PyTorch

https://pytorch.org/

Open Source - Deep Learning Frameworks

Chainer
mxnet
Caffe2
Microsoft CNTK
TensorFlow
Keras
GLUON
PYTORCH
theano

Why do we need a special Deep Learning Framework ?

KNN from scratch

VS

KNN from Sklearn

Python code for Neural Network

This is just a 2-layered MLP !!

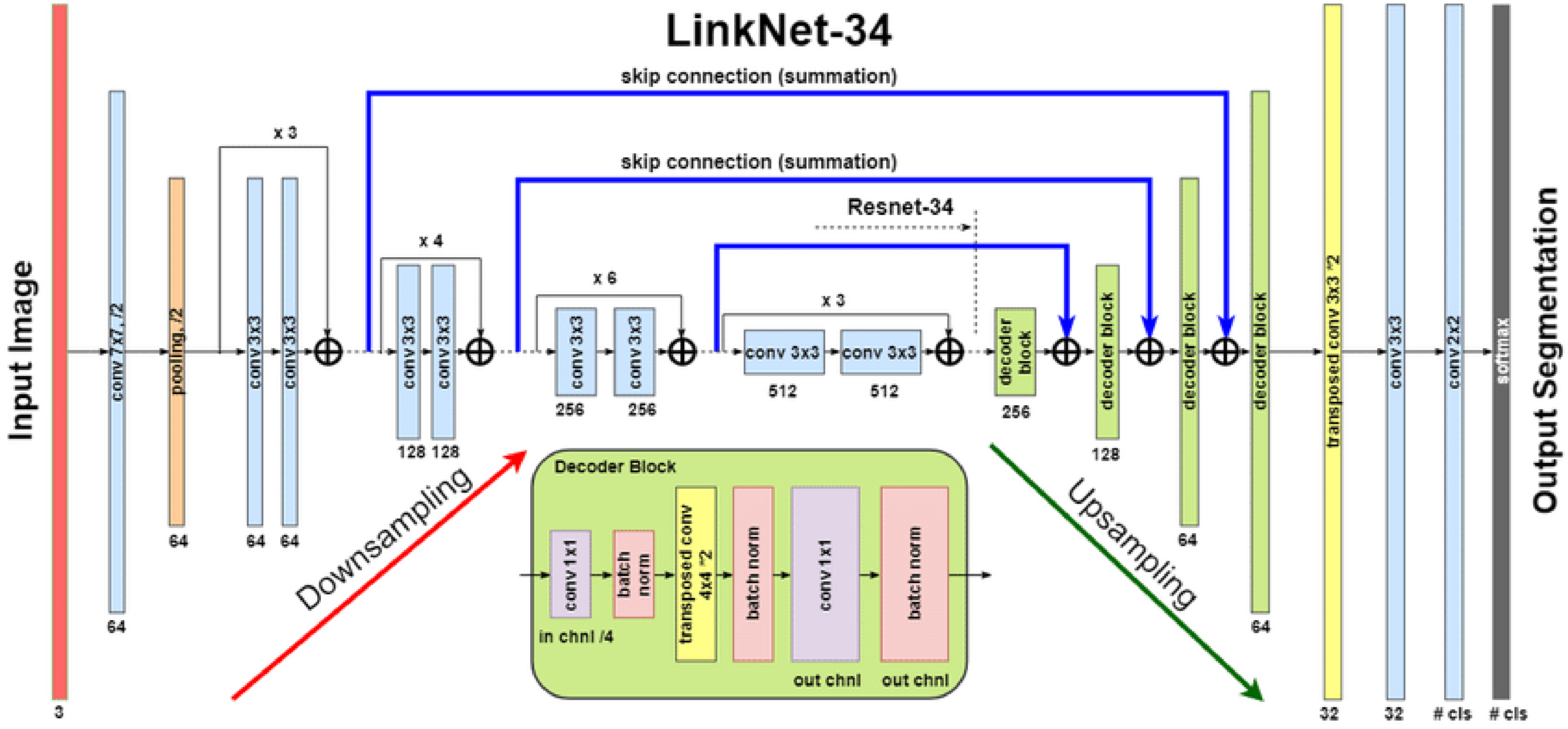
NSE talent sprint | IIIT Hyderabad

```python
def build_model(nn_hdim, num_passes=20000, print_loss=False):
    np.random.seed(0)
    W1 = np.random.randn(nn_input_dim, nn_hdim) / np.sqrt(nn_input_dim)
    b1 = np.zeros((1, nn_hdim))
    W2 = np.random.randn(nn_hdim, nn_output_dim) / np.sqrt(nn_hdim)
    b2 = np.zeros((1, nn_output_dim))

    # This is what we return at the end
    model = {}

    # Gradient descent. For each batch...
    for i in range(0, num_passes):

        # Forward propagation
        z1 = X.dot(W1) + b1
        a1 = np.tanh(z1)
        z2 = a1.dot(W2) + b2
        exp_scores = np.exp(z2)
        probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

        # Backpropagation
        delta3 = probs
        delta3[range(num_examples), y] -= 1
        dW2 = (a1.T).dot(delta3)
        db2 = np.sum(delta3, axis=0, keepdims=True)
        delta2 = delta3.dot(W2.T) * (1 - np.power(a1, 2))
        dW1 = np.dot(X.T, delta2)
        db1 = np.sum(delta2, axis=0)

        # Add regularization terms (b1 and b2 don't have regularization terms)
        dW2 += reg_lambda * W2
        dW1 += reg_lambda * W1

        # Gradient descent parameter update
        W1 += -epsilon * dW1
        b1 += -epsilon * db1
        W2 += -epsilon * dW2
        b2 += -epsilon * db2

        # Assign new parameters to the model
        model = { 'W1': W1, 'b1': b1, 'W2': W2, 'b2': b2}

        # Optionally print the loss.
        # This is expensive because it uses the whole dataset, so we don't want to do it too often.
        if print_loss and i % 1000 == 0:
            print("Loss after iteration %i: %f" %(i, calculate_loss(model)))

    return model
```

```python
# Helper function to evaluate the total loss on the dataset
def calculate_loss(model):
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']
    # Forward propagation to calculate our predictions
    z1 = X.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    # Calculating the loss
    corect_logprobs = -np.log(probs[range(num_examples), y])
    data_loss = np.sum(corect_logprobs)
    # Add regulatization term to loss (optional)
    data_loss += reg_lambda/2 * (np.sum(np.square(W1)) + np.sum(np.square(W2)))
    return 1./num_examples * data_loss
```

```python
# Helper function to predict an output (0 or 1)
def predict(model, x):
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']
    # Forward propagation
    z1 = x.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    return np.argmax(probs, axis=1)
```

What about coding for this network



And finding the gradients !!

Standard Deep Learning Framework

Computational graphs

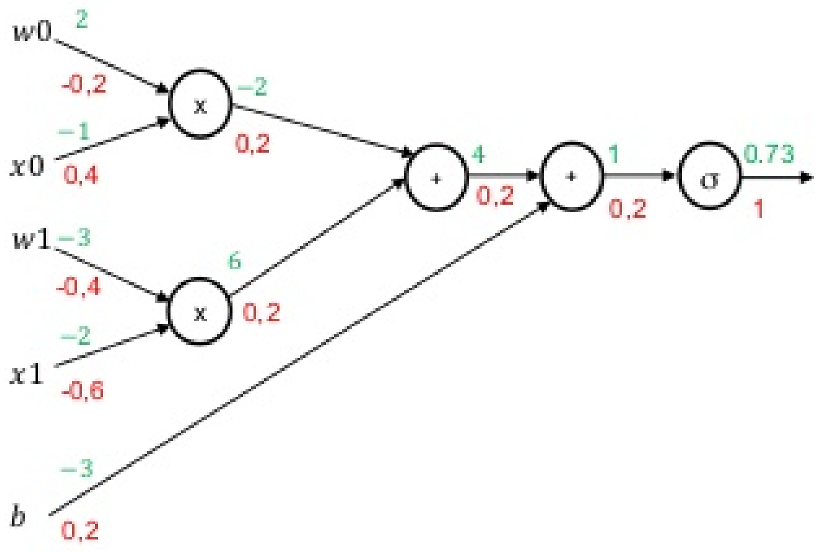
Numerical Examples

$$f(x,y,z) = \sigma(w_0x_0 + w_1x_1 + b)$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{d\sigma(x)}{x} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{(1+e^{-x})}\right)\left(\frac{1}{(1+e^{-x})}\right)$$

$$\frac{d\sigma(x)}{x} = (1-\sigma(x))(\sigma(x))$$



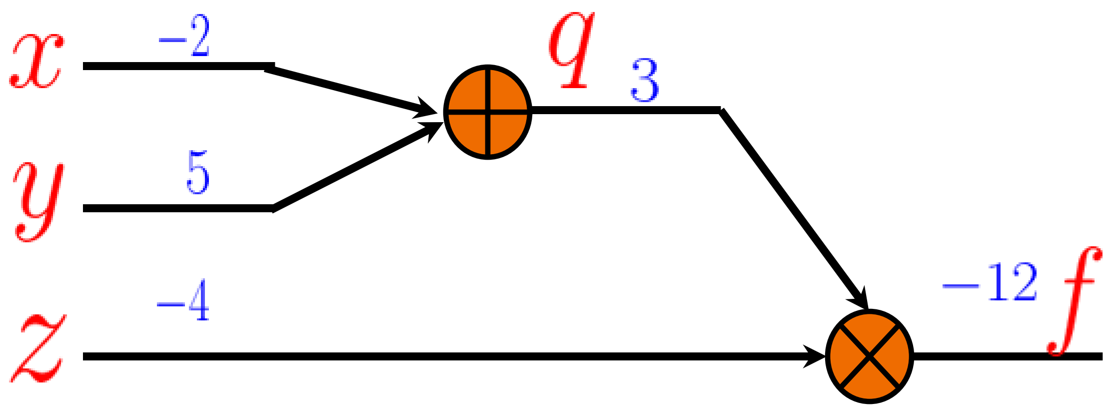
From Stanford Course: Convolutional Neural Networks for Visual Recognition

What is computational graph?

- A directed graph, where every node represents a mathematical operation and edges represent the variables which are to be operated.

$$f(x, y, z) = (x + y)z$$

$$eg: x = -2, y = 5, z = -4$$



Types of Computational Graphs

Static	Dynamic
Cannot add nodes at runtime	Allows addition of nodes at runtime
Building graph is separated from execution	Every execution there is a new graph

Standard Deep Learning Framework

Build and operate Computational Graphs

Auto-differentiation

Compute and take derivatives of huge composition functions

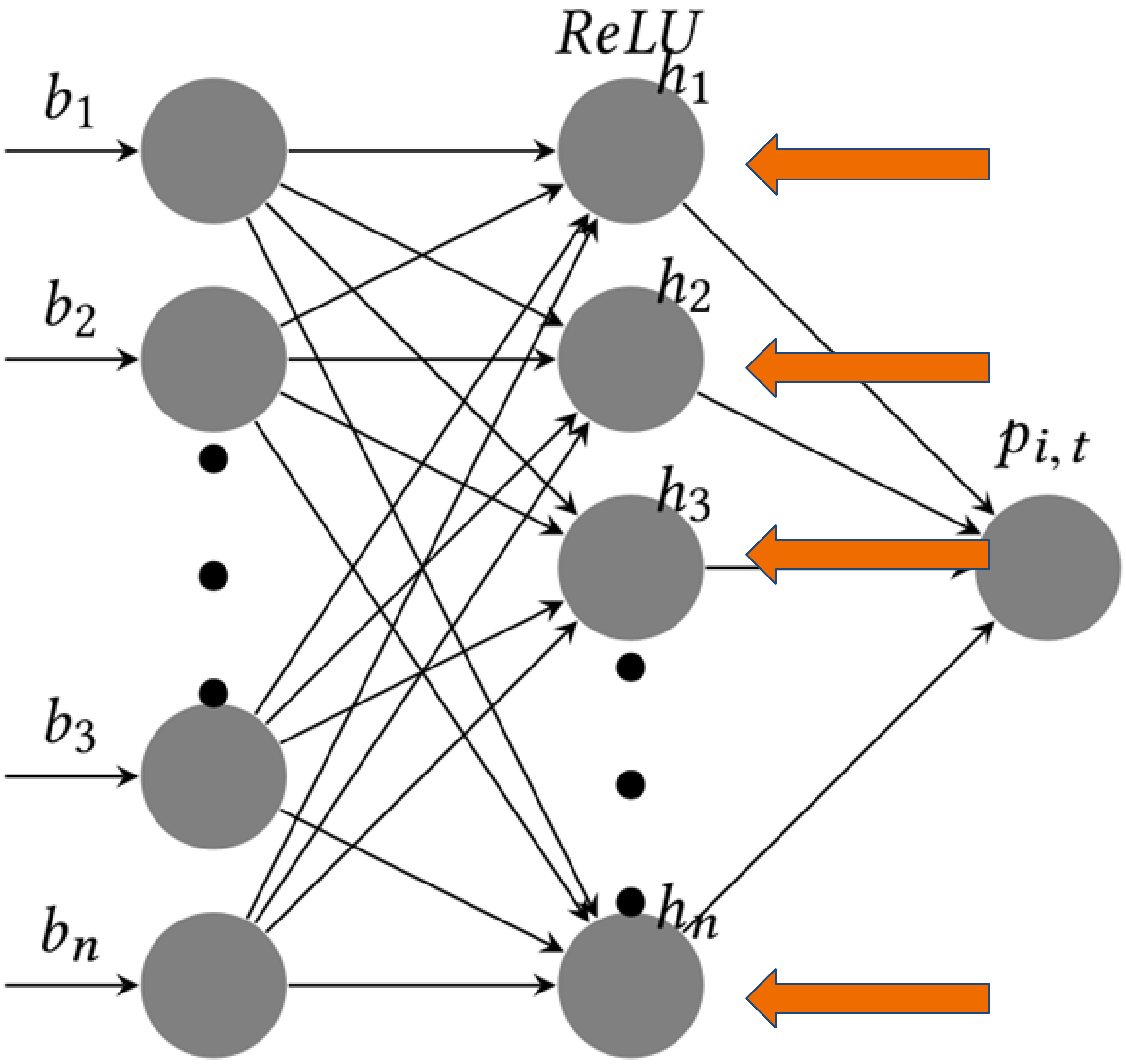
Standard Deep Learning Framework

Build and operate Computational Graphs

Auto-differentiation

Parallelizing on GPU

Parallelizing



GPU
Many cores

Standard Deep Learning Framework

Build and operate Computational Graphs

Perform forward and backward propagation

Parallelizing on GPU

Provide with Standard Architectures and other widely used primitives

Pytorch

FAIR (Facebook Artificial Intelligence Research)

Based on Torch

Dynamic Computational graph

Easy to implement, debug,

Developer Friendly

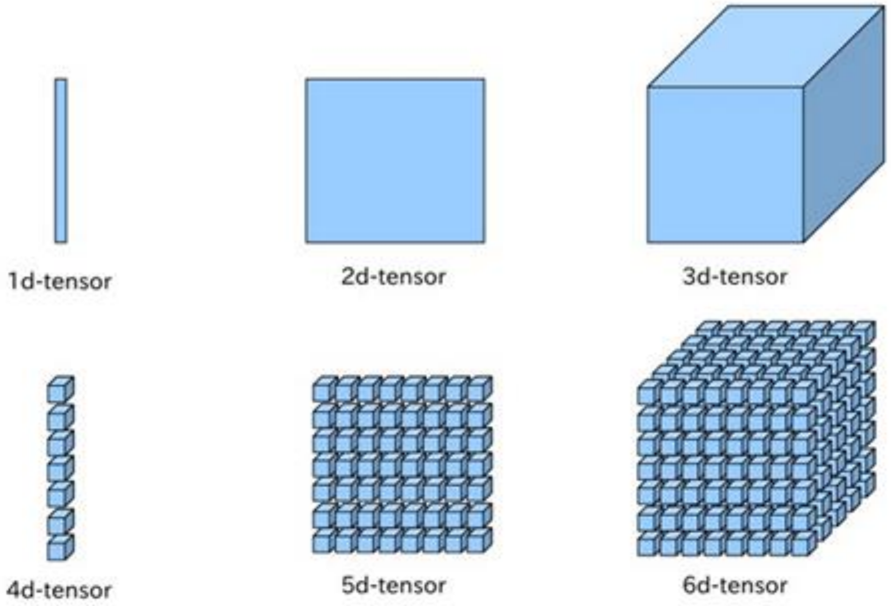
Tensor computation with strong GPU acceleration

Efficient Memory Usage

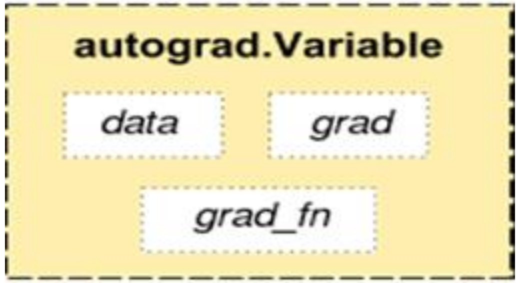
Basics of Pytorch

The building blocks

Tensors



Variables



- A Variable class wraps a tensor. You can access this tensor by calling .data attribute of a Variable.
- The Variable also stores the gradient of a scalar quantity (say, loss) with respect to the parameter it holds. This gradient can be accessed by calling the .grad attribute.
- The third attribute a Variable holds is a grad_fn, a Function object which created the variable.
  - c = a + b. Then c is a new variable, and it's grad_fn is something called AddBackward

The building blocks

Tensors

Like Numpy arrays but can run on GPU

```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)
```

GPU

```
import torch

dtype = torch.cuda.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)
```

The building blocks

Variables are nodes in a computational graph which store the data and gradient

Variables

```
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)
```

Difference

```python
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t 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_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

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

```python
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()
    loss.backward()
    w1.data-=learning_rate*w1.grad
    w2.data-=learning_rate*w2.grad
    w1.grad.data.zero_()
    w2.grad.data.zero_()
```

Neural Network

torch.nn

NN module

Higher-level wrappers for working with neural nets

- Layers
- Activation Functions
- Loss functions

⊟ torch.nn

- Parameters
- ⊞ Containers
- ⊟ Convolution Layers
  - Conv1d
  - Conv2d
  - Conv3d
  - ConvTranspose1d
  - ConvTranspose2d
  - ConvTranspose3d

Other layers:
Dropout, Linear,
Normalization Layer

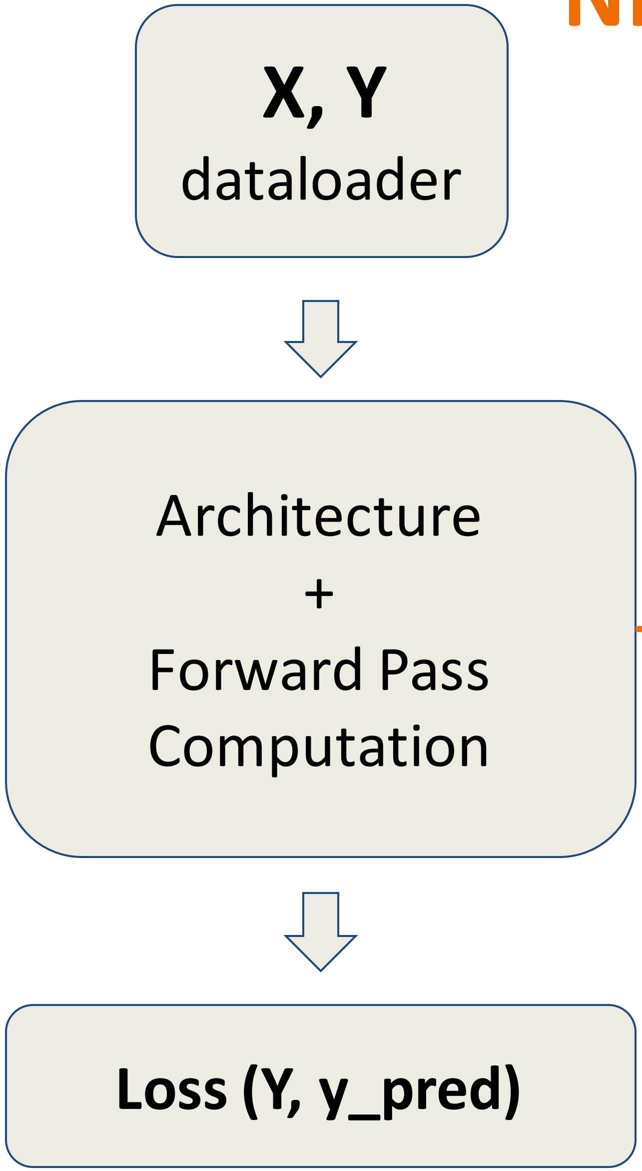
⊟ torch.nn

- Parameters
- ⊞ Containers
- ⊞ Convolution Layers
- ⊟ Pooling Layers
  - MaxPool1d
  - MaxPool2d
  - MaxPool3d
  - MaxUnpool1d
  - MaxUnpool2d
  - MaxUnpool3d
  - AvgPool1d
  - AvgPool2d
  - AvgPool3d
  - FractionalMaxPool2d
  - LPPool2d
  - AdaptiveMaxPool1d
  - AdaptiveMaxPool2d
  - AdaptiveMaxPool3d
  - AdaptiveAvgPool1d
  - AdaptiveAvgPool2d
  - AdaptiveAvgPool3d

⊟ Loss functions

- L1Loss
- MSELoss
- CrossEntropyLoss
- NLLLoss
- PoissonNLLLoss
- KLDivLoss
- BCELoss
- BCEWithLogitsLoss
- MarginRankingLoss
- HingeEmbeddingLoss
- MultiLabelMarginLoss
- SmoothL1Loss
- SoftMarginLoss
- MultiLabelSoftMarginLoss
- CosineEmbeddingLoss
- MultiMarginLoss
- TripletMarginLoss

NN Block Diagram




```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(320, 10) # 320 -> 10

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = F.relu(self.mp(self.conv2(x)))
        x = x.view(in_size, -1) # flatten the tensor
        x = self.fc(x)
        return F.log_softmax(x)
```

torch.nn.Conv2d(in_channels, out_channels, kernel_size, …)
torch.nn.MaxPool2d(kernel_size, ..)

NN Block Diagram

X, Y
dataloader

↓

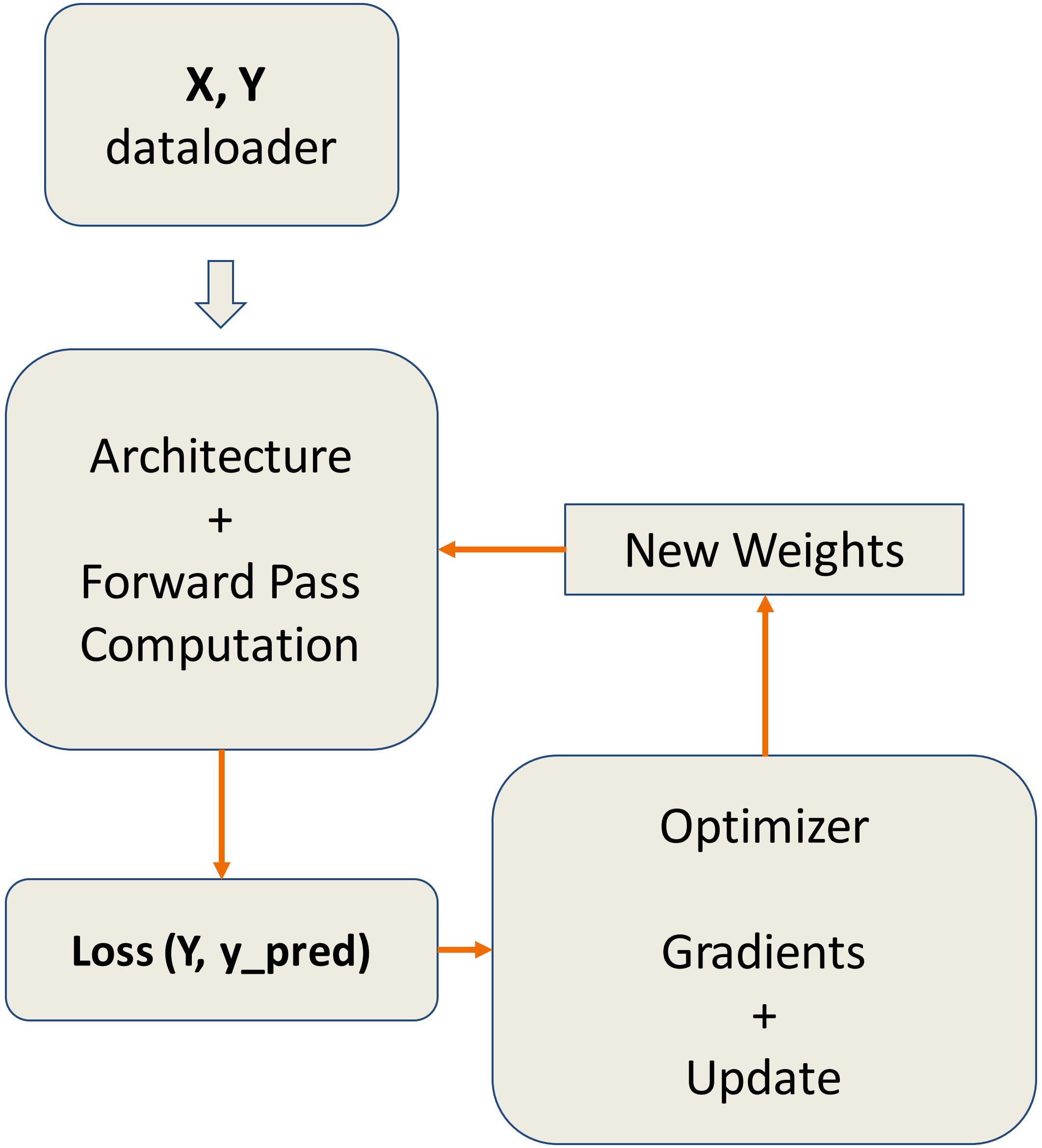
Architecture
+
Forward Pass
Computation

↓

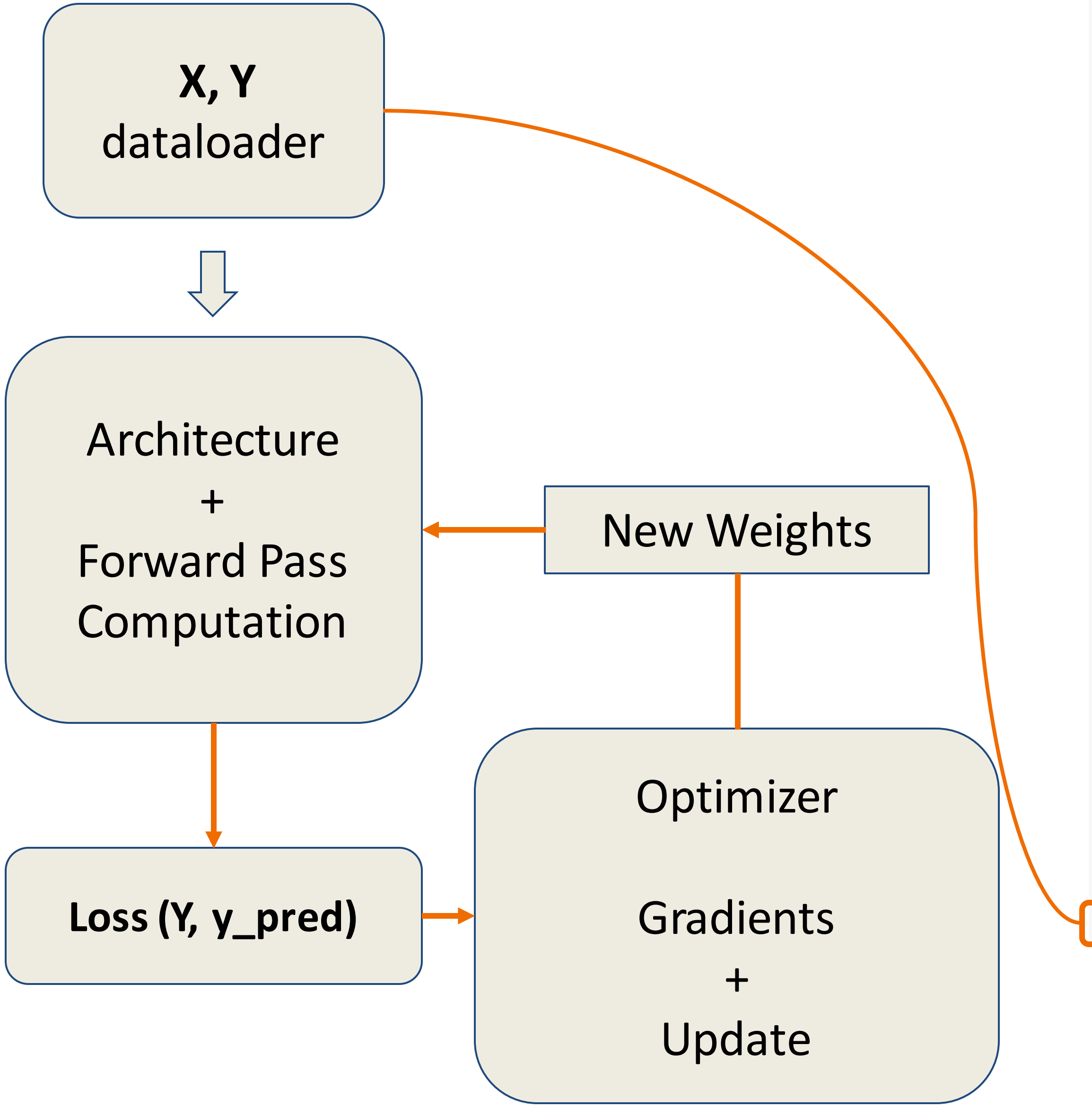
Loss (Y, y_pred)

```python
import torch
import torch.nn as nn
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)
        self.softmax = nn.Softmax(dim=1)

        #Weight Initialization
        for m in self.modules():
            if isinstance(m,nn.Linear):
                weight_init.xavier_normal_(m.weight)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.softmax(out)
        return out

N, D_in, H, D_out = 64, 1000, 100, 10
net=Net(D_in, H, D_out)
criterion=nn.CrossEntropyLoss()
```

NN Training

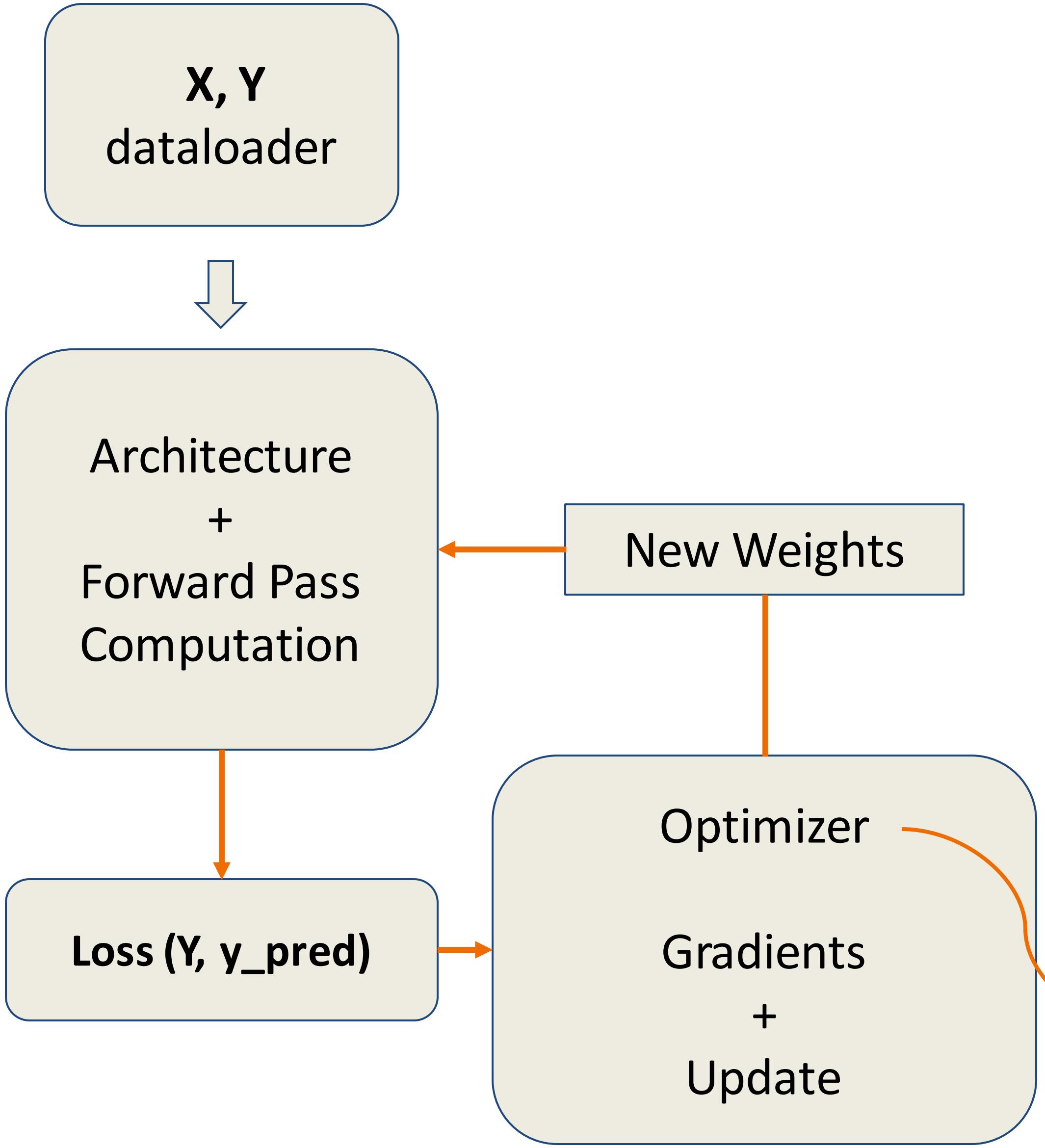

X, Y
dataloader
Architecture
+
Forward Pass
Computation
New Weights
Loss (Y, y_pred)
Optimizer
Gradients
+
Update

NN Training



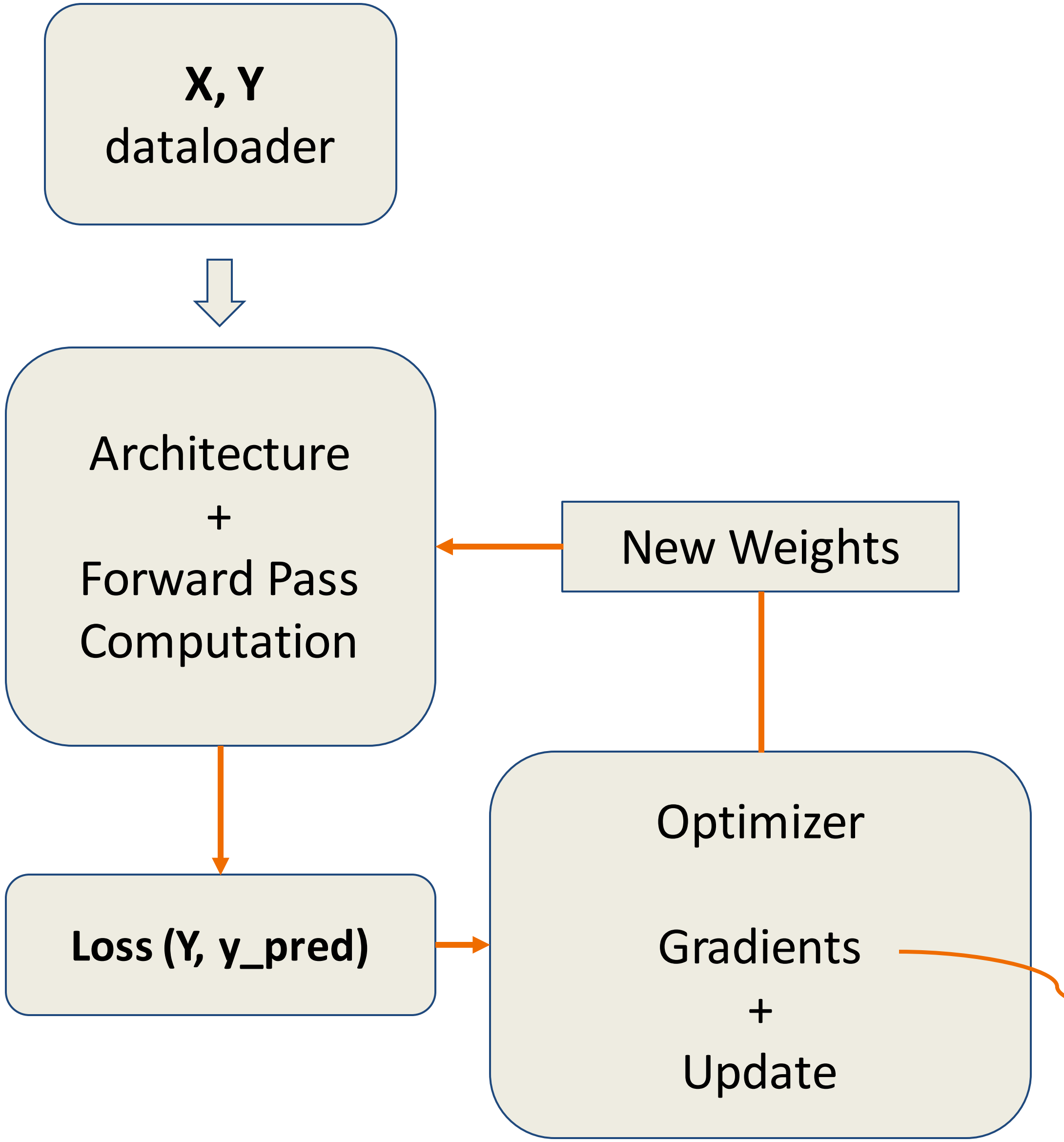

```
import torch
import torch.nn as nn
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)
        self.softmax = nn.Softmax(dim=1)

        #Weight Initialization
        for m in self.modules():
            if isinstance(m,nn.Linear):
                weight_init.xavier_normal_(m.weight)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.softmax(out)
        return out

N, D_in, H, D_out = 64, 1000, 100, 10
net=Net(D_in, H, D_out)
criterion=nn.CrossEntropyLoss()
X=torch.randn(N, D_in)
y=torch.randn(N, D_out)
train_loader=Dataloader(TensorDataset(X,y), batchsize=8)
```

NN Training



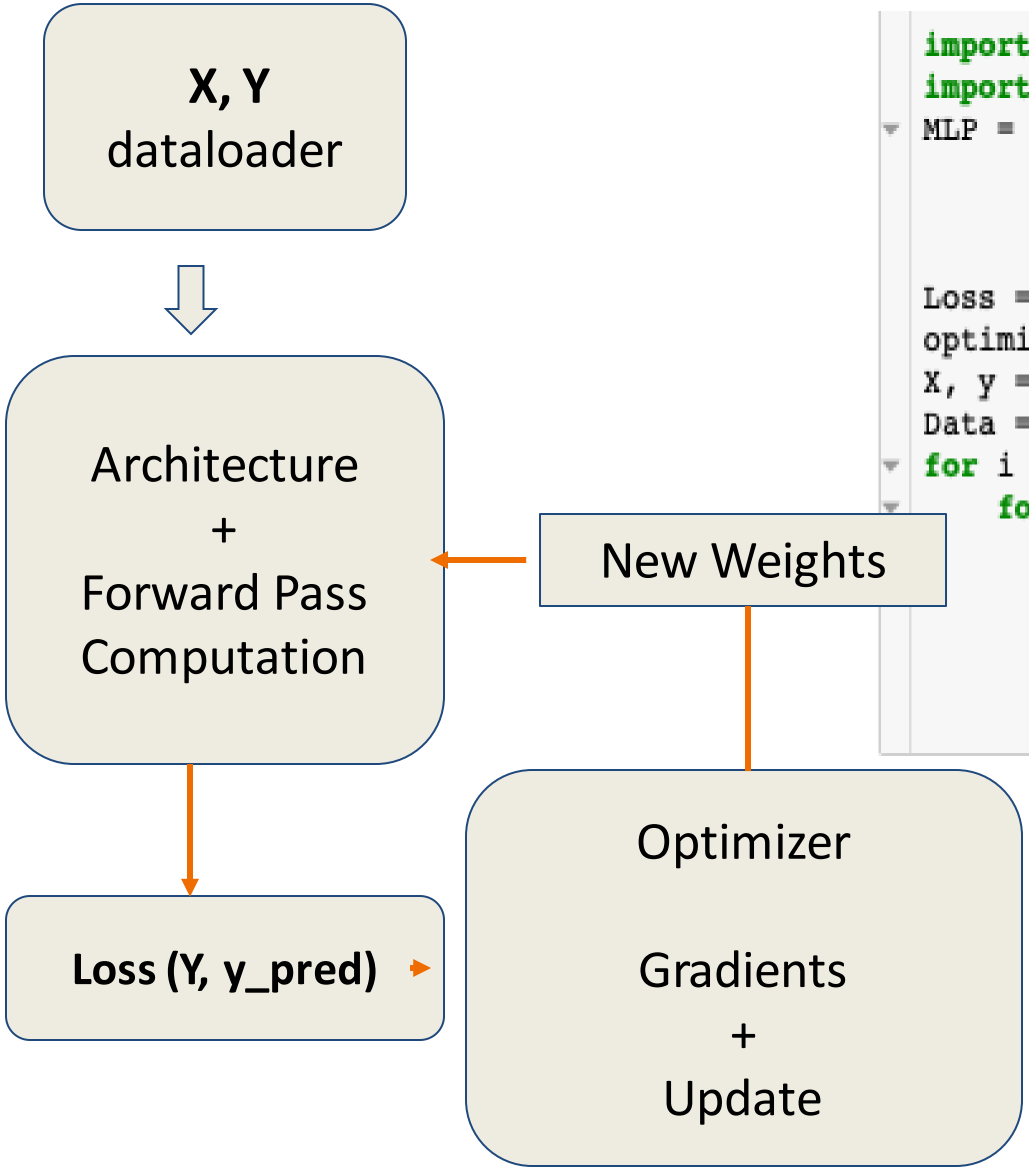

```python
use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")

import torch
import torch.nn as nn
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)
        self.softmax = nn.Softmax(dim=1)

        #Weight Initialization
        for m in self.modules():
            if isinstance(m,nn.Linear):
                weight_init.xavier_normal_(m.weight)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.softmax(out)
        return out

N, D_in, H, D_out = 64, 1000, 100, 10
net=Net(D_in, H, D_out)
criterion=nn.CrossEntropyLoss()
X=torch.randn(N, D_in)
y=torch.randn(N, D_out)
train_loader=Dataloader(TensorDataset(X,y), batchsize=8)
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
net = net.to(device)
```

NN Training




```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)
        self.softmax = nn.Softmax(dim=1)

        #Weight Initialization
        for m in self.modules():
            if isinstance(m,nn.Linear):
                weight_init.xavier_normal_(m.weight)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.softmax(out)
        return out

N, D_in, H, D_out = 64, 1000, 100, 10
net=Net(D_in, H, D_out)
criterion=nn.CrossEntropyLoss()
X=torch.randn(N, D_in)
y=torch.randn(N, D_out)
train_loader=Dataloader(TensorDataset(X,y), batchsize=8)
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
net = net.to(device)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.view(-1, 28*28)
        labels = labels
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = net(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

NN Training (alternate)




```python
import torch
import torch.nn as nn
MLP = nn.Sequential(nn.Linear(2,3)
                ,nn.ReLU()
                , nn.Linear(3,2)
                ,nn.Softmax()).double()
Loss = nn.CrossEntropyLoss()
optimiser = torch.optim.SGD(MLP.parameters(), lr = 0.01, weight_decay= 0.01)
X, y = sklearn.datasets.make_moons(200, noise=0.20)
Data = zip(torch.tensor(X), torch.tensor(y))
for i in range(10):
    for x,y in Data:
        optimiser.zero_grad()
        ypred = MLP(x)
        loss = Loss(ypred.unsqueeze(0),y.unsqueeze(0))
        loss.backward()
        optimiser.step()
```

PyTorch

- Important Packages:

Neural networks library deeply integrated with autograd designed for maximum flexibility

a Tensor library like NumPy, with strong GPU support

torch

torch. autograd

torch.nn

torch. optim

Auto differentiation library that supports all differentiable Tensor operations in torch

Optimization package with standard optimization methods such as SGD, RMSProp, LBFGS, Adam etc.

Computation on GPU

Computational Graph: GPU

```python
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)
        self.softmax = nn.Softmax(dim=1)

        #Weight Initialization
        for m in self.modules():
            if isinstance(m,nn.Linear):
                weight_init.xavier_normal_(m.weight)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.softmax(out)
        return out

N, D_in, H, D_out = 64, 1000, 100, 10
net=Net(D_in, H, D_out)
criterion=nn.CrossEntropyLoss()
X=torch.randn(N, D_in)
y=torch.randn(N, D_out)
train_loader=Dataloader(TensorDataset(X,y), batchsize=8)
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
net = net.to(device)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.view(-1, 28*28)
        labels = labels
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = net(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

Computational Graph: Pytorch

Computational Graph with Pytorch on GPU

```
import torch
```

```
use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")
```

```
x=torch.randn(3,4, requires_grad=True).to(device)
y=torch.randn(3,4, requires_grad=True).to(device)
z=torch.randn(3,4, requires_grad=True).to(device)
```

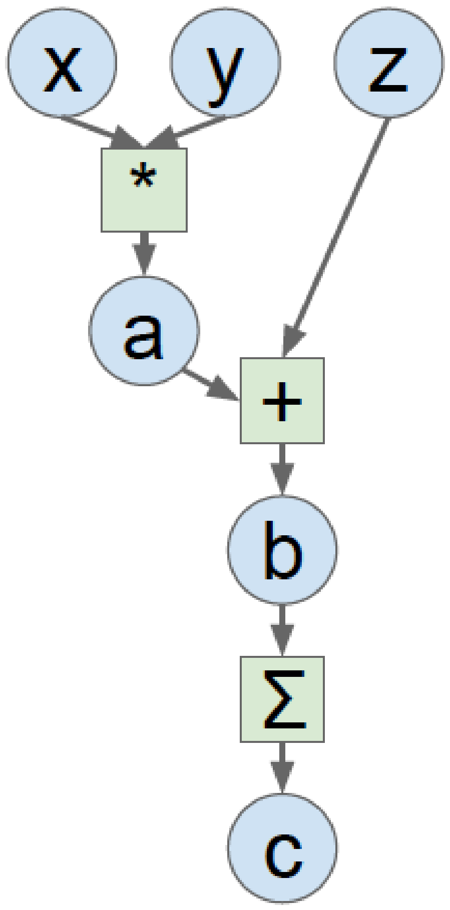
```
a = x*y
b = a+z
c = torch.sum(b)
```

```
c.backward()
```

```
print(x.grad.data, '\n', y.grad.data, z.grad.data)
```

```
tensor([[ 2.0230,  1.5964,  0.2990,  1.0220],
        [-0.3020, -1.5197, -1.1080, -0.3008],
        [-1.9656, -0.7705, -1.2423, -0.0137]]) tensor([[-1.8837, -1.5356, -0.4437,  1.4743],
        [-0.7601, -1.1921, -0.6407,  0.6169],
        [ 1.5937, -0.0240,  0.5942, -0.7387]]) tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]])
```

Computation happens on GPU, if available



CPU vs GPU

CPU vs GPU

	# Cores	Clock Speed	Memory	Price
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.4 GHz	Shared with system	$339
CPU (Intel Core i7-6950X)	10 (20 threads with hyperthreading)	3.5 GHz	Shared with system	$1723
GPU (NVIDIA Titan Xp)	3840	1.6 GHz	12 GB GDDR5X	$1200
GPU (NVIDIA GTX 1070)	1920	1.68 GHz	8 GB GDDR5	$399

CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

GPU: More cores, but each core is much slower and "dumber"; great for parallel tasks

Thanks!!

Questions?