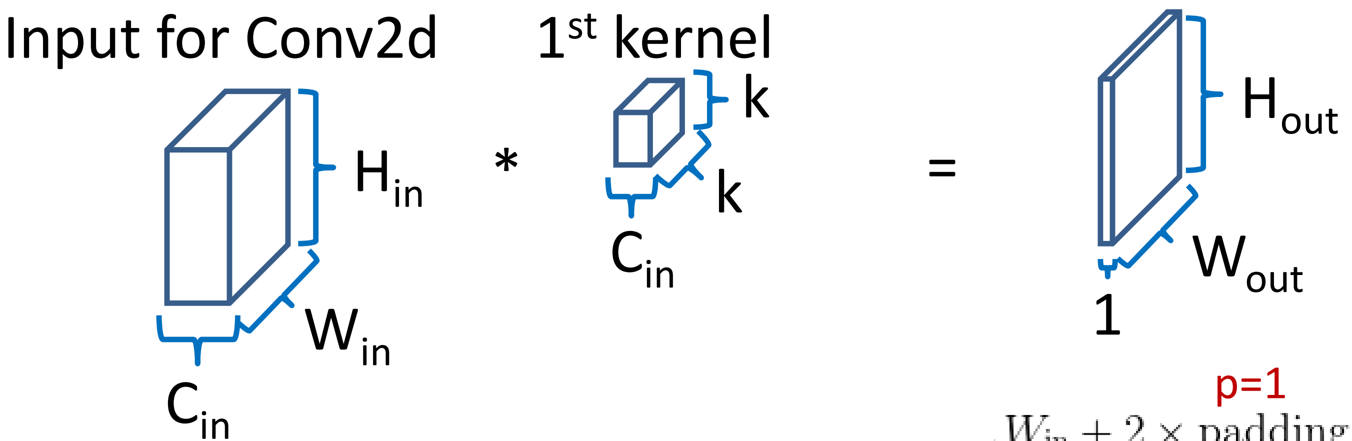
$$W_{\text{out}} = \text{floor}(\frac{W_{\text{in}} + 2 \times 1 - 1 \times (3-1) - 1}{1} + 1) = W_{\text{in}}$$

*: convolution

NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D





p=1 d=1 k=3

$$W_{\text{out}} = \text{floor}\left(\frac{W_{\text{in}} + 2 \times \text{padding} - \text{dilation} \times (k-1) - 1}{\text{stride}} + 1\right)$$

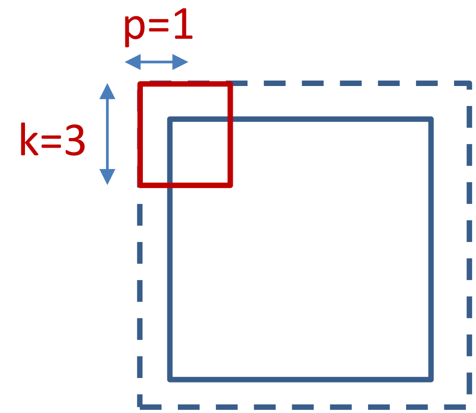
s=1, moving step size

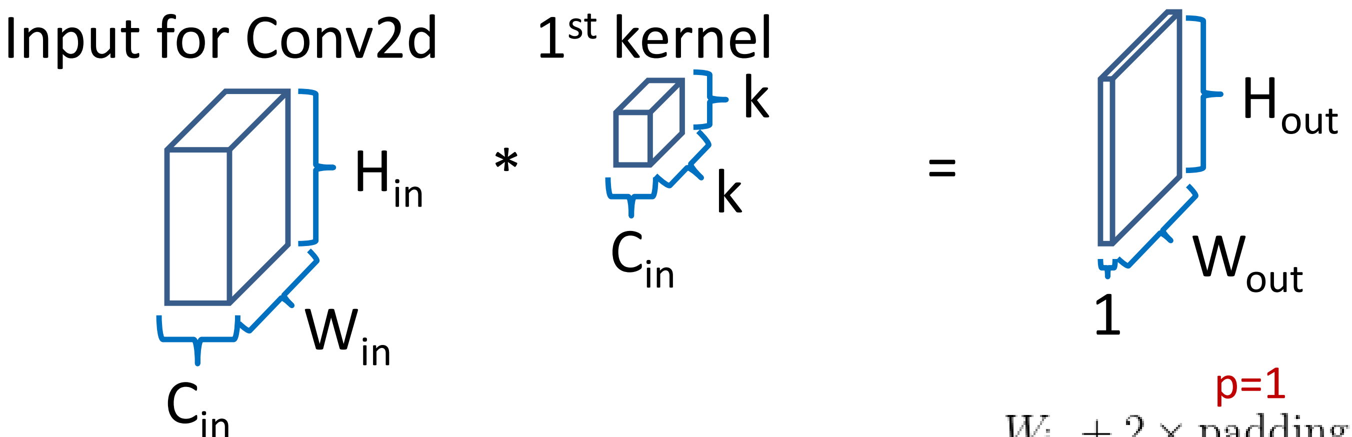
$$W_{\text{out}} = \text{floor}\left(\frac{W_{\text{in}} + 2 \times 1 - 1 \times (3-1) - 1}{1} + 1\right) = W_{\text{in}}$$

*: convolution

NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D





p=1 d=1 k=3

$$W_{out} = \text{floor}(\frac{W_{in} + 2 \times \text{padding} - \text{dilation} \times (k-1) - 1}{\text{stride}} + 1)$$

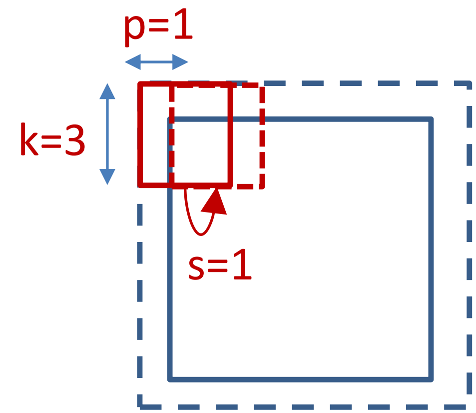
s=1, moving step size

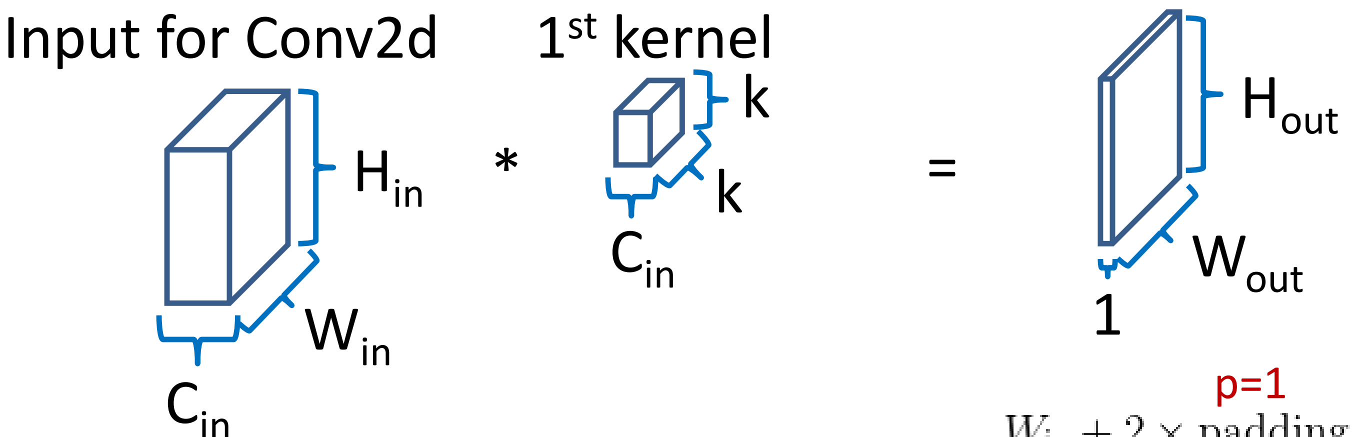
$$W_{out} = \text{floor}(\frac{W_{in} + 2 \times 1 - 1 \times (3-1) - 1}{1} + 1) = W_{in}$$

*: convolution

NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D





p=1 d=1 k=3

$$W_{\text{out}} = \text{floor}\left(\frac{W_{\text{in}} + 2 \times \text{padding} - \text{dilation} \times (k-1) - 1}{\text{stride}} + 1\right)$$

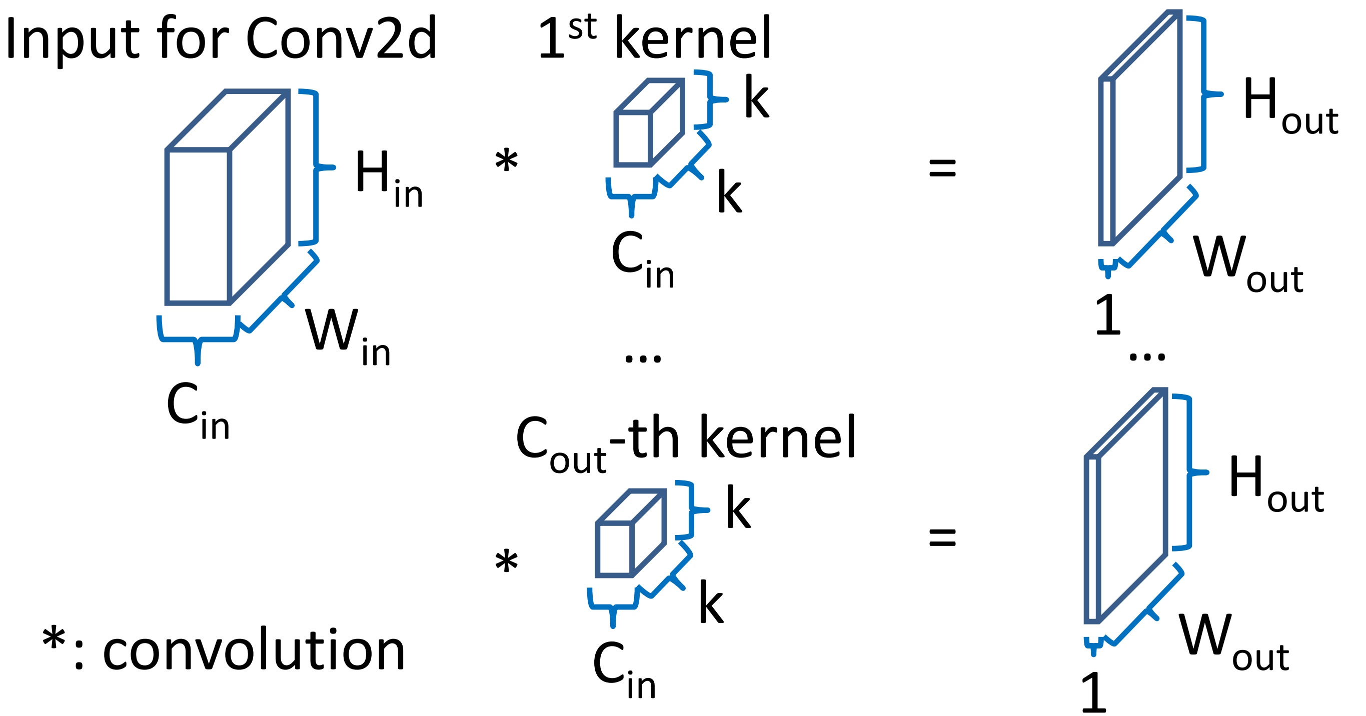
s=1, moving step size

$$W_{\text{out}} = \text{floor}\left(\frac{W_{\text{in}} + 2 \times 1 - 1 \times (3-1) - 1}{1} + 1\right) = W_{\text{in}}$$

*: convolution

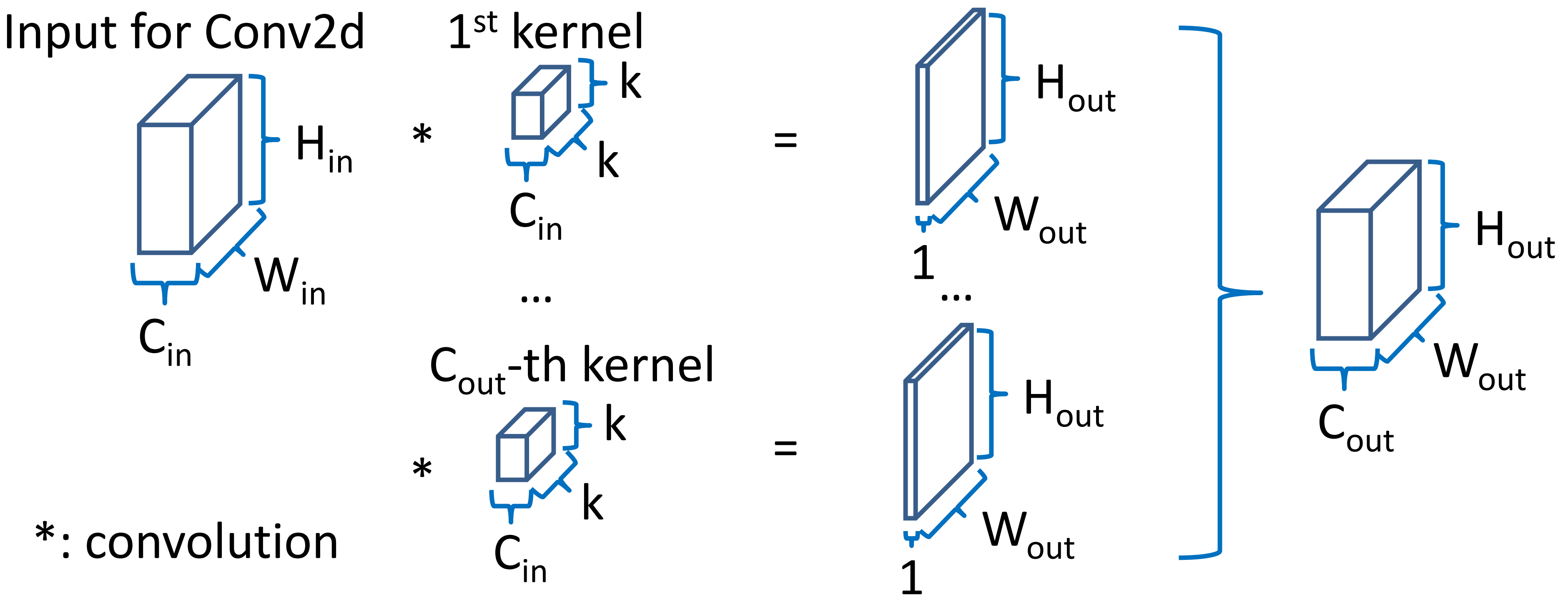
NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D



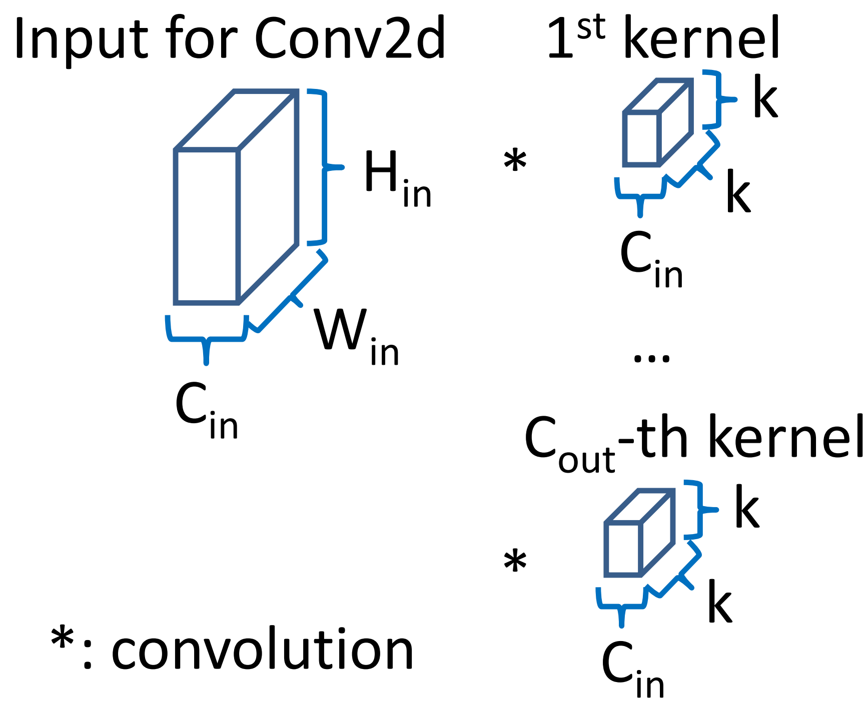
NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D



NN Modules

- Convolution Layer
  - N-th Batch (N), Channel (C)
  - torch.nn.Conv1d: input [N, C, W] # moving kernel in 1D
  - torch.nn.Conv2d: input [N, C, H, W] # moving kernel in 2D
  - torch.nn.Conv3d: input [N, C, D, H, W] # moving kernel in 3D

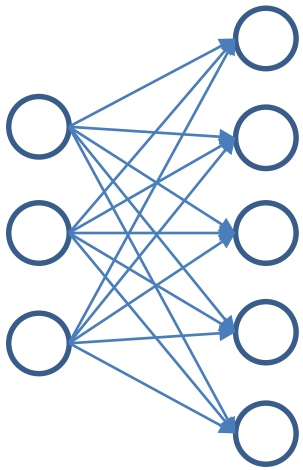


# of parameters

$$O(C_{\text{in}} \times k^2 \times C_{\text{out}})$$

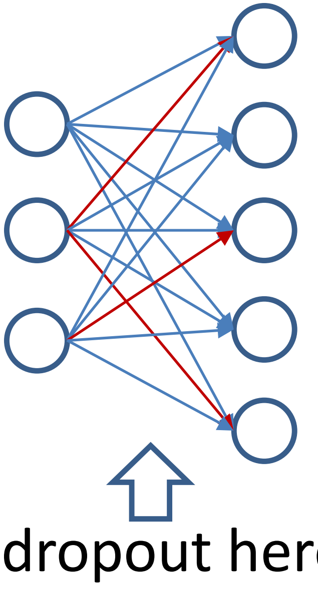
NN Modules

- Linear Layer
  - torch.nn.Linear(in_features=3, out_features=5)
  - y=Ax+b



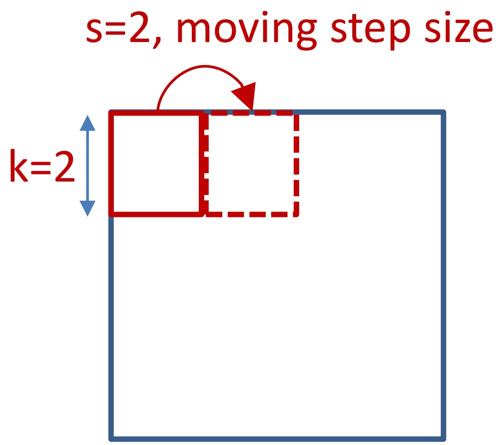
NN Modules

- Dropout Layer
  - torch.nn.Dropout(p)
  - Random zeros the input with probability p
  - Output are scaled by 1/(1-p)



NN Modules

- Pooling Layer
  - torch.nn.AvgPool2d(kernel_size=2, stride=2, padding=0)
  - torch.nn.MaxPool2d(kernel_size=2, stride=2, padding=0)



$$W_{\text{out}} = \text{floor}\left(\frac{W_{\text{in}} + 2 \times \overset{p=0}{\text{padding}} - \overset{d=1}{\text{dilation}} \times (\overset{k=2}{k} - 1) - 1}{\underset{\text{s=2, moving step size}}{\text{stride}}} + 1\right)$$

$$W_{\text{out}} = \text{floor}\left(\frac{W_{\text{in}} + 2 \times 0 - 1 \times (2 - 1) - 1}{2} + 1\right) = \frac{W_{\text{in}}}{2}$$

Concepts of PyTorch

- Modules of PyTorch

  Data:
  - Tensor
  - Variable (for Gradient)

  Function:
  - **NN Modules**
  - Optimizer
  - Loss Function
  - Multi-Processing

- NN Modules (torch.nn)
  - Modules built on Variable
  - Gradient handled by PyTorch

- Common Modules
  - Convolution layers
  - Linear layers
  - Pooling layers
  - Dropout layers
  - Etc…

Concepts of PyTorch

- Modules of PyTorch

  Data:
  - Tensor
  - Variable (for Gradient)

  Function:
  - NN Modules
  - **Optimizer**
  - **Loss Function**
  - Multi-Processing

- Optimizer (torch.optim)
  - SGD
  - Adagrad
  - Adam
  - RMSprop
  - …
  - 9 Optimizers (PyTorch 0.2)

- Loss (torch.nn)
  - L1Loss
  - MSELoss
  - CrossEntropy
  - …
  - 18 Loss Functions (PyTorch 0.2)

What We Build?

Define modules (must have)

Build network (must have)

```python
# -*- coding: utf-8 -*-
import torch
from torch.autograd import Variable


class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables.
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs, and wrap them in Variables
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    print(t, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

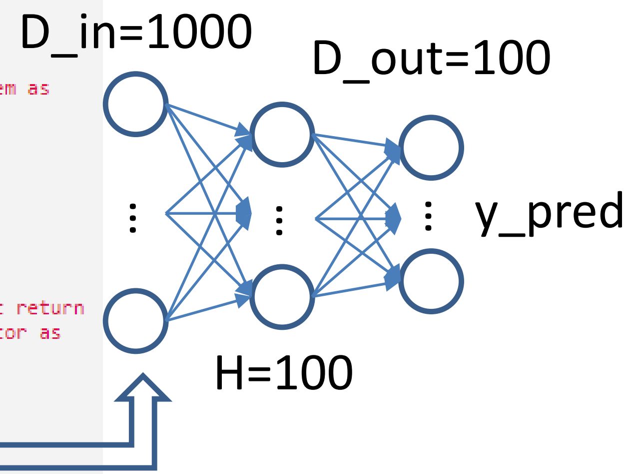
http://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-optim

What We Build?

Define modules (must have)

Build network (must have)

```python
# -*- coding: utf-8 -*-
import torch
from torch.autograd import Variable


class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables.
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred


# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs, and wrap them in Variables
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    print(t, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```



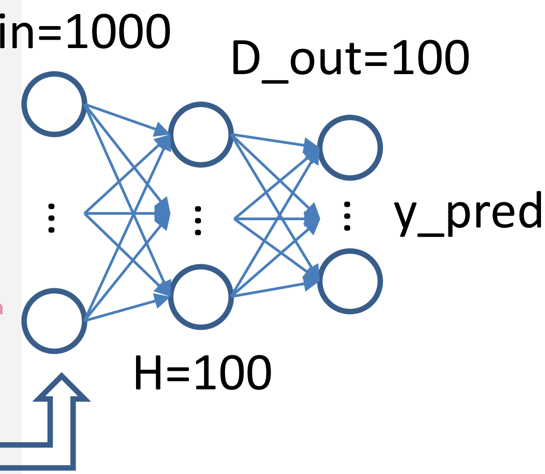
http://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-optim

```python
# -*- coding: utf-8 -*-
import torch
from torch.autograd import Variable


class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables.
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred


# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs, and wrap them in Variables
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    print(t, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Define modules (must have)

Build network (must have)

What We Build?



Don't Update y (y are labels here)

Construct Our Model

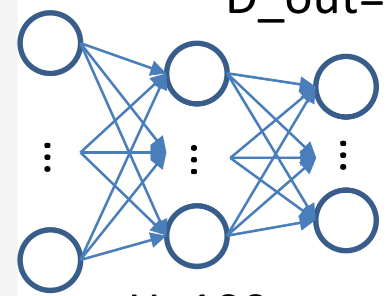
Optimizer and Loss Function

http://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-optim

What We Build?

D_in=1000

D_out=100



y_pred

H=100

```
# -*- coding: utf-8 -*-
import torch
from torch.autograd import Variable


class TwoLayerNet(torch.nn.Module):
```

Define modules (must have)

```
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables.
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)
```

Build network (must have)

```
    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred


# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs, and wrap them in Variables
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)
```

Don’t Update y (y are labels here)

```
# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)
```

Construct Our Model

```
# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
```

Optimizer and Loss Function

```
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    print(t, loss.data[0])
```

Reset Gradient
Backward
Update Step

```
    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

http://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-optim

Concepts of PyTorch

- Modules of PyTorch

  Data:
  - Tensor
  - Variable (for Gradient)

  Function:
  - NN Modules
  - Optimizer
  - Loss Function
  - **Multi-Processing**

- Basic Method
  - torch.nn.DataParallel
  - Recommend by PyTorch

- Advanced Methods
  - torch.multiprocessing
  - Hogwild (async)

Multi-GPU Processing

- torch.nn.DataParallel
  - gpu_id = '6,7‘
  - os.environ['CUDA_VISIBLE_DEVICES'] = gpu_id
  - net = torch.nn.DataParallel(model, device_ids=[0, 1, 2])
  - output = net(input_var)

- Important Notes:
  - Device_ids must start from 0
  - (batch_size/GPU_size) must be integer

Saving Models

- First Approach (Recommend by PyTorch)
  - # save only the model parameters
  - torch.save(the_model.state_dict(), PATH)

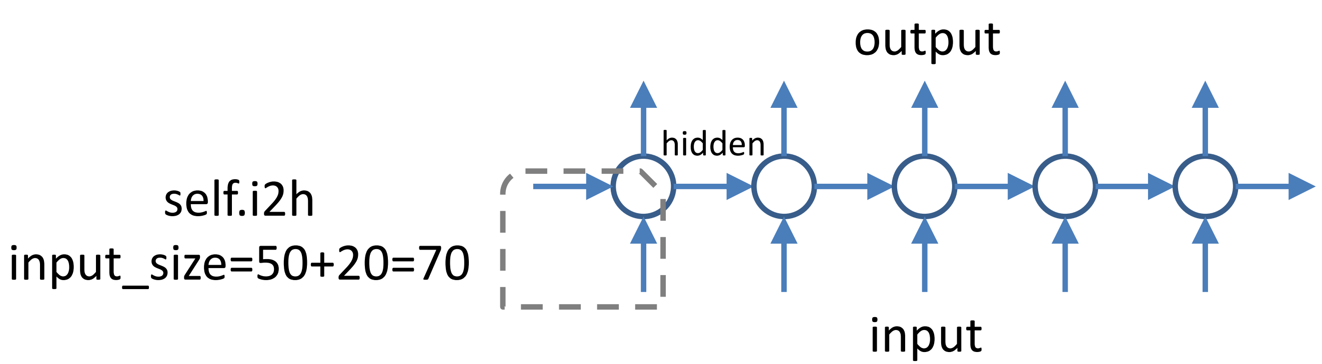
  - # load only the model parameters
  - the_model = TheModelClass(*args, **kwargs)
  - the_model.load_state_dict(torch.load(PATH))
- Second Approach
  - torch.save(the_model, PATH) # save the entire model
  - the_model = torch.load(PATH) # load the entire model

http://pytorch.org/docs/master/notes/serialization.html#recommended-approach-for-saving-a-model

Recurrent Neural Network (RNN)

```
class RNN(nn.Module):

    # you can also accept arguments in your model constructor
    def __init__(self, data_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        input_size = data_size + hidden_size

        self.i2h = nn.Linear(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)

    def forward(self, data, last_hidden):
        input = torch.cat((data, last_hidden), 1)
        hidden = self.i2h(input)
        output = self.h2o(hidden)
        return hidden, output


rnn = RNN(50, 20, 10)
```

```
loss_fn = nn.MSELoss()

batch_size = 10
TIMESTEPS = 5

# Create some fake data
batch = Variable(torch.randn(batch_size, 50))
hidden = Variable(torch.zeros(batch_size, 20))
target = Variable(torch.zeros(batch_size, 10))

loss = 0
for t in range(TIMESTEPS):
    # yes! you can reuse the same network several times,
    # sum up the losses, and call backward!
    hidden, output = rnn(batch, hidden)
    loss += loss_fn(output, target)
loss.backward()
```



http://pytorch.org/tutorials/beginner/former_torchies/nn_tutorial.html#example-2-recurrent-net

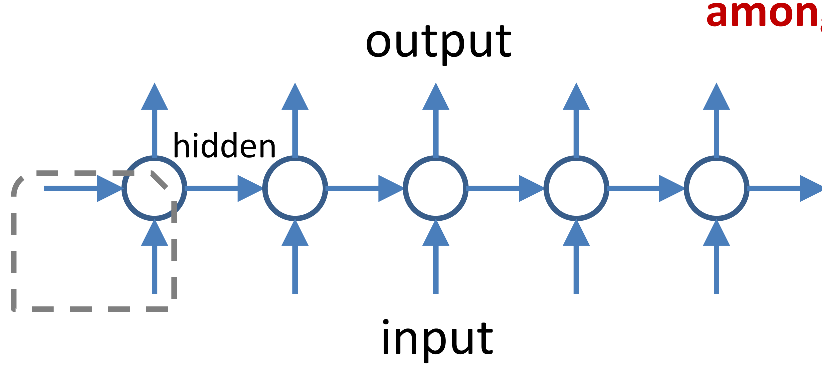
Recurrent Neural Network (RNN)

```python
class RNN(nn.Module):

    # you can also accept arguments in your model constructor
    def __init__(self, data_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        input_size = data_size + hidden_size

        self.i2h = nn.Linear(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)

    def forward(self, data, last_hidden):
        input = torch.cat((data, last_hidden), 1)
        hidden = self.i2h(input)
        output = self.h2o(hidden)
        return hidden, output


rnn = RNN(50, 20, 10)
```

```python
loss_fn = nn.MSELoss()

batch_size = 10
TIMESTEPS = 5

# Create some fake data
batch = Variable(torch.randn(batch_size, 50))
hidden = Variable(torch.zeros(batch_size, 20))
target = Variable(torch.zeros(batch_size, 10))

loss = 0
for t in range(TIMESTEPS):
    # yes! you can reuse the same network several times,
    # sum up the losses, and call backward!
    hidden, output = rnn(batch, hidden)
    loss += loss_fn(output, target)
loss.backward()
```

Same module (i.e. same parameters) among the time

self.i2h
input_size=50+20=70



http://pytorch.org/tutorials/beginner/former_torchies/nn_tutorial.html#example-2-recurrent-net

Transfer Learning

- Freeze the parameters of original model
  - requires_grad = False

- Then add your own modules

```
model = torchvision.models.resnet18(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
# Replace the last fully-connected layer
# Parameters of newly constructed modules have requires_grad=True by default
model.fc = nn.Linear(512, 100)

# Optimize only the classifier
optimizer = optim.SGD(model.fc.parameters(), lr=1e-2, momentum=0.9)
```

http://pytorch.org/docs/master/notes/autograd.html#excluding-subgraphs-from-backward

Comparison with TensorFlow

Properties	TensorFlow	PyTorch
Graph	Static Dynamic (TensorFlow Fold)	Dynamic
Ramp-up Time	-	Win
Graph Creation and Debugging	-	Win
Feature Coverage	Win	Catch up quickly
Documentation	Tie	Tie
Serialization	Win (support other lang.)	-
Deployment	Win (Cloud & Mobile)	-
Data Loading	-	Win
Device Management	Win	Need .cuda()
Custom Extensions	-	Win

Summarized from https://awni.github.io/pytorch-tensorflow/

Remind: Platform & Final Project

Thank You~!