



DEEP LEARNING FOR RESEARCH

JANUARY 17TH

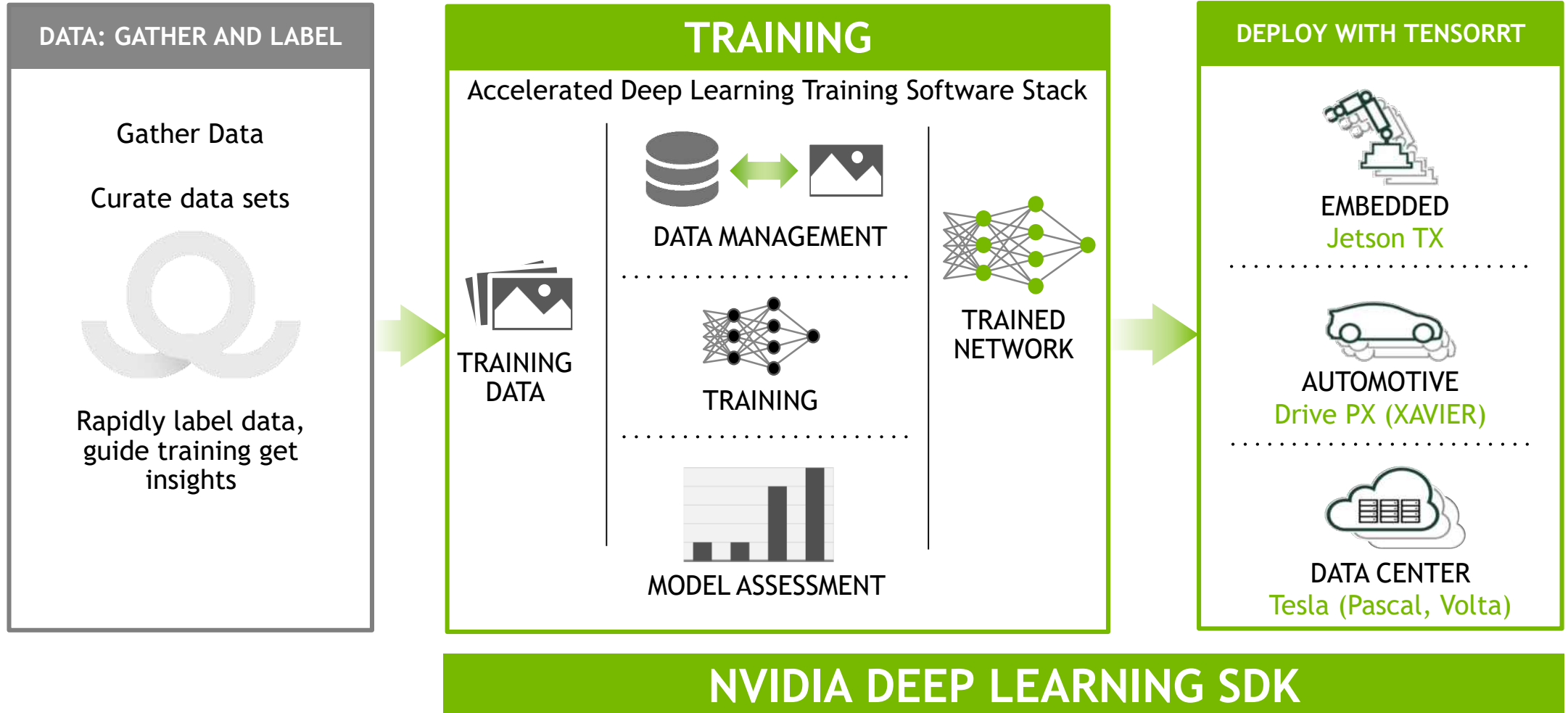
Ashish Sardana | Deep Learning Solutions Architect

AGENDA

Deep Learning with PyTorch

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

DEEP LEARNING WORKFLOW





WHERE DOES PYTORCH COMES INTO PLAY?

Computational Graphs

Numpy

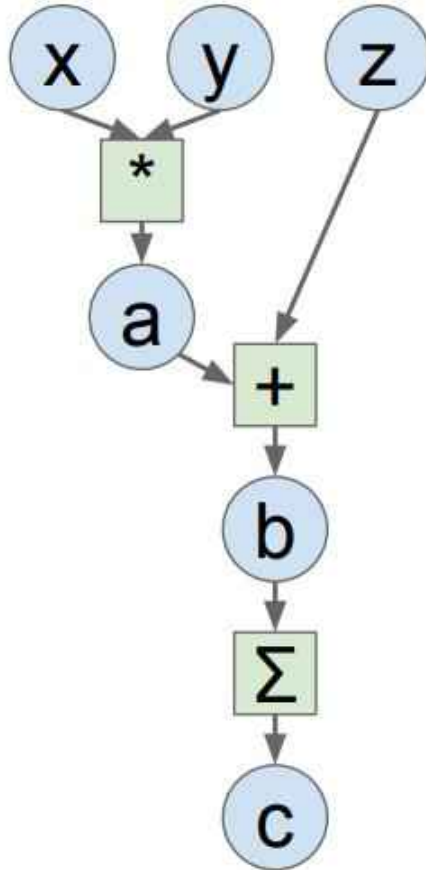
```
import numpy as np
np.random.seed(0)

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

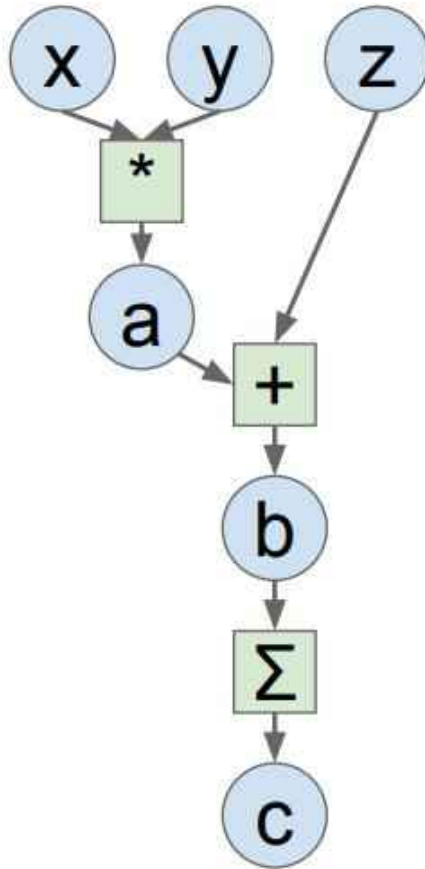
```
import numpy as np
np.random.seed(0)

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

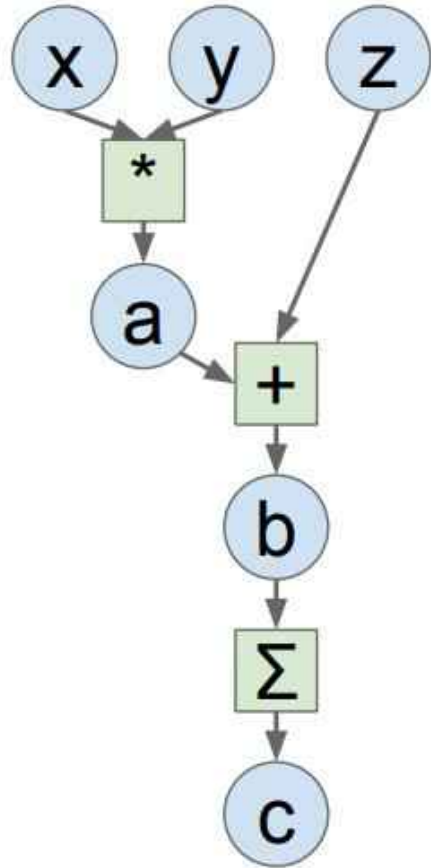
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

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

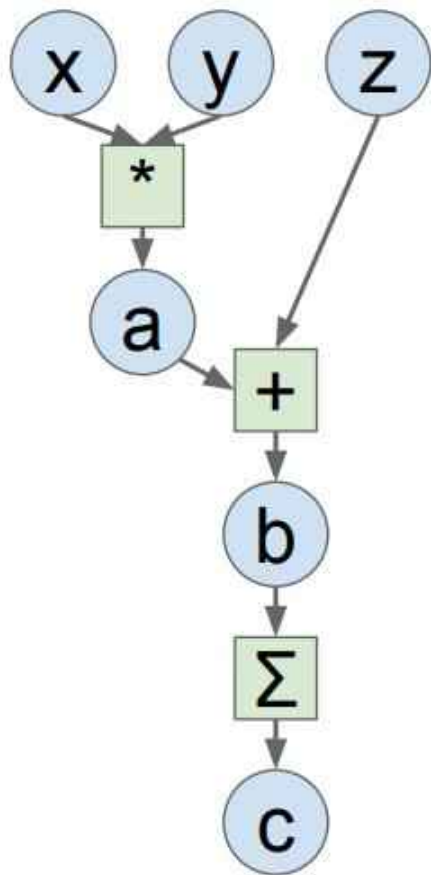
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

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
```

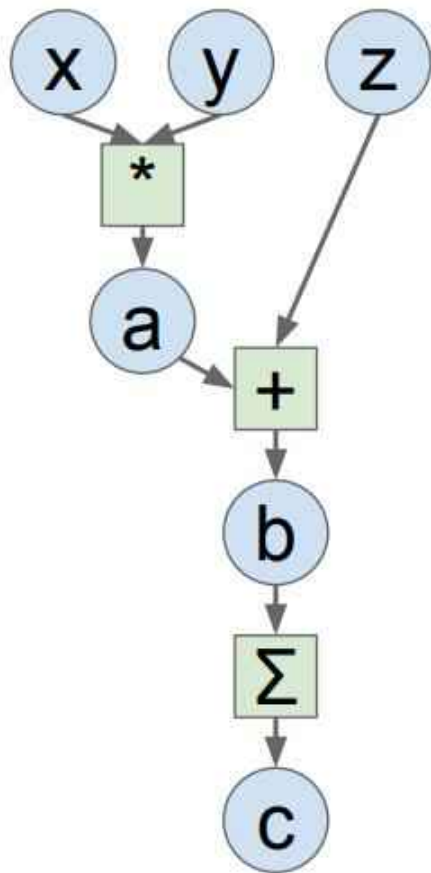
```
x = Variable(torch.randn(N, D),
              requires_grad=True)
y = Variable(torch.randn(N, D),
              requires_grad=True)
z = Variable(torch.randn(N, D),
              requires_grad=True)
```

```
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

```
import torch
from torch.autograd import Variable

N, D = 3, 4

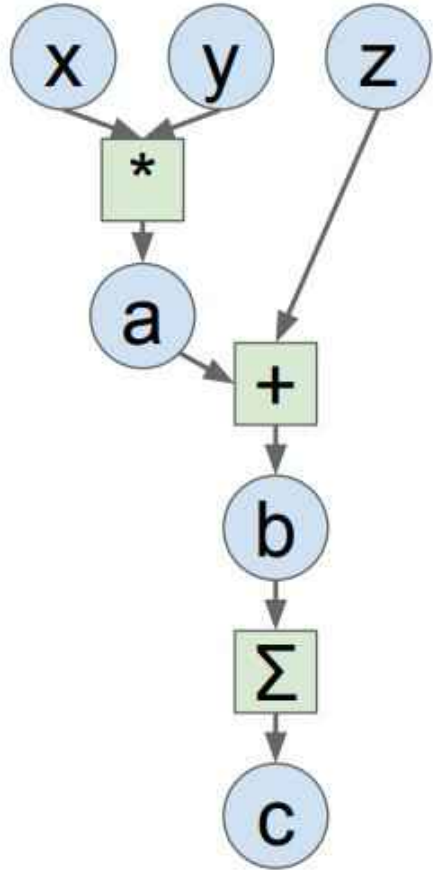
x = Variable(torch.randn(N, D),
              requires_grad=True)
y = Variable(torch.randn(N, D),
              requires_grad=True)
z = Variable(torch.randn(N, D),
              requires_grad=True)

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

```
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

```
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

```
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 differentiation
- **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()`

```
1 from torch import nn
2
3 class SimpleCNN(nn.Module):
4     def __init__(self):
5         super(SimpleCNN, self).__init__()
6
7         self.block = nn.Sequential(
8             nn.Conv2d(3, 16, kernel_size=3, padding=1),
9             nn.BatchNorm2d(16),
10            nn.ReLU(),
11            nn.MaxPool2d(kernel_size=4, stride=4))
12        self.fc = nn.Linear(64*16, 10)
13
14    def forward(self, x):
15        x = self.block(x)
16        x = x.view(x.size(0), -1)
17        x = self.fc(x)
18        return x
```

Group into
`nn.Sequential`

WRITE NEW MODELS

Using nn.functional

```
1 from torch import nn
2 import torch.nn.functional as F
3
4 class SimpleCNN(nn.Module):
5     def __init__(self):
6         super(SimpleCNN, self).__init__()
7         self.conv = nn.Conv2d(3, 16, kernel_size=3, padding=1)
8         self.bn = BatchNorm2d(16)
9         self.fc = nn.Linear(64*16, 10)
10
11     def forward(self, x):
12         x = self.conv(x)
13         x = F.relu(x)
14         x = F.max_pool2d(x, 4, stride=4)
15         x = x.view(x.size(0), -1)
16         x = self.fc(x)
17         return x
```

Functional
interface

Not recommended, but
sometimes useful

NEURAL NETWORK

4-Layered CNN

```
1  class Net(nn.Module):
2      def __init__(self):
3          super(Net, self).__init__()
4          self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
5          self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
6          self.conv2_drop = nn.Dropout2d()
7          self.fc1 = nn.Linear(320, 50)
8          self.fc2 = nn.Linear(50, 10)
9
10     def forward(self, x):
11         x = F.relu(F.max_pool2d(self.conv1(x), 2))
12         x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
13         x = x.view(-1, 320)
14         x = F.relu(self.fc1(x))
15         x = F.dropout(x, training=self.training)
16         x = self.fc2(x)
17         return F.log_softmax(x)
18
19  model = Net()
20  input = Variable(torch.randn(1, 1, 1, 1))
21  output = model(input)
```


OPTIMIZATION PACKAGE

SGD, AdaGrad, RMSProp, LBFGS, etc.

```
1 net = Net()
2 optimizer = torch.optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
3
4 for input, target in dataset:
5     optimizer.zero_grad()
6     output = model(input)
7     loss = F.cross_entropy(output, target)
8     loss.backward()
9     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

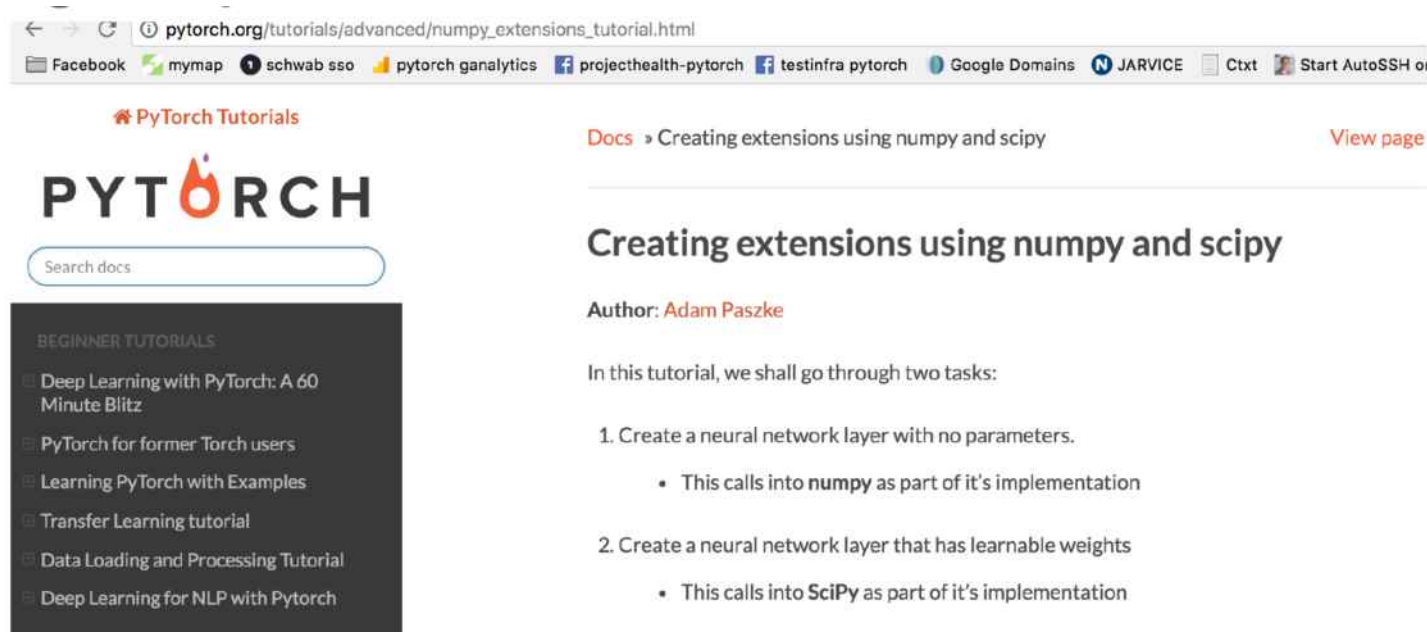
DISTRIBUTED DATA PARALLEL

```
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
- Including scikit-learn, scipy, matplotlib etc.



The screenshot shows a web browser window with the URL `pytorch.org/tutorials/advanced/numpy_extensions_tutorial.html`. The browser's address bar and tabs are visible at the top. The main content area features the PyTorch logo and a search bar. On the left, a sidebar lists 'BEGINNER TUTORIALS' including 'Deep Learning with PyTorch: A 60 Minute Blitz', 'PyTorch for former Torch users', 'Learning PyTorch with Examples', 'Transfer Learning tutorial', 'Data Loading and Processing Tutorial', and 'Deep Learning for NLP with PyTorch'. The main content area displays the title 'Creating extensions using numpy and scipy' by Adam Paszke. The text states: 'In this tutorial, we shall go through two tasks: 1. Create a neural network layer with no parameters. • This calls into **numpy** as part of it's implementation 2. Create a neural network layer that has learnable weights • This calls into **SciPy** as part of it's implementation'.

MIXED PRECISION COMPUTATION

The background of the slide features a smooth gradient from a deep green on the left to a bright yellow on the right. Overlaid on this gradient is a complex, abstract network of small white dots connected by thin white lines, resembling a molecular structure or a data network. The density of these dots and lines increases towards the right side of the image.

MISPERCEPTIONS

- “Mixed Precision is less accurate!”
- “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.

The background is a dark space filled with a complex network of glowing green and yellow lines and dots. The lines connect various points, creating a web-like structure. A prominent, dense cluster of yellow lines and dots is located on the right side, while the left side features more sparse green connections. The overall effect is one of a dynamic, interconnected system.

THANK YOU!

~QUESTIONS?