A Brief Introduction to PyTorch

(a deep learning library)

Wanxiang Che

pytorch.org

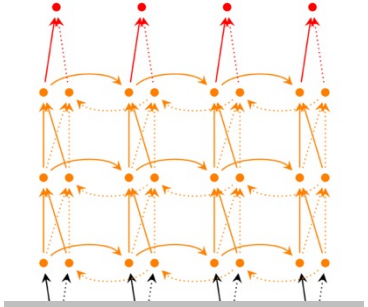
Outline

- Why deep learning libraries
- A brief introduction to PyTorch
- PyTorch in action

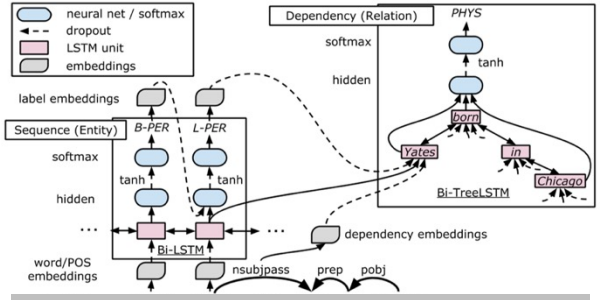
Outline

- Why deep learning libraries
- A brief introduction to PyTorch
- PyTorch in action

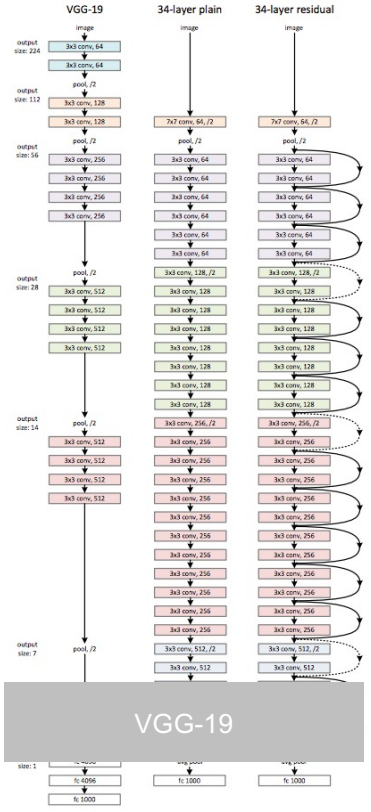
Challenges for Modern Neural Network
deeper, larger, and complex



Stacked LSTM

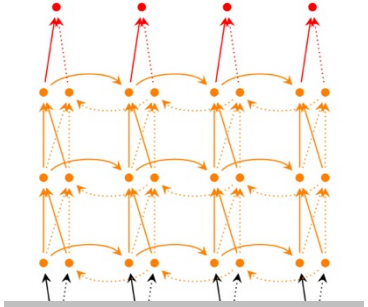


Research-driven Neural Network

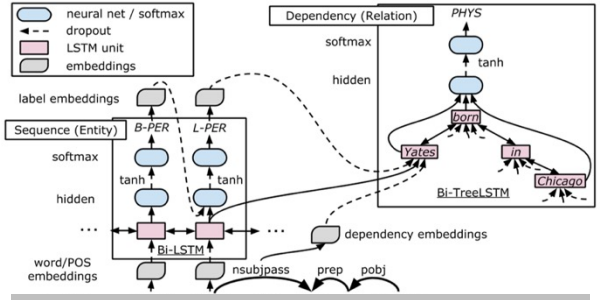


VGG-19

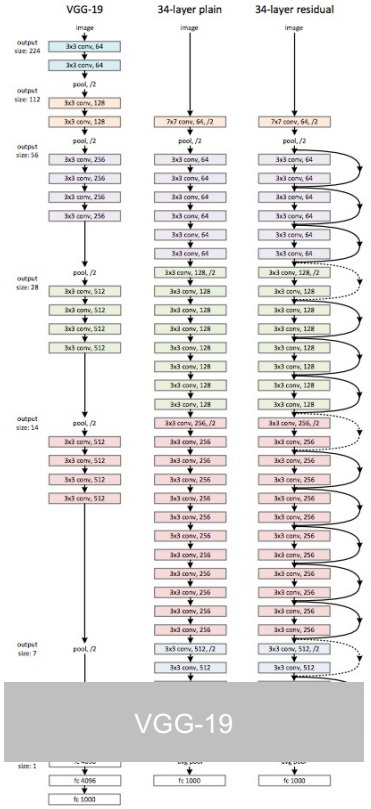
They share basic building blocks
computation graph & backpropagation



Stacked LSTM



Research-driven Neural Network

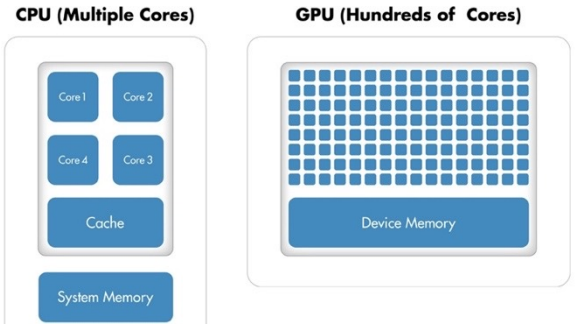


VGG-19

Deep learning from the perspective of SE
abstraction and reuse

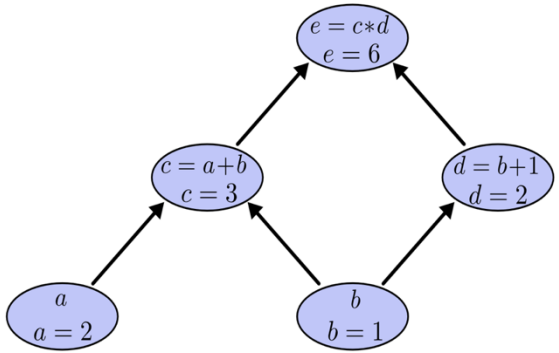
Tensor

Imperative ndarray



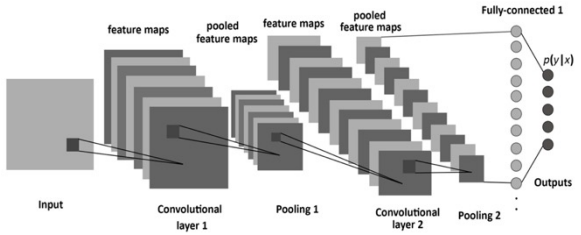
Expression

Node in a computational graph (data, grad)

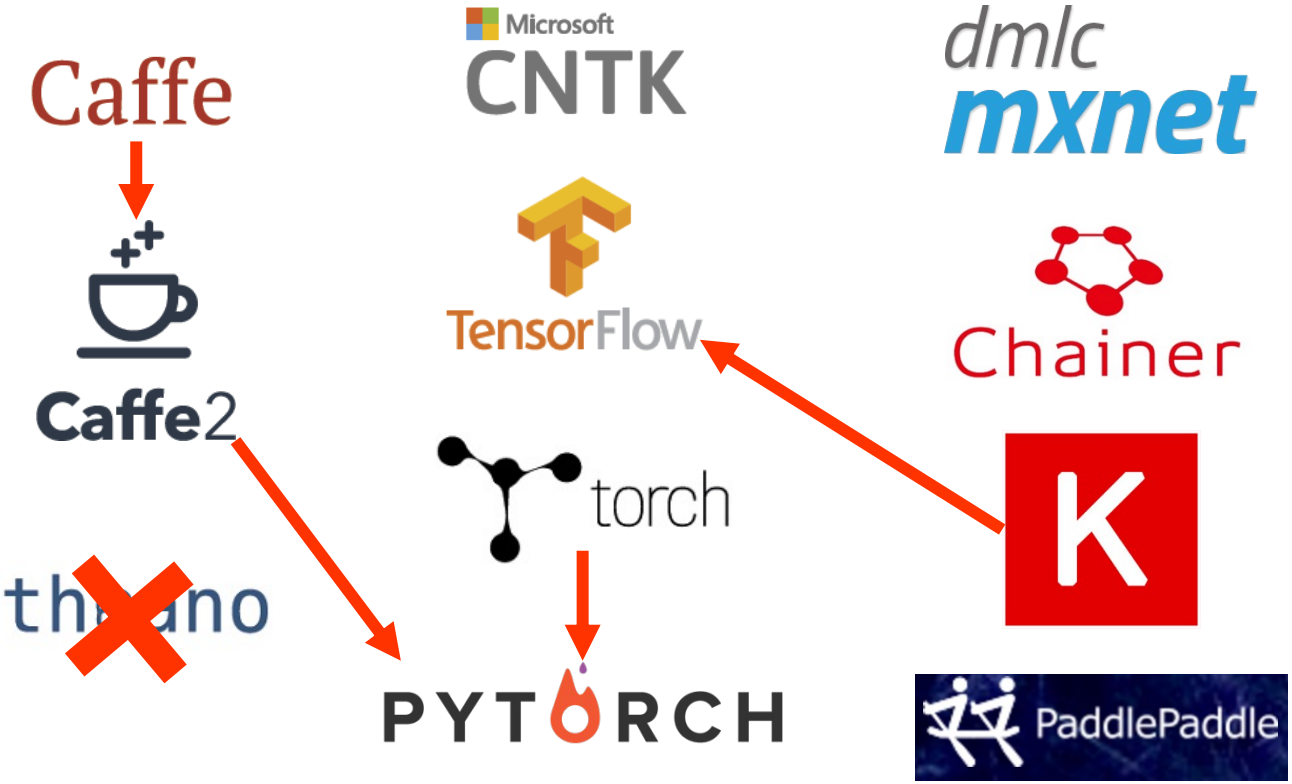


Module

A neural network layer



Popular Deep Learning Libraries

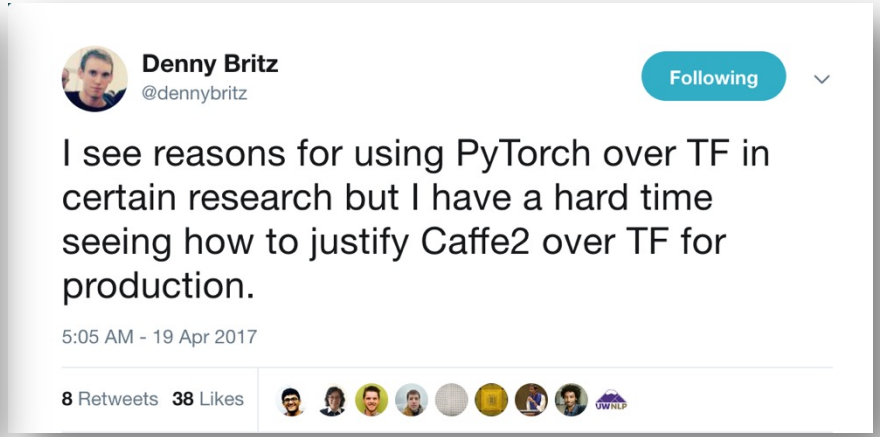


Why PYTORCH

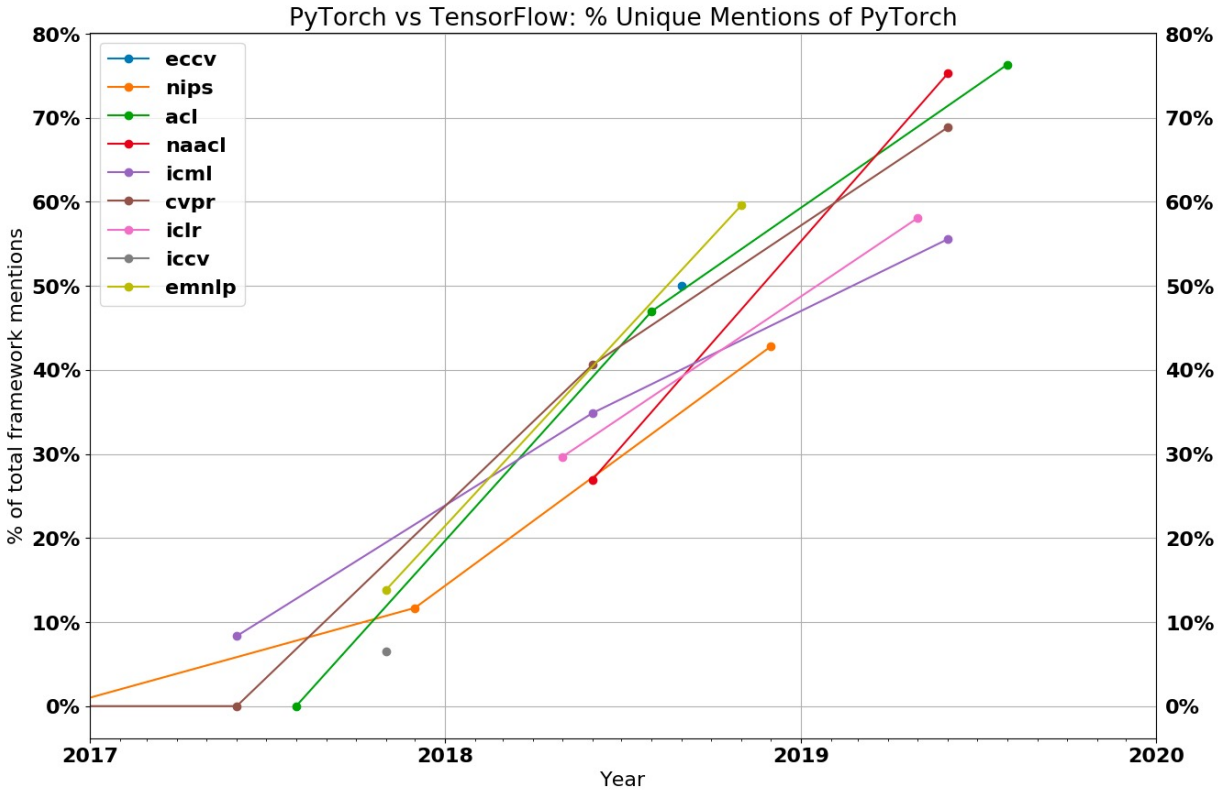
	Pro.	Con.
	• Static computation graph • Deployment features (e.g. quantization)	• Static computation graph • Over software-engineered • Very complex abstraction
PYTORCH	• Dynamic c.g., eager execution • Understandable API	• Inefficient memory • Lack of deployment features

Why PYTORCH


Andrej Karpathy
@karpathy
Following
I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eye sight has improved.
2:56 AM - 27 May 2017
402 Retweets 1,540 Likes


Denny Britz
@dennybritz
Following
I see reasons for using PyTorch over TF in certain research but I have a hard time seeing how to justify Caffe2 over TF for production.
5:05 AM - 19 Apr 2017
8 Retweets 38 Likes

PyTorch vs. TensorFlow in Research

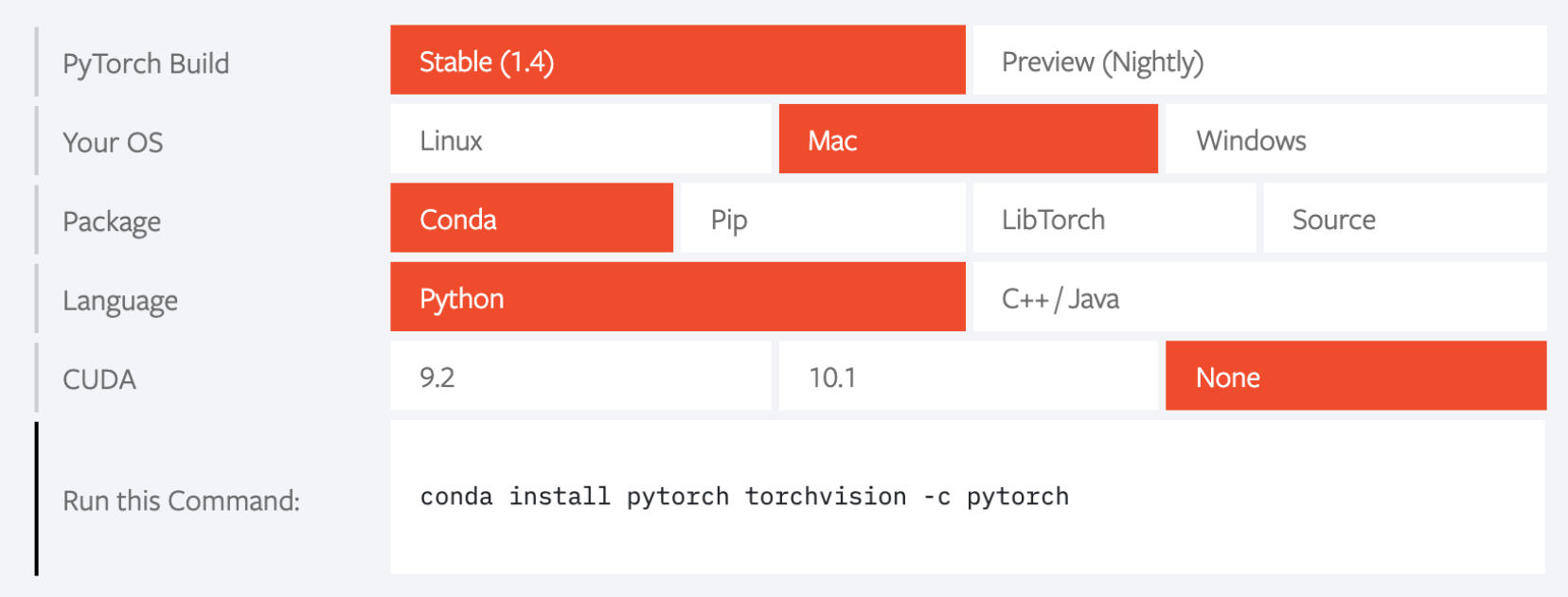


Outline

- Why deep learning libraries
- A brief introduction to PyTorch
- PyTorch in action

Personal suggestion for installation with **conda**

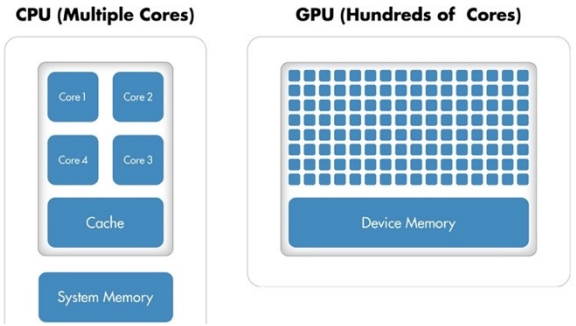
https://pytorch.org/



Deep learning from the perspective of SE
abstraction and reuse

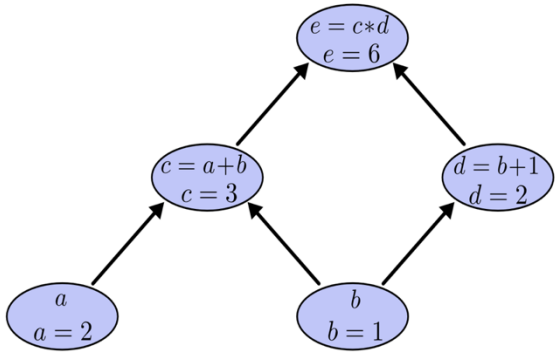
Tensor

Imperative ndarray



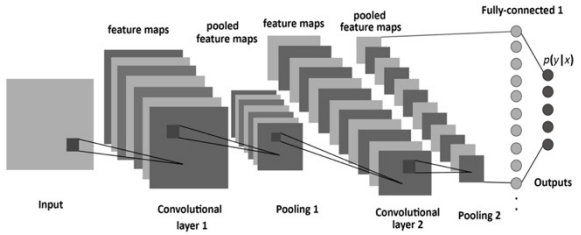
Expression

Node in a computational graph (data, grad)



Module

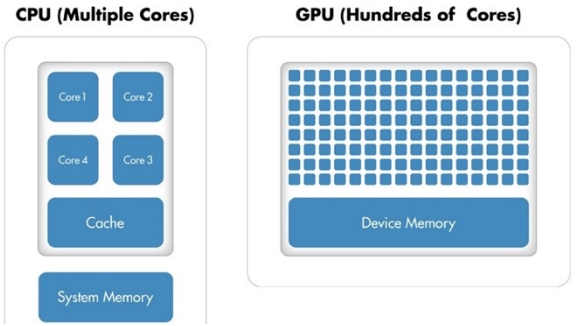
A neural network layer



Deep learning from the perspective of SE
abstraction and reuse

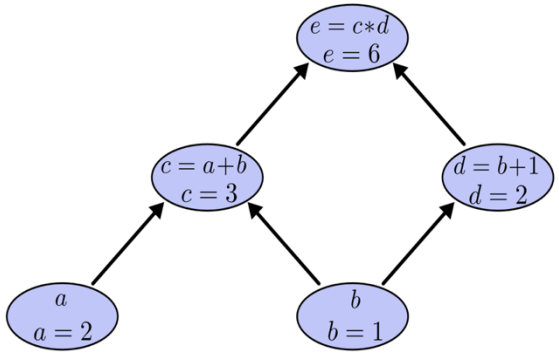
Tensor

Imperative ndarray



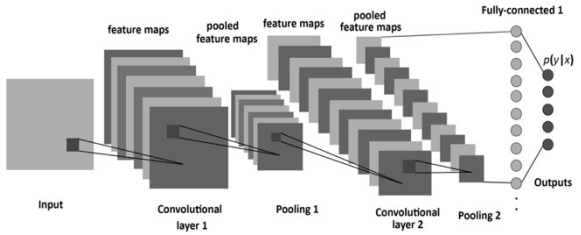
Expression

Node in a computational graph (data, grad)



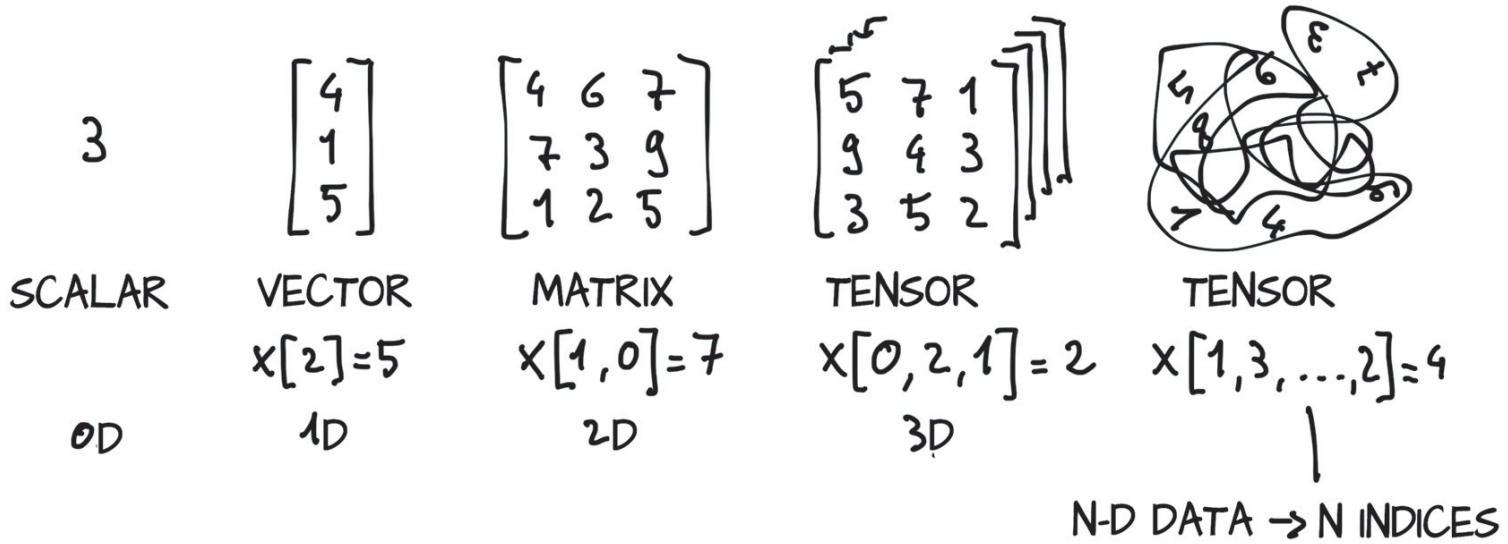
Module

A neural network layer



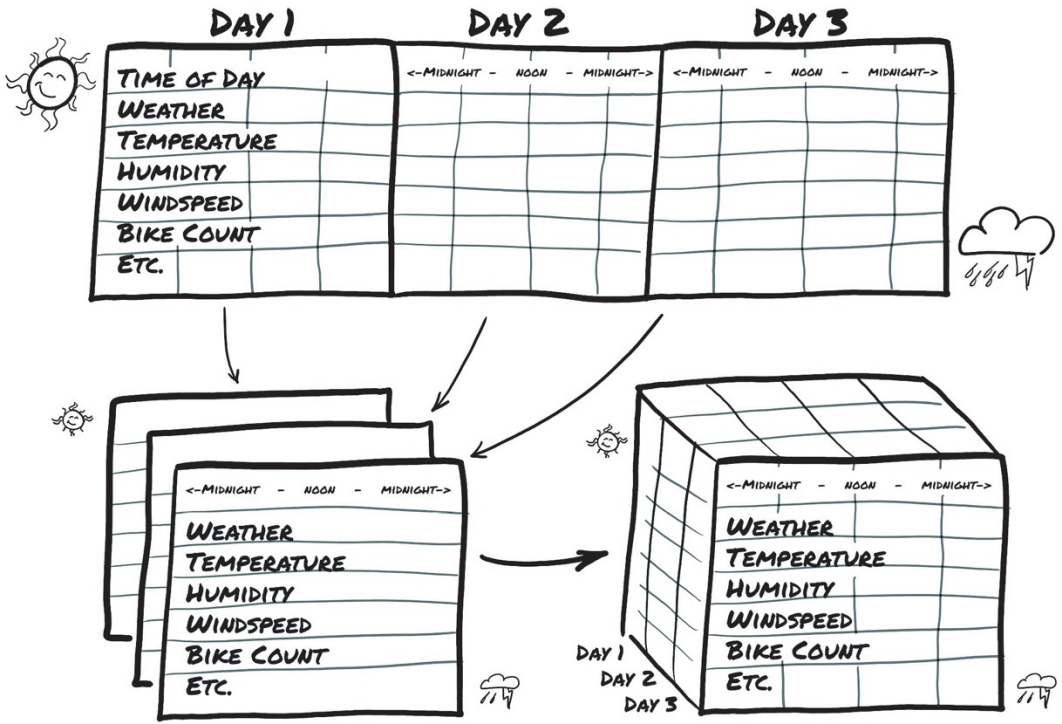
Tensor

- Tensors are multidimensional arrays



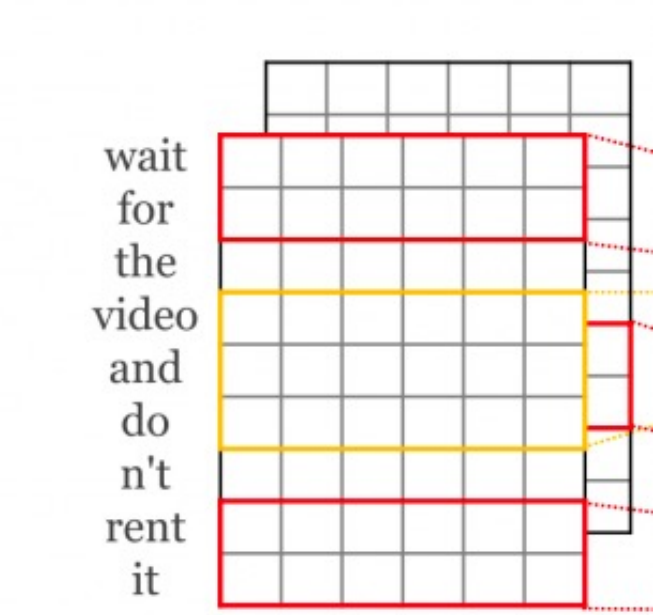
Real-world data representation with tensors

- Tabular data
- Time series data



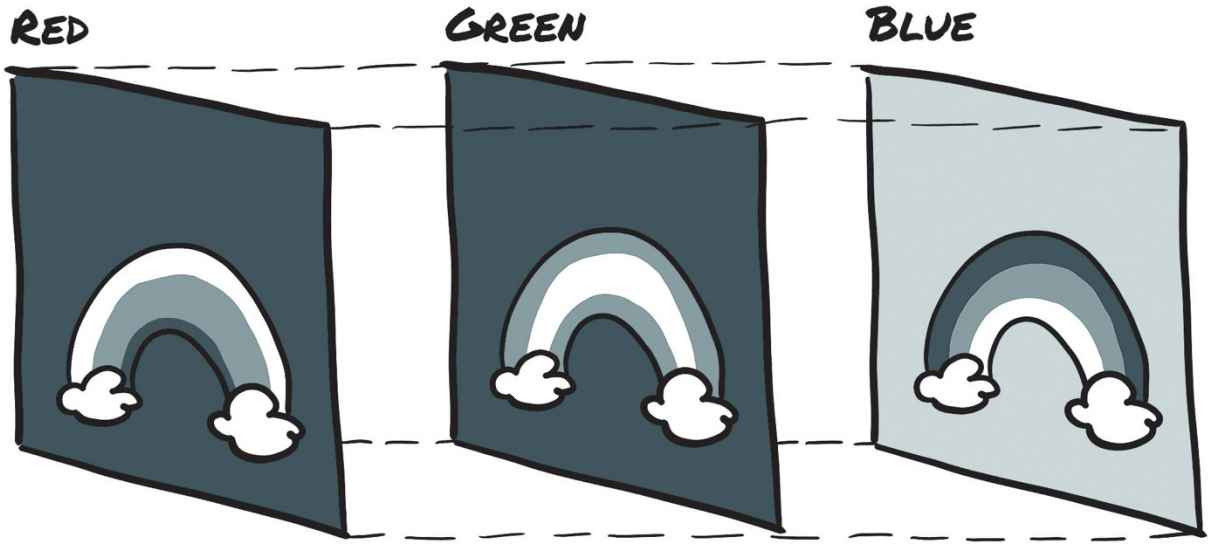
Real-world data representation with tensors

- Tabular data
- Time series data
- Text



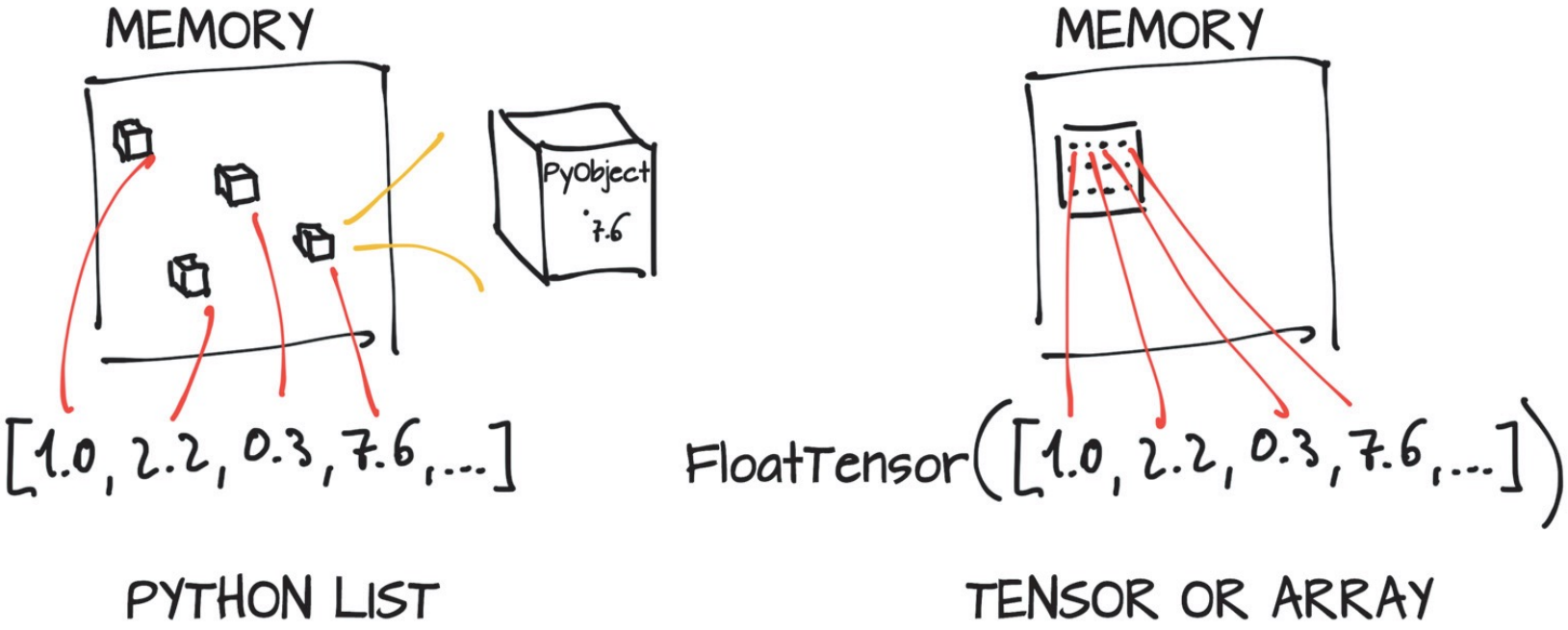
Real-world data representation with tensors

- Tabular data
- Time series data
- Text
- Images




```
batch = torch.zeros(100, 3, 256, 256, dtype=torch.uint8)
```

Python List vs. Tensor



torch
PyTorch as A Tensor Library

- Tensor operations: slicing, indexing, math operations, linear algebra, reductions, mask
  - CPU & GPU
  - Fast! (comparison on speed of **matrix multiplication**)

$$M * M * M \quad M \in \mathbb{R}^{1000\times1000}$$

Numpy

```
In [2]: M = numpy.random.randn(1000,1000)

In [3]: timeit -n 500 M.dot(M).dot(M)
500 loops, best of 3: 30.7 ms per loop
```

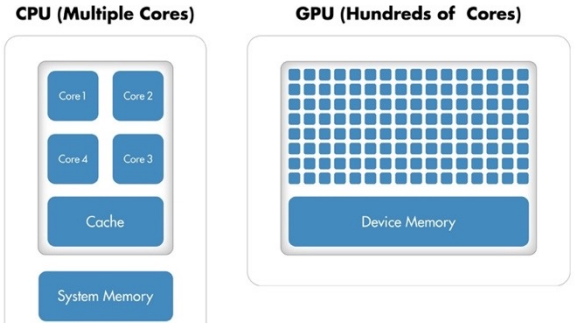
PyTorch

```
In [4]: N = torch.randn(1000,1000).cuda()

In [5]: timeit -n 500 N.mm(N).mm(N)
500 loops, best of 3: 474 µs per loop
```

Deep learning from the perspective of SE
abstraction and reuse

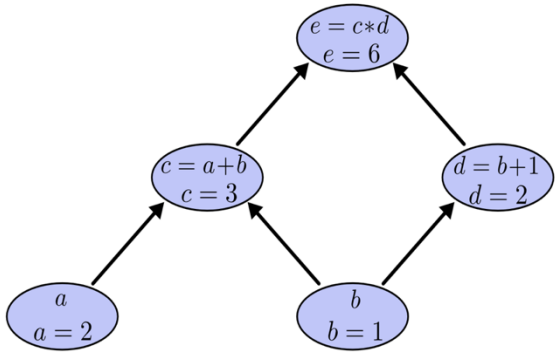
Tensor

Imperative ndarray



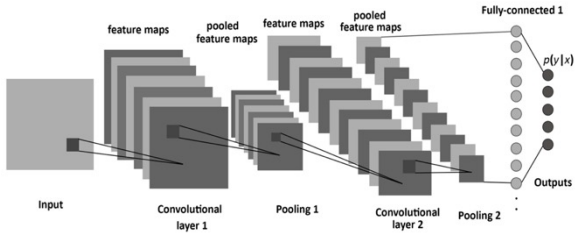
Expression

Node in a computational graph (data, grad)



Module

A neural network layer

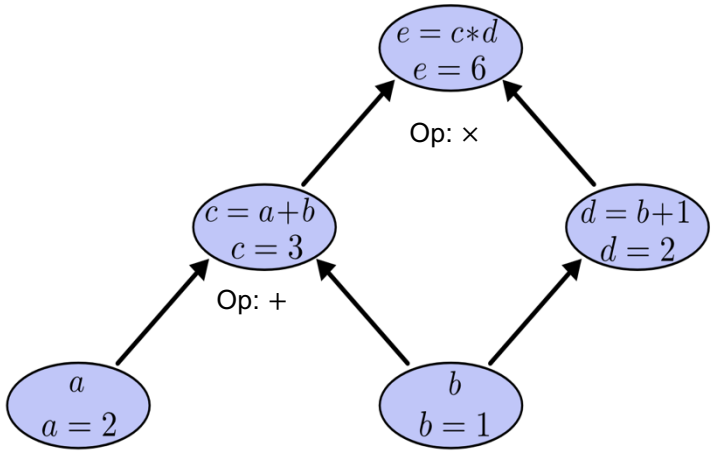


$e = (a + b) \times (b + 1)$: Forward

```
a = torch.tensor([2.], requires_grad=True)
b = torch.tensor([1.], requires_grad=True)

c = a + b
d = b + 1
e = c * d

print(c)
print(d)
print(e)
```

```
tensor([3.], grad_fn=<AddBackward0>)
tensor([2.], grad_fn=<AddBackward0>)
tensor([6.], grad_fn=<MulBackward0>)
```

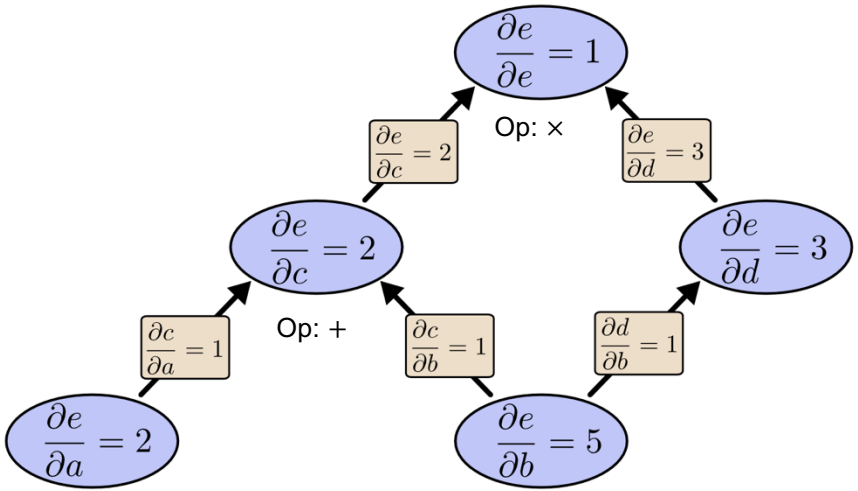
$$\frac{\partial e}{\partial a} = b + 1, \frac{\partial e}{\partial b} = 2b + a + 1$$



$e = (a + b) \times (b + 1)$:
Backward

$$\frac{\partial e}{\partial a} = b + 1, \frac{\partial e}{\partial b} = 2b + a + 1$$

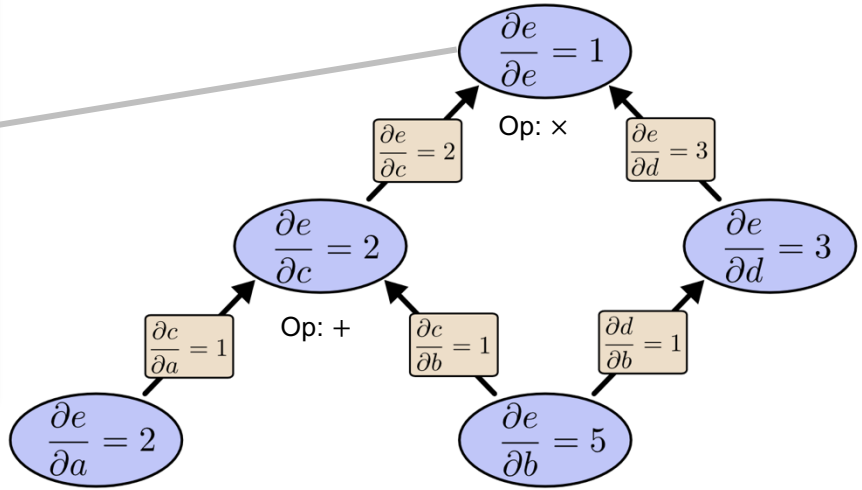
```
e.backward()

print(a.grad)
print(b.grad)

tensor([2.])
tensor([5.])
```



$e = (a + b) \times (b + 1)$:
Backward

```
class MultiplyNode:
  def __init__(self, node1, node2):
    self.node1 = node1
    self.node2 = node2

  def forward(self):
    self.val = self.node1.val * self.node2.val

  def backward(self):
    node1.err = self.err * self.node2.val
    node2.err = self.err * self.node1.val

terminal_node.error = 1
for node in topological_order(cg.nodes):
  if node is not terminal:
    node.backward()
```

$$\frac{\partial e}{\partial a} = b + 1, \frac{\partial e}{\partial b} = 2b + a + 1$$



$$e = (a + b) \times (b + 1)$$

TensorFlow

```
import tensorflow as tf

a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)

c = a + b
d = b + 1
e = c * d

grad_a, grad_b = tf.gradients(e, [a, b])

with tf.Session() as sess:
  grad_a, grad_b = sess.run([grad_a, grad_b],
                            feed_dict={a: 2., b: 1.})
  print(grad_a)
  print(grad_b)
```

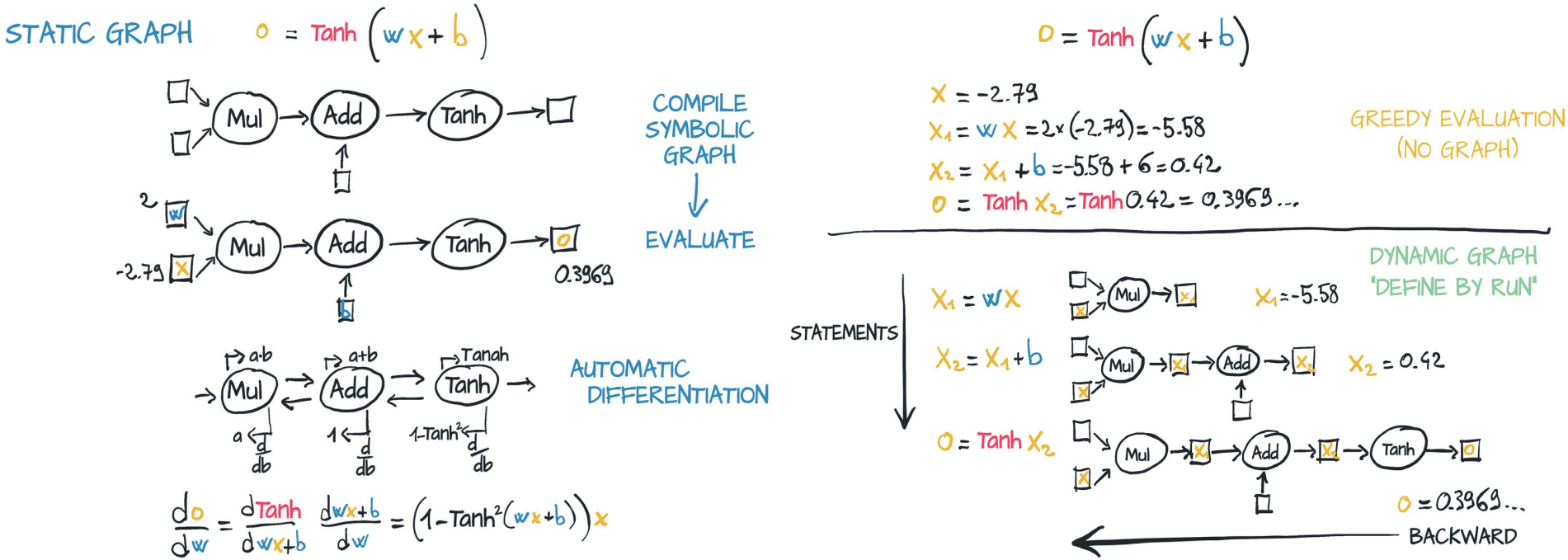
```
2.0
5.0
```

PYTORCH

```
import torch
a = torch.tensor([2.], requires_grad=True)
b = torch.tensor([1.], requires_grad=True)

c = a + b
d = b + 1
e = c * d

e.backward()

print(a.grad)
print(b.grad)
```

```
tensor([2.])
tensor([5.])
```

Static Graph vs. Dynamic Graph



Imperative vs. Declarative

Imperative (e.g. PyTorch)

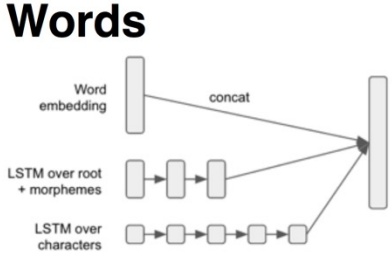
- Implicitly defining the model as execution goes
- ADVANTAGES
  - Natural, flexible program model
  - Ability to use Python ecosystem
  - Easy to debug
- DISADVANTAGES
  - Difficult to optimize
  - No clear model serialization
  - Overhead can be significant

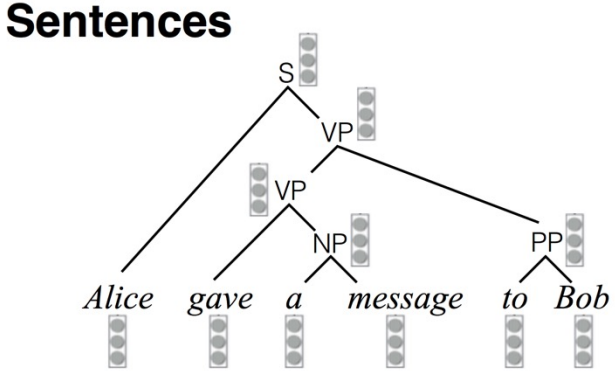
Declarative (e.g. TensorFlow)

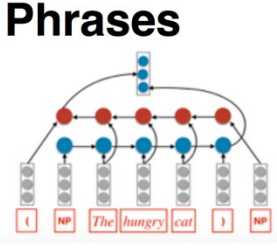
- Declare and compile a model
- Repeatedly execute the model
- ADVANTAGES
  - End-to-end program optimization
  - Easy to serialize for production deployment
- DISADVANTAGES
  - Non-intuitive programing model
  - Difficult to design and maintain

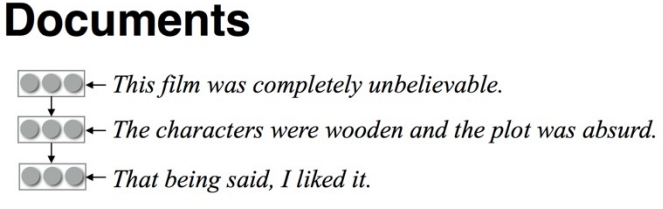
Dynamic Structure

- Hierarchical structures exist in language





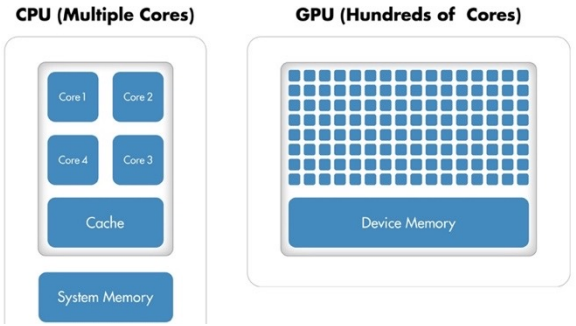




Deep learning from the perspective of SE
abstraction and reuse

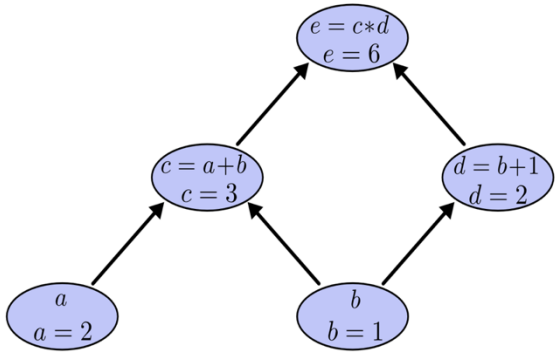
Tensor

Imperative ndarray



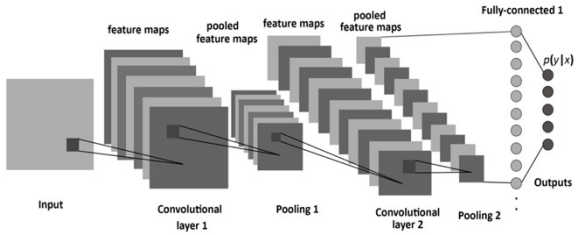
Expression

Node in a computational graph (data, grad)



Module

A neural network layer



Major Components of PYTORCH

Package	Abstraction lvl.	Description
`torch`	Tesnor / Expression	a Tensor library like NumPy, with strong GPU support. When configured with requires_grad, it supports differentiation
`torch.nn`	Module	a neural networks library deeply integrated with autograd designed for maximum flexibility
`torch.optim`	Module	an optimization package to be used with torch.nn with standard optimization methods such as SGD, RMSProp, LBFGS, Adam etc.
`torch.utils`	Module	DataLoader, Dataset and other utility functions for convenience

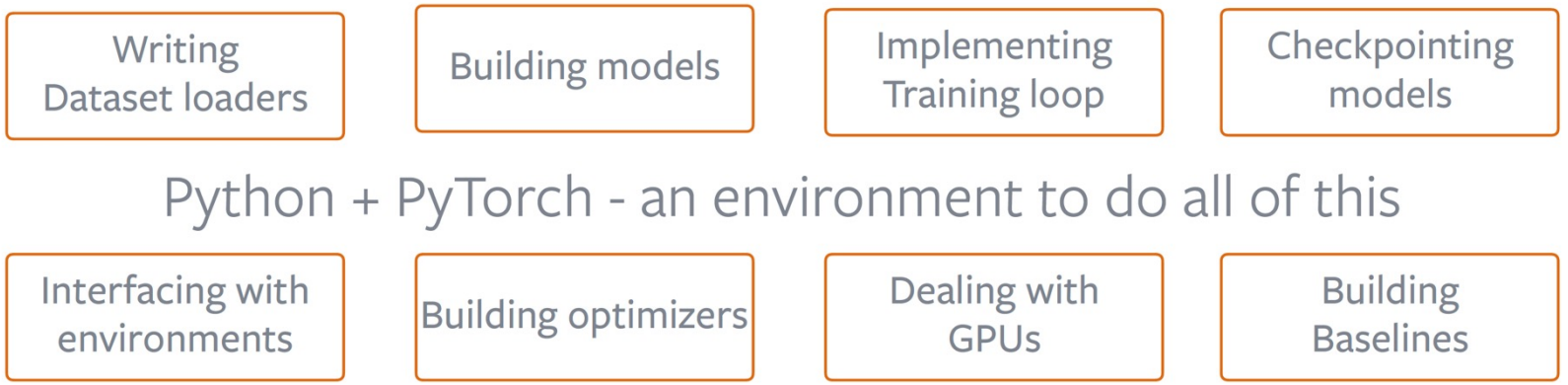
`torch.nn`
a neural networks library

Containers	`Module, Sequential, ModuleList, ParameterList`
Convolution Layers	`Conv1d, Conv2d, Conv3d …`
Recurrent Layers	`RNN, LSTM, GRU, RNNCell …`
Linear Layers	`Linear, Bilinear`
Non-linear Activations	`ReLU, Sigmoid, Tanh, LeakyReLU …`
Loss Functions	`NLLLoss, BCELoss, CrossEntropyLoss …`
Dropout	Dropout, Dropout2d, AlphaDropout …
Initialization	xavier_uniform_, kaiming_uniform_, orthogonal_ …

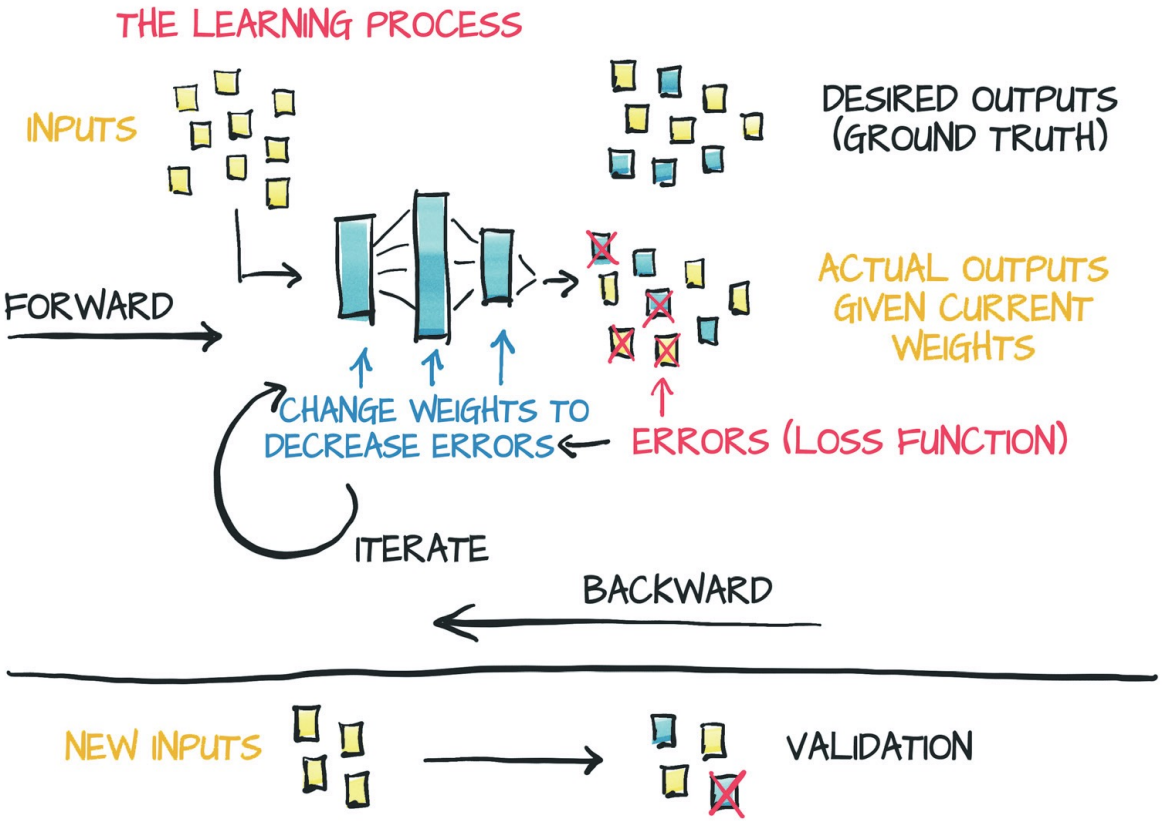
Outline

- Why deep learning libraries
- A brief introduction to PyTorch
- PyTorch in action

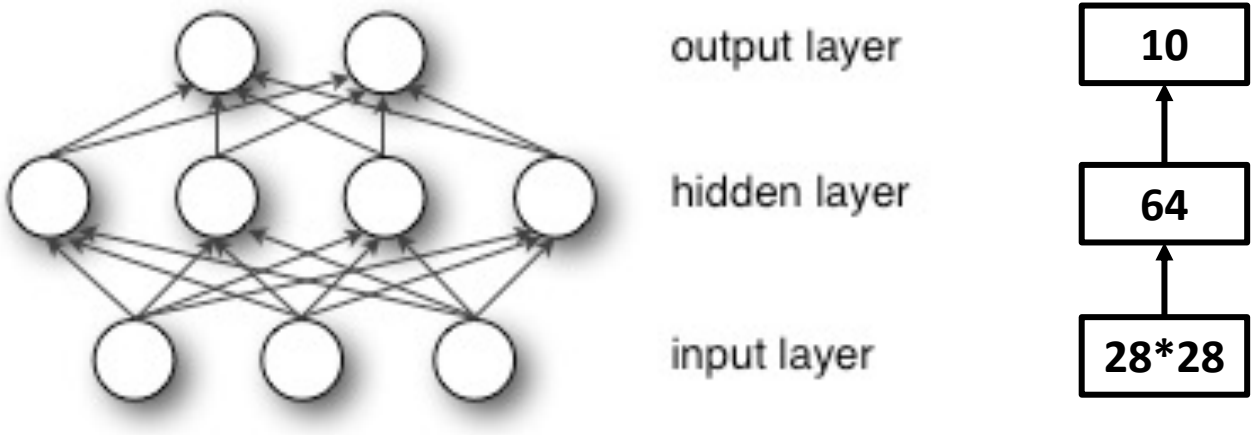
Work items in practice



The learning process



MLP for MNIST



MLP for MNIST – Dataset loader

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms

if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

batch_size = 32

train_dataset = datasets.MNIST('./data', train=True, download=True, transform=transforms.ToTensor())
validation_dataset = datasets.MNIST('./data', train=False, transform=transforms.ToTensor())

train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
validation_loader = torch.utils.data.DataLoader(dataset=validation_dataset, batch_size=batch_size, shuffle=False)
```

MLP for MNIST – Build model

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 64)
        self.fc2 = nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = F.relu(self.fc1(x))
        return F.log_softmax(self.fc2(x), dim=1)
```

MLP for MNIST – Train

Build optimizer

```
model = Net().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.CrossEntropyLoss()
```

Training loop

```
def train(epoch, log_interval=200):
    # Set model to training mode
    model.train()

    # Loop over each batch from the training set
    for batch_idx, (data, target) in enumerate(train_loader):
        # Copy data to GPU if needed
        data = data.to(device)
        target = target.to(device)
        # Zero gradient buffers
        optimizer.zero_grad()
        # Pass data through the network
        output = model(data)
        # Calculate loss
        loss = criterion(output, target)
        # Backpropagate
        loss.backward()
        # Update weights
        optimizer.step()

        if batch_idx % log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.data.item()))
```

MLP for MNIST – Test

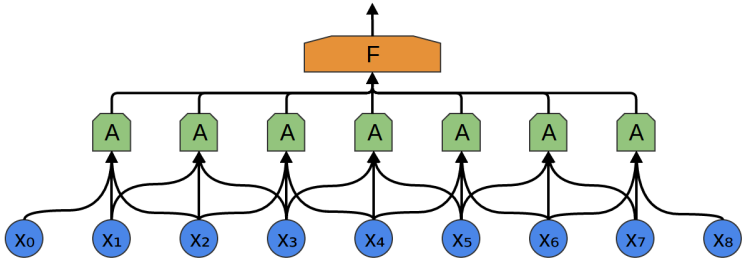
```
def validate(loss_vector, accuracy_vector):
    model.eval()
    val_loss, correct = 0, 0
    for data, target in validation_loader:
        data = data.to(device)
        target = target.to(device)
        output = model(data)
        val_loss += criterion(output, target).data.item()
        # get the index of the max log-probability
        pred = output.data.max(1)[1]
        correct += pred.eq(target.data).cpu().sum()

    val_loss /= len(validation_loader)
    loss_vector.append(val_loss)

    accuracy = 100. * correct.to(torch.float32) / \
        len(validation_loader.dataset)
    accuracy_vector.append(accuracy)

    print('\nValidation set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        val_loss, correct, len(validation_loader.dataset), accuracy))
```

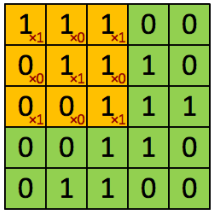
MLP for MNIST – Run

```
epochs = 10

lossv, accv = [], []
for epoch in range(1, epochs + 1):
    train(epoch)
    validate(lossv, accv)
```

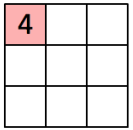
Test Accuracy: ~96%

CNN for MNIST

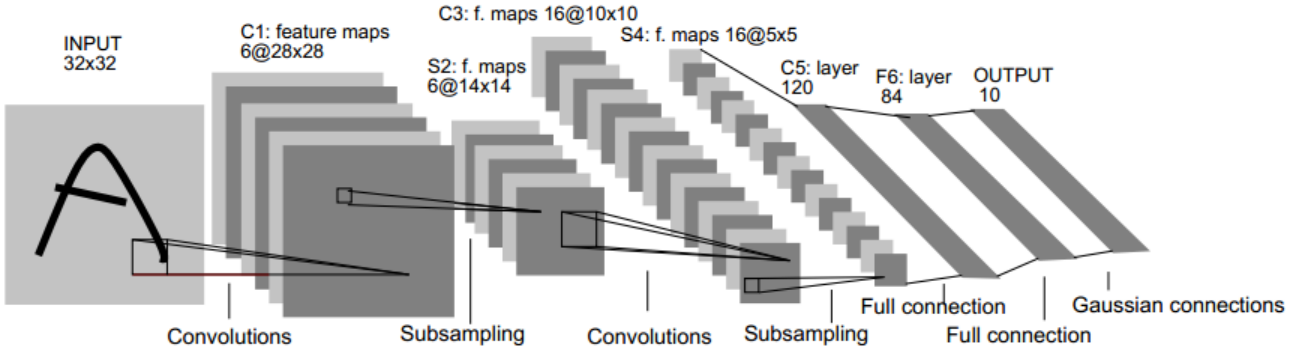




Image



Convolved Feature



CNN for MNIST

Test Accuracy: ~99%

```python
class LeNet(nn.Module):
    def __init__(self, n_class=10):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(
            in_channels = 1,
            out_channels = 20,
            kernel_size = 5
        )
        self.conv2 = nn.Conv2d(
            in_channels = 20,
            out_channels = 50,
            kernel_size = 5
        )
        self.fc1 = nn.Linear(4*4*50, 500)
        self.fc2 = nn.Linear(500, n_class)

    def forward(self, x):
        x = F.relu(self.conv1(x))    # x:[batch_size,1,28,28] => x:[batch_size,20, 24, 24]
        x = F.max_pool2d(x, 2, 2)    # x:[batch_size,20,24,24] => x:[batch_size,20, 12, 12]
        x = F.relu(self.conv2(x))    # x:[batch_size,20,12,12] => x:[batch_size,50, 8, 8]
        x = F.max_pool2d(x, 2, 2)    # x:[batch_size,50,8,8] => x:[batch_size,50, 4, 4]
        x = x.view(-1, 4*4*50)       # x:[batch_size,50,4,4] => x:[batch_size,50*4*4]
        x = F.relu(self.fc1(x))      # x:[batch_size,50*4*4] => x:[batch_size,500]
        x = self.fc2(x)              # x:[batch_size,500] => x:[batch_size,10]
        return x
```

Assignments

1. Try more models (Bi-LSTM, Seq2Seq…)
2. L2 Regularization
3. Dropout
4. Data Argumentation
5. Batch Normalization
6. Try other optimization algorithms (SGD…)
7. Try more activation function (Tanh, Sigmoid…)
8. Gradient Clipping