DEEP LEARNING FOR RESEARCH

JANUARY 17TH

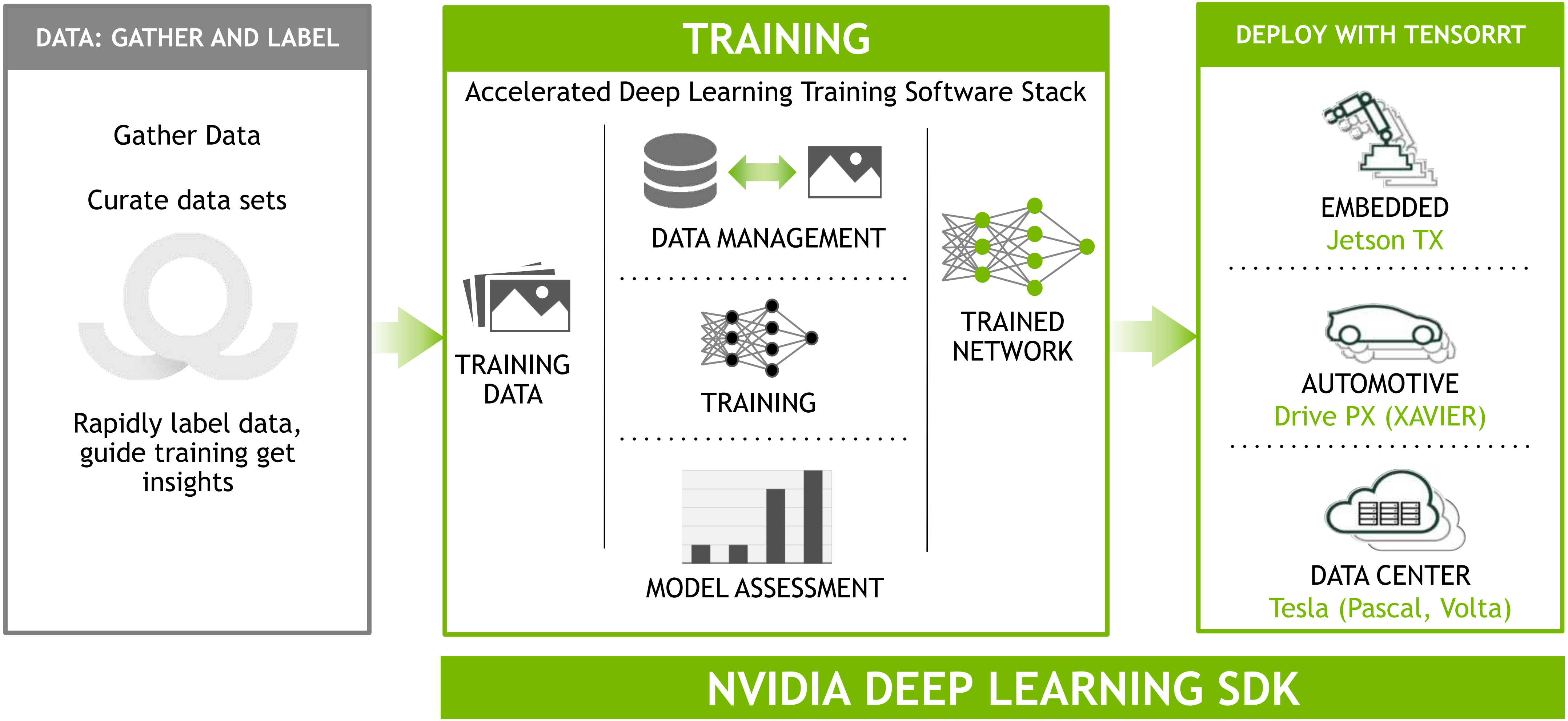
Ashish Sardana | Deep Learning Solutions Architect

Deep Learning with PyTorch

AGENDA

- What is PyTorch
- Why PyTorch
- NN in PyTorch’s way

DEEP LEARNING WORKFLOW

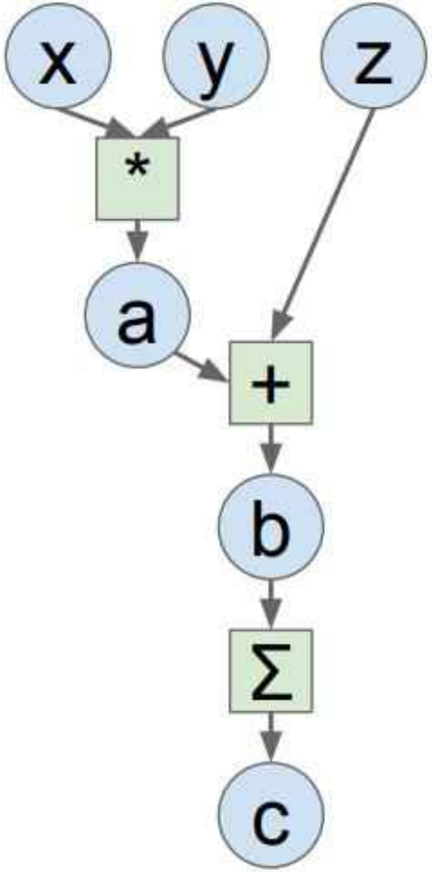


WHERE DOES PYTORCH COMES INTO PLAY?

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
```

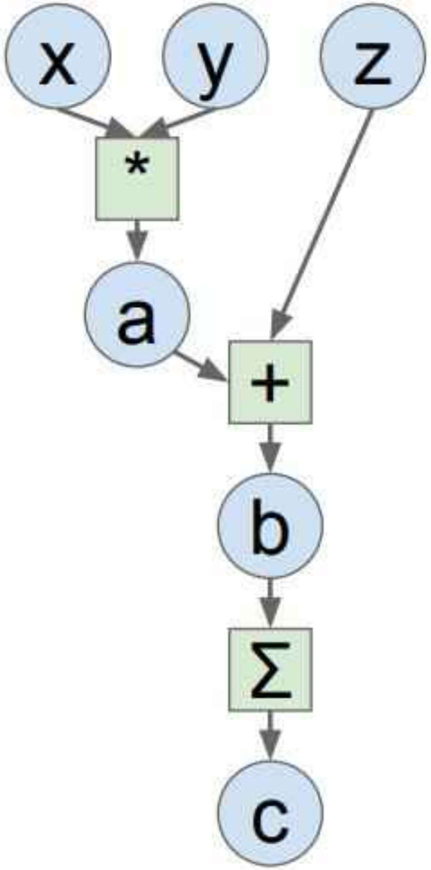


Problems:
- Can't run on GPU
- Have to compute our own gradients

Computational Graphs

Numpy

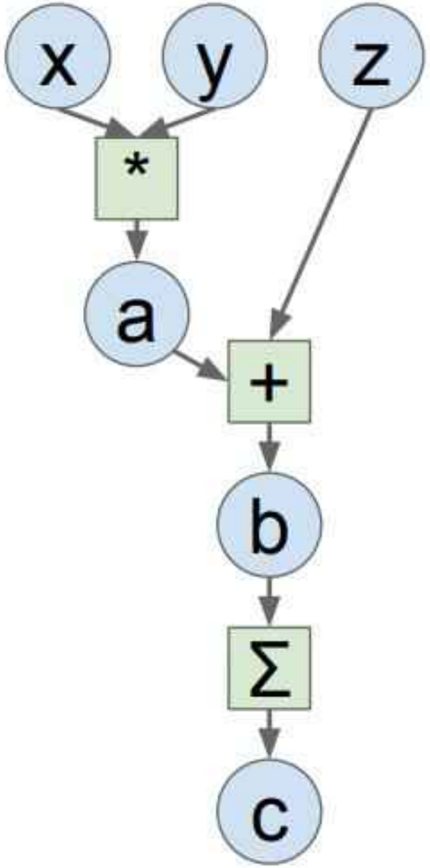
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

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

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



TensorFlow

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Computational Graphs

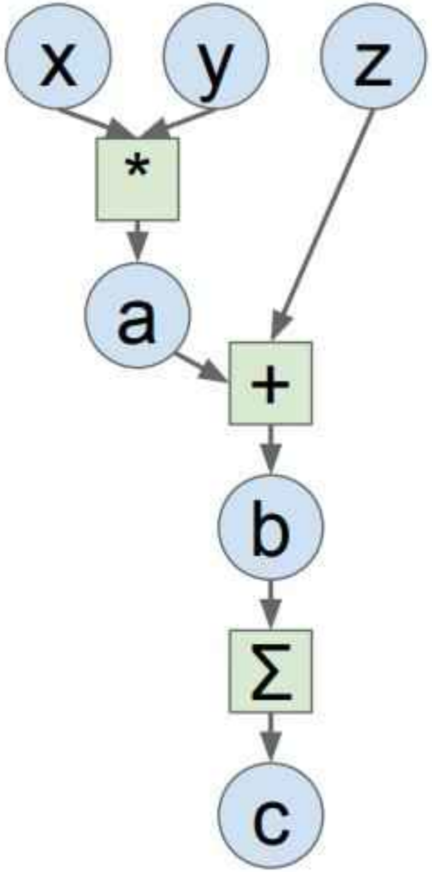


Ask TensorFlow to compute gradients

TensorFlow

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Computational Graphs

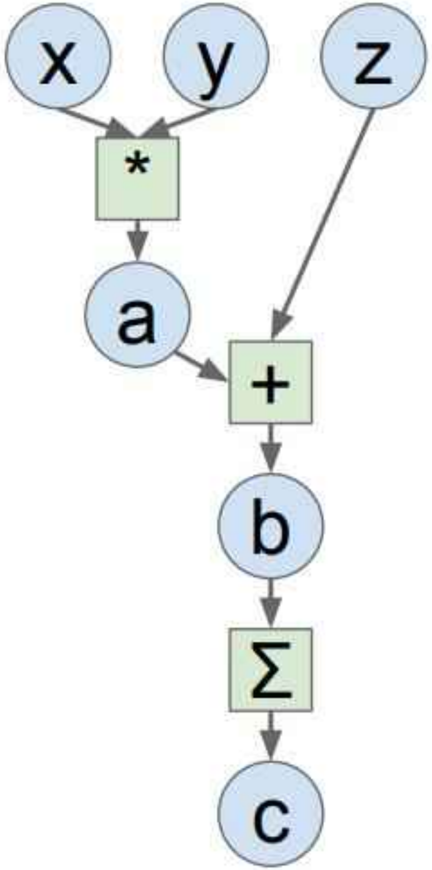


Define **Variables** to start building a computational graph

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)
```

Computational Graphs

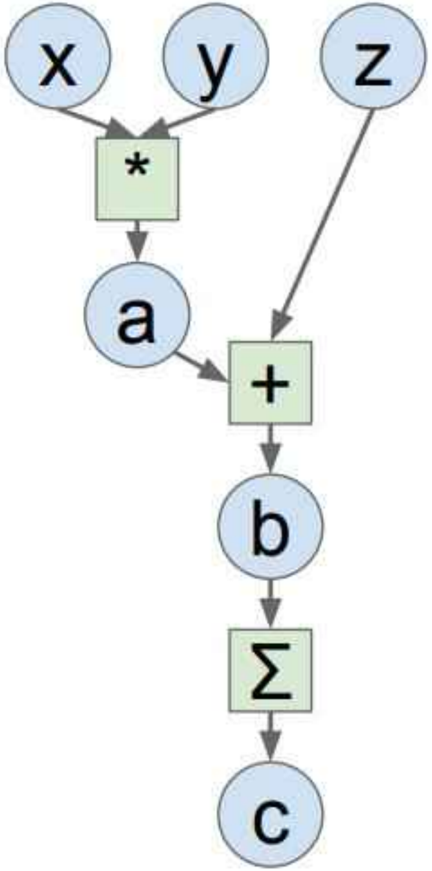


Forward pass looks just like numpy

PyTorch

```python
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)
```

Computational Graphs



Run on GPU by casting to .cuda()

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

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

Numpy

```python
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
```

TensorFlow

```python
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```python
import torch
from torch.autograd import Variable

N, D = 3, 4

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

WHAT IS PYTORCH?

automatic differentiation engine

Ndarray library with GPU support

gradient based optimization package

Utilities (data loading, etc.)

Deep Learning

Reinforcement Learning

Numpy-alternative

BASIC CONCEPTS

Key Terms

- **torch.Tensor** – similar to numpy.array, with GPU
- **autograd.Variable** – wraps a tensor and enables auto differentitation
- **autograd.Function** – operate on Variables. Implements forward and backward
- **nn.Parameter** – a special Variable
- **nn.Module** – contain Parameters and define functions on input Variables

WRITE NEW MODELS

Using nn.sequential()

```python
from torch import nn

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()

        self.block = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=4, stride=4))
        self.fc = nn.Linear(64*16, 10)

    def forward(self, x):
        x = self.block(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

Group into

nn.Sequential

WRITE NEW MODELS

Using nn.functional

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

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.bn = BatchNorm2d(16)
        self.fc = nn.Linear(64*16, 10)

    def forward(self, x):
        x = self.conv(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 4, stride=4)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

Functional interface

Not recommended, but sometimes useful

NEURAL NETWORK

4-Layered CNN

```
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.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)

model = Net()
input = Variable(torch.randn(10, 20))
output = model(input)
```

OPTIMIZATION PACKAGE

SGD, AdaGrad, RMSProp, LBFGS, etc.

```
net = Net()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01, momentum=0.9)

for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = F.cross_entropy(output, target)
    loss.backward()
    optimizer.step()
```

DISTRIBUTED PYTORCH

Across Nodes

- MPI Style Distribution communication
- Broadcast tensors to other nodes
- Reduce tensors among nodes
    - Eg – sum gradients among all nodes

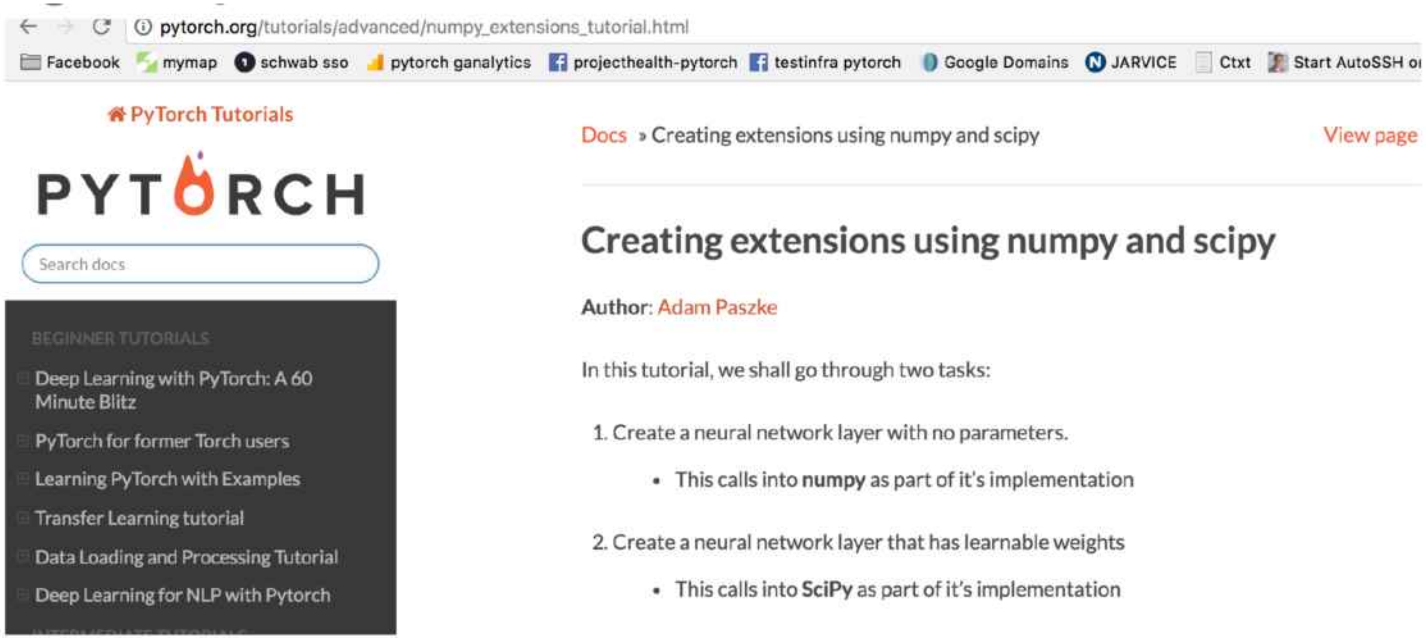
DISTRIBUTED DATA PARALLEL

```
for epoch in range(max_epochs):
    for data, target in enumerate(training_data):
        output = model(data)
        model = nn.DistributedDataParallel(model)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

ECOSYSTEM

It's not just PyTorch but it's surroundings

- Use the entire Python ecosystem at your will
- Inculuding scikit-learn, scipy, matplotlib etc.



MIXED PRECISION COMPUTATION

MISPERCEPTIONS

- “Mixed Precision is less acurate!“
- “My code benefits from Mixed Precision without doing anything!“
  - Likely wrong in 95% of cases. Most code that‘s out on github is initialized with fp32.
  - A little bit has to be done.
- Mixed Precision is all about speed
  - It is about **larger input size** and **larger minbatches**
  - It is **doubling** memory throughput

MIXED PRECISION AND ACCURACY

DNN Model	FP32	Mixed Precision
AlexNet	56.77%	56.93%
VGG-D	65.40%	65.43%
GoogLeNet	68.33%	68.43%
Inception v1	70.03%	70.02%
Resnet50	73.61%	73.75%

Table 1. Top-1 accuracy on ILSVRC12 validation data.

THANK YOU!

~QUESTIONS?