BIGSKYEARTH

Big Data Era in Sky and Earth Observation

TD COST Action TD 1403

BSETS 2018

Deep Learning Frameworks

Marc Masana (mmasana@cvc.uab.cat)

COST

EUROPEAN COOPERATION IN SCIENCE & TECHNOLOGY

Overview

- Introduction - CPU vs GPU.
- Overview of Deep Learning Frameworks
- Hands on Tensorflow
- A glimpse of PyTorch
- Tensorboard

Introduction

CPU vs GPU

	model	# cores	clock speed	memory	price
CPU fewer cores, but each core is much faster and with much higher capabilities; great at sequential tasks.	Intel Core i7-7700k	4 (8 threads with hyperthreading)	4.4 GHz	Shared with system	279€
	Intel Core i7-6950X	10 (20 threads with hyperthreading)	3.5 GHz	Shared with system	1,399€
GPU more cores, but each core is much slower and "simpler"; great for parallel tasks.	NVIDIA Titan Xp	3,840	1.60 GHz	12Gb GDDR5	*1,299€
	NVIDIA GTX 1070 Ti	2,432	1.68 GHz	8Gb GDDR5	469€
	NVIDIA GTX 1080 Ti	3,584	1.58 GHz	11Gb GDDR5	*769€

* currently unavailable

- CUDA (NVIDIA only)
  - Write C-like code that runs directly on the GPU
  - Higher-level APIs: cuBLAS, cuFFT, cuDNN, etc.
  - **Spoiler alert** more to come on lectures after Easter.
- OpenCL
  - Runs on anything.
  - Usually slower.

CPU vs GPU

- Performance in practice
  - Comparison on VGG-19 used in the ILSVRC-2014 competition.
  - Notice that GPU performs much better than CPU.
  - But also that cuDNN is better than “unoptimized” CUDA.

	model	cuDNN	Forward (ms)	Backward (ms)	Total (ms)
CPU	Dual Xeon E5-2630 v3	-	3609.78	6239.45	9849.23
GPU	NVIDIA Titan Xp	-	121.69	318.39	440.08
	NVIDIA GTX 1080 Ti	-	176.36	453.22	629.57
	NVIDIA Titan Xp	5.1	48.09	99.23	147.32
	NVIDIA GTX 1080 Ti	5.1	48.15	100.04	148.19

Data from https://github.com/jcjohnson/cnn-benchmarks

CPU / GPU Communication

If you are not careful, training can bottleneck on reading data and transferring it to the GPU.

Solutions:

- Read all data into RAM (if the dataset is small enough).
- Use SSD instead of HDD (will reduce the bottleneck but might not totally solve the problem.
- Use multiple CPU threads to prefetch data (best solution).

Deep Learning Frameworks

Caffe

torch

theano

TensorFlow

PYTORCH

Chainer

dmlc mxnet

Microsoft CNTK

K

DL4J DEEPLEARNING4J

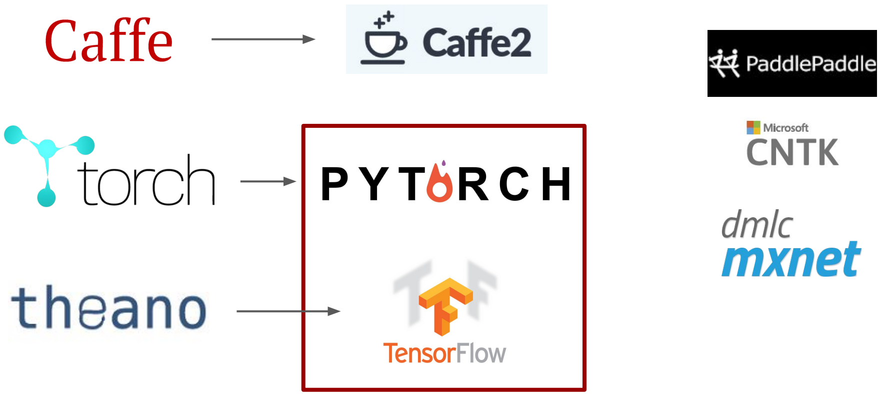
PaddlePaddle

Introduction

What is the point of deep learning frameworks?

1. Easily build large computational graphs.
2. Easily compute gradients in computational graphs.
3. Run all of it efficiently on a GPU.

This year



Which framework?

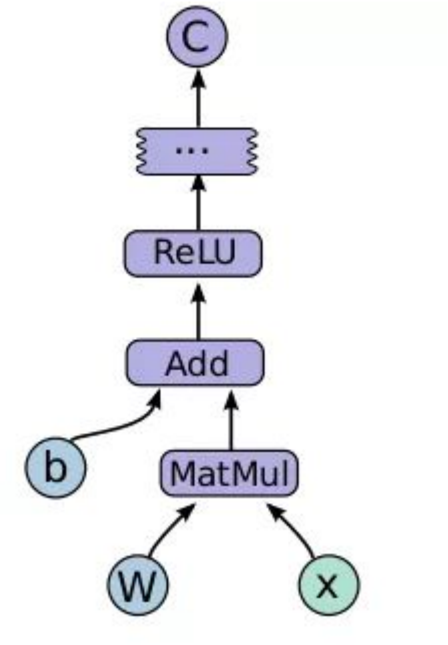
		Languages	Tutorials and training materials	CNN modeling capability	RNN modeling capability	Architecture: easy-to-use and modular front end	Speed	Multiple GPU support	Keras compatible
Montreal Univ.	Theano	Python, C++	++	++	++	+	++	+	+
Google Brain	Tensor-Flow	Python	+++	+++	++	+++	++	++	+
Collobert et al.	Torch	Lua, Python (new)	+	+++	++	++	+++	++	
BVL and FB	Caffe	C++	+	++		+	+	+	
Apache	MXNet	R, Python, Julia, Scala	++	++	+	++	++	+++	
Intel	Neon	Python	+	++	+	+	++	+	
Microsoft	CNTK	C++	+	+	+++	+	++	+	

By Matthew Rubashkin, Silicon Valley Data Science (March 2017)

Symbolic vs Imperative

Symbolic computation frameworks:

- **Network:** a symbolic graph of tensor operations
  (e.g., add, multiply, conv).
- **Layer:** a composition of tensor operations.
- **Examples:** Theano, TensorFlow
- **Require high level libraries:** e.g., Keras, Lasagne, Blocks



Imperative frameworks:

- **Network:** a graph of layers (Dense, Conv2D, Pool)
- **Layer:** Implemented in low level imperative language (e.g., c++)
- **Examples:** Caffe, Torch

Symbolic vs Imperative

Symbolic frameworks

PROS:

- Very **expressive**
- Are **compilers**
  - Compile into **different languages**
  - **Automatic optimization**
  - **Automatic differentiation**
  - **Better memory reuse**

CONS:

- **Need of high level libraries**
- **Much compilation time**
- **Worse performance**
- **Think symbolically**
- **Difficult to debug**

Non-symbolic frameworks

PROS:

- **Easy to create a network**
- Internal **code easy to understand**
- **No compilation** overhead
- **Faster**
- **Easy to debug**

CONS:

- **Less expressive**: New layers have to be implemented from scratch
- The **gradients have to be computed manually**
- It **requires manual optimization**
- It **requires more GPU memory**

Static vs Dynamic

Static graphs:

- Build graph once, then run many times.
- Once built, can be **serialized** and run it without the code that built the graph.
- The framework can **optimize** the graph for you before it runs.
- Can use **special control flow operators** for conditionals and loops.
- As in Tensorflow.

Dynamic graphs:

- Each forward pass defines a new graph.
- Graph building and execution are **connected**.
- Makes it easier for **recurrent** networks and **modular** networks.
- As in PyTorch.

Popularity

Top libraries by Github issues opened		
#1:	8370	tensorflow/tensorflow
#2:	5806	fchollet/keras
#3:	4558	dmlc/mxnet
#4:	3908	BVLC/caffe
#5:	2465	Theano/Theano
#6:	2462	baidu/paddle
#7:	2264	deeplearning4j/deeplearning4j
#8:	2124	Microsoft/CNTK
#9:	1601	pytorch/pytorch
#10:	1139	NVIDIA/DIGITS
#11:	1005	pfnet/chainer
#12:	738	caffe2/caffe2
#13:	709	tflearn/tflearn
#14:	664	davisking/dlib
#15:	575	torch/torch7
#16:	488	Lasagne/Lasagne
#17:	469	clab/dynet
#18:	324	NervanaSystems/neon
#19:	47	deepmind/sonnet

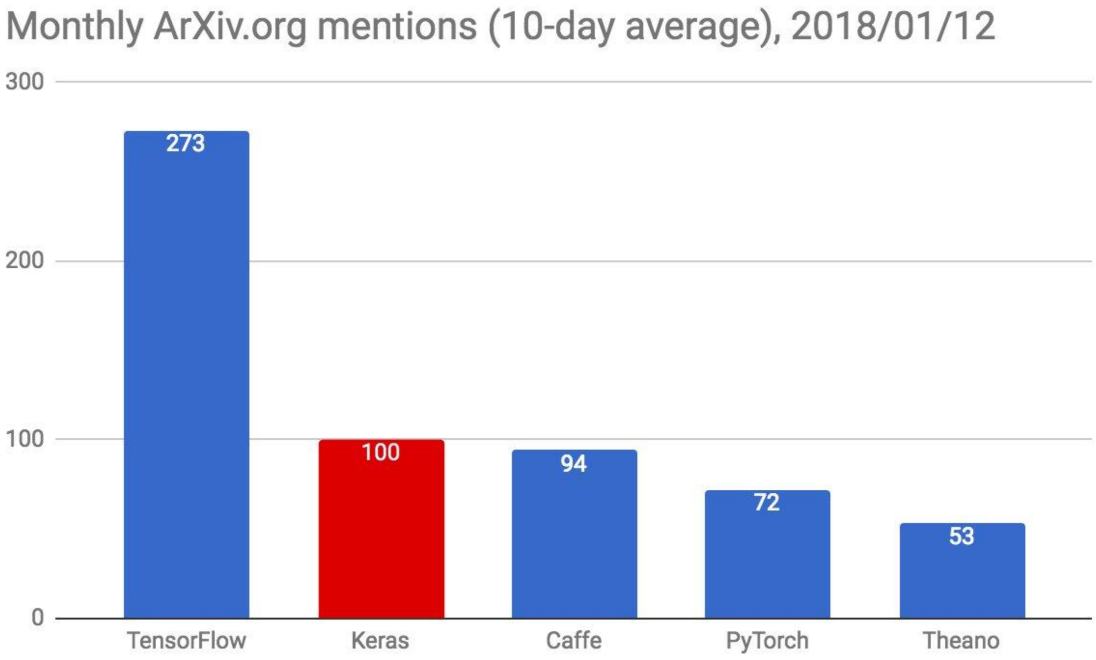
Top libraries by Github stars		
#1:	71627	tensorflow/tensorflow
#2:	20489	BVLC/caffe
#3:	20038	fchollet/keras
#4:	12558	Microsoft/CNTK
#5:	11369	dmlc/mxnet
#6:	7712	pytorch/pytorch
#7:	7332	torch/torch7
#8:	7297	deeplearning4j/deeplearning4j
#9:	6981	Theano/Theano
#10:	6767	tflearn/tflearn
#11:	5742	caffe2/caffe2
#12:	5544	baidu/paddle
#13:	5336	deepmind/sonnet
#14:	3242	Lasagne/Lasagne
#15:	3232	NervanaSystems/neon
#16:	2987	pfnet/chainer
#17:	2833	davisking/dlib
#18:	2525	NVIDIA/DIGITS
#19:	1775	clab/dynet

Top libraries by Github contributors		
#1:	1079	tensorflow/tensorflow
#2:	537	fchollet/keras
#3:	432	dmlc/mxnet
#4:	322	Theano/Theano
#5:	318	pytorch/pytorch
#6:	249	BVLC/caffe
#7:	149	Microsoft/CNTK
#8:	139	pfnet/chainer
#9:	134	torch/torch7
#10:	125	deeplearning4j/deeplearning4j
#11:	125	caffe2/caffe2
#12:	104	tflearn/tflearn
#13:	84	clab/dynet
#14:	79	davisking/dlib
#15:	77	baidu/paddle
#16:	70	NervanaSystems/neon
#17:	62	Lasagne/Lasagne
#18:	42	NVIDIA/DIGITS
#19:	16	deepmind/sonnet

Top libraries by Github forks		
#1:	35371	tensorflow/tensorflow
#2:	12575	BVLC/caffe
#3:	7293	fchollet/keras
#4:	4256	dmlc/mxnet
#5:	3659	deeplearning4j/deeplearning4j
#6:	3272	Microsoft/CNTK
#7:	2302	Theano/Theano
#8:	2166	torch/torch7
#9:	1599	pytorch/pytorch
#10:	1483	baidu/paddle
#11:	1450	tflearn/tflearn
#12:	1278	caffe2/caffe2
#13:	974	davisking/dlib
#14:	921	NVIDIA/DIGITS
#15:	908	Lasagne/Lasagne
#16:	798	pfnet/chainer
#17:	716	NervanaSystems/neon
#18:	656	deepmind/sonnet
#19:	447	clab/dynet

Francois Chollet @fchollet (Twitter Oct 2017)

Popularity



Francois Chollet @fchollet (Twitter)

Hands on

TensorFlow

Simple Example

Simple Example

Training a two layer network (with ReLU but no biases) on random data with an L2-loss

N: batch size
D: in/out dimension size
H: hidden layer size

```
import numpy as np
import tensorflow as tf

N, D, H = 64, 1000, 100
x  = tf.placeholder(tf.float32, shape=(N, D))
y  = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D)}
    out = sess.run([loss, grad_w1, grad_w2], feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

Simple Example

<u>**Define** computational graph</u>

1. Create the **placeholders** for the inputs, outputs (labels) and weights.

2. Define the **forward pass** (network, metrics and losses).

3. Tell Tensorflow to compute the **loss** of the gradient with respect to the weights.

No computation is happening during this part!

```
N, D, H = 64, 1000, 100
x  = tf.placeholder(tf.float32, shape=(N, D))
y  = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])
```

Simple Example

Run the graph

1. Enter a **session** so we can run the graph.

2. Create the **data** that will fill the placeholders.

3. **Run** the graph: feed the inputs and get the loss arrays for each weight matrix.

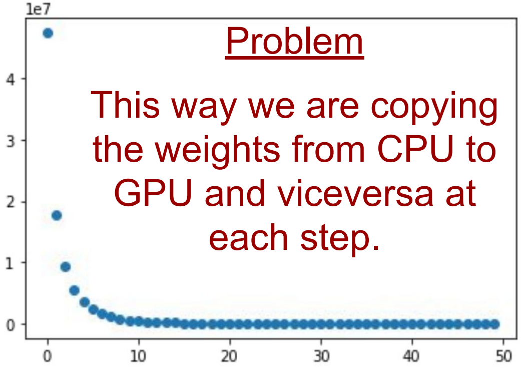
However, this is only doing it one time.

```
with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D)}
    out = sess.run([loss, grad_w1, grad_w2], feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

Simple Example

Train the network

Run the graph repeatedly and use the gradient to update the weights in order to learn the task.



Problem

This way we are copying the weights from CPU to GPU and viceversa at each step.

```
with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D)}
    learning_rate = 1e-5
    for t in range(50):
        out = sess.run([loss, grad_w1, grad_w2], feed_dict=values)
        loss_val, grad_w1_val, grad_w2_val = out
        values[w1] -= learning_rate * grad_w1_val
        values[w2] -= learning_rate * grad_w2_val
```

Simple Example

Updating weights

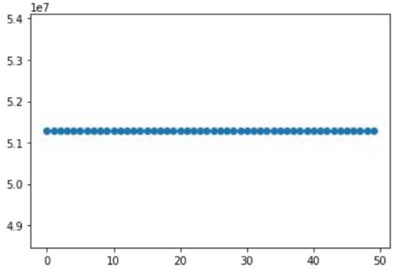
1. Replace weights from placeholder to **Variable**, which persist in the graph between calls.

2. Add **assign** operations to do the weight updates as part of the graph.

3. **Initialize** the variables from the graph.

Problem

Loss does not go down. Assign calls are not executed.




```
N, D, H = 64, 1000, 100
x  = tf.placeholder(tf.float32, shape=(N, D))
y  = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1_val)
new_w2 = w2.assign(w2 - learning_rate * grad_w2_val)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D)}
    for t in range(50):
        loss_val = sess.run([loss], feed_dict=values)
```

Simple Example

Updating weights

We can add a simple graph node that deals with the updates, and then tell the graph to compute it.

```
N, D, H = 64, 1000, 100
x  = tf.placeholder(tf.float32, shape=(N, D))
y  = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1_val)
new_w2 = w2.assign(w2 - learning_rate * grad_w2_val)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D)}
    for t in range(50):
        loss_val = sess.run([loss, updates], feed_dict=values)
```

Optimizers and Losses

<u>Optimizers</u>

Can use an optimizer to compute the gradients and update weights. Remember to run the output of the optimizer on the graph.

<u>Losses</u>

Can use predefined common losses.

```
N, D, H = 64, 1000, 100
x  = tf.placeholder(tf.float32, shape=(N, D))
y  = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.losses.mean_squared_error(y_pred, y)

learning_rate = 1e-5
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D)}
    for t in range(50):
        loss_val = sess.run([loss, updates], feed_dict=values)
```

Layers

Initialization

Can use predefined weight initializations.

Layers

Using Tensorflow Layer class automatically sets up weights and biases. There are different predefined layers that can be used to build most known architectures.

```
N, D, H = 64, 1000, 100
x  = tf.placeholder(tf.float32, shape=(N, D))
y  = tf.placeholder(tf.float32, shape=(N, D))

init = tf.contrib.layers.xavier_initializer()
h = tf.layers.dense(inputs=x, units=H,
                    activation=tf.nn.relu, kernel_initializer=init)
y_pred = tf.layers.dense(inputs=h, units=D,
                         kernel_initializer=init)

loss = tf.losses.mean_squared_error(y_pred, y)

learning_rate = 1e-5
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D)}
    for t in range(50):
        loss_val = sess.run([loss, updates], feed_dict=values)
```

Graphs and scopes

tf.Graph():

- The graph default is instantiated when the library is imported.
- Can create Graph objects instead of default to have multiple models in a file.
- Each Graph will not depend on each other.

with *graph1*.as _default():

- Variables outside this instance will go to the default graph.
- You can get the *handle* for it.

tf.name_scope(*string*):

- Allows to break down the model into individual pieces, which helps control the complexity of large networks.
- Can be used with Tensorboard.
- Can be nested inside other scope to create hierarchies.

Sessions

tf.Session():

- Encapsulates the environments of operations and tensors to be executed/evaluated.
- Sessions can have their own variables, queues and readers.
- Need to close() the session when finished.
- Arguments:
  - target: the execution engine to connect to,
  - graph to be launched,
  - config: protocol buffer with configuration options for the session.

tf.InteractiveSession():

- Is the exact same as above but is targeted for using IPython and Jupyter Notebooks.
- It allows to not pass the Session object explicitly.

Wrappers

Other **High-Level Wrappers** for Tensorflow

- Keras (https://keras.io/)
- TFLearn (http://tflearn.org/)
- TensorLayer (http://tensorlayer.readthedocs.io/en/latest/)
- Tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)
- TF-Slim (https://github.com/tensorflow/models/tree/master/inception/inception/slim)
- Tf.contrib.learn (https://www.tensorflow.org/get_started/tflearn)
- Pretty Tensor (https://github.com/google/prettytensor) → from Google
- Sonnet (https://github.com/deepmind/sonnet) → from DeepMind

Come with Tensorflow

A glimpse of

PYTORCH

Main concepts

- There is no built-in notion of computational graph, gradients, or even deep learning.
- However, Tensors and Variables have the same API and remember how they were created.

Concept	Tensorflow equivalent
Tensor: imperative array that runs on GPU	Numpy array
Variable: computational graph node that stores data and gradient	Tensor, Variable, Placeholder
Module: neural network layer that stores state or learnable weights	tf.layers, TF-Slim, or other wrapper

Simple Example

Simple example

Training a two layer network (with ReLU but no biases) on random data with an L2-loss.

N: batch size
D: feature dimension size
H: hidden layer size
C: number of classes

```
import torch

N, D, H, C = 64, 1000, 100, 10
x = torch.randn(N, D).type(torch.FloatTensor)
y = torch.randn(N, C).type(torch.FloatTensor)
w1 = torch.randn(D, H).type(torch.FloatTensor)
w2 = torch.randn(H, C).type(torch.FloatTensor)

learning_rate = 1e-6
for t in range(50):
    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
```

Simple Example

<u>Train</u> <u>the network</u>

1. Create random **tensors** for data and weights.

2. Compute the **forward pass** (network, metrics and losses).

3. **Manually** compute the **gradients** for the **backward pass**.

4. **Gradient descent** step on weights

To run on GPU, just need to cast the tensors to a cuda datatype like:

```
torch.cuda.FloatTensor
```

```
N, D, H, C = 64, 1000, 100, 10
x = torch.randn(N, D).type(torch.FloatTensor)
y = torch.randn(N, C).type(torch.FloatTensor)
w1 = torch.randn(D, H).type(torch.FloatTensor)
w2 = torch.randn(H, C).type(torch.FloatTensor)

learning_rate = 1e-6
for t in range(50):
    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
```

Autograd

<u>Variable</u>

A PyTorch Variable is a node in a computational graph

x.data is a Tensor

x.grad is a Variable of gradients (same shape as x.data)

x.grad.data is a Tensor of gradients

N: batch size
D: feature dimension size
H: hidden layer size
C: number of classes

```
import torch
from torch.autograd import Variable

N, D, H, C = 64, 1000, 100, 10
x = Variable(torch.randn(N, D), requires_grad=False)
y = Variable(torch.randn(N, C), requires_grad=False)
w1 = Variable(torch.randn(D, H), requires_grad=True)
w2 = Variable(torch.randn(H, C), requires_grad=True)

learning_rate = 1e-6
for t in range(50):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1 -= learning_rate * w1.grad.data
    w2 -= learning_rate * w2.grad.data
```

Autograd

Input variables don't want gradients; weights do. ■

Forward pass is the same but everything is a Variable. ■

Zero out the gradients first and then compute the gradient of the loss. ■

Make the gradient descent step. ■

```
import torch
from torch.autograd import Variable

N, D, H, C = 64, 1000, 100, 10
x = Variable(torch.randn(N, D), requires_grad=False)
y = Variable(torch.randn(N, C), requires_grad=False)
w1 = Variable(torch.randn(D, H), requires_grad=True)
w2 = Variable(torch.randn(H, C), requires_grad=True)

learning_rate = 1e-6
for t in range(50):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1 -= learning_rate * w1.grad.data
    w2 -= learning_rate * w2.grad.data
```

Autograd

<u>New functions</u>

Define your own autograd functions by writing forward and backward for Tensors.

```
class ReLU(torch.autograd.Function):
    def forward(self, x):
        self.save_for_backward(x)
        return x.clamp(min=0)

    def backward(self, grad_y):
        x, = self.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```

nn Wrapper

Higher-level **wrapper** for working with neural networks, similar to Keras.

```
import torch
from torch.autograd import Variable

N, D, H, C = 64, 1000, 100, 10
x = Variable(torch.randn(N, D))
y = Variable(torch.randn(N, C), requires_grad=False)

model = torch.nn.Sequential(
            torch.nn.Linear(D, H),
            torch.nn.ReLU(),
            torch.nn.Linear(H, C))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-5
for t in range(50):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

Optimizer

Use an **optimizer** for different update rules and update all the parameters after computing the gradients.

```
import torch
from torch.autograd import Variable

N, D, H, C = 64, 1000, 100, 10
x = Variable(torch.randn(N, D))
y = Variable(torch.randn(N, C), requires_grad=False)

model = torch.nn.Sequential(
            torch.nn.Linear(D, H),
            torch.nn.ReLU(),
            torch.nn.Linear(H, C))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-5
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for t in range(50):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
```

Modules

Define a new module

A PyTorch Module is a neural network layer which inputs and outputs Variables.

Modules can contain weights (as Variables) or other Modules.

Autograd works on your own modules, no need to define backward pass.

```
import torch
from torch.autograd import Variable

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D, H, C):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D, H)
        self.linear2 = torch.nn.Linear(H, C)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D, H, C = 64, 1000, 100, 10
x = Variable(torch.randn(N, D))
y = Variable(torch.randn(N, C), requires_grad=False)

model = TwoLayerNet(D, H, C)
criterion = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-5
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
for t in range(50):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Data Loaders

A **DataLoader** wraps a **Dataset** and provides mini-batching, shuffling, multithreading, and so on.

When you need to load custom data, just write your own **Dataset** class.

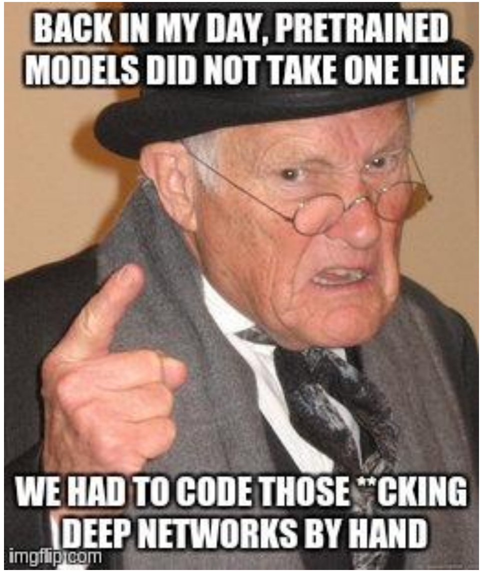
DataLoader gives Tensors, so remember to wrap them in Variables.

```
import torch
from torch.autograd import Variable
from torch.utils.data import TensorDataset, DataLoader

N, D, H, C = 64, 1000, 100, 10
x = torch.randn(N, D)
y = torch.randn(N, C)

loader = DataLoader(TensorDataset(x, y), batch_size=8)

model = TwoLayerNet(D, H, C)
criterion = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-5
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
for epoch in range(10):
    for x_batch, y_batch in loader:
        x_var, y_var = Variable(x), Variable(y)
        y_pred = model(x_var)
        loss = criterion(y_pred, y_var)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Pretrained Models

- Nowadays is super easy to use pretrained models using torchvision.
- Check it out: https://github.com/pytorch/vision

```python
import torch
import torchvision

alexnet = torchvision.models.alexnet(pretrained=True)
vgg16 = torchvision.models.vgg16(pretrained=True)
resnet101 = torchvision.models.resnet101(pretrained=True)
```



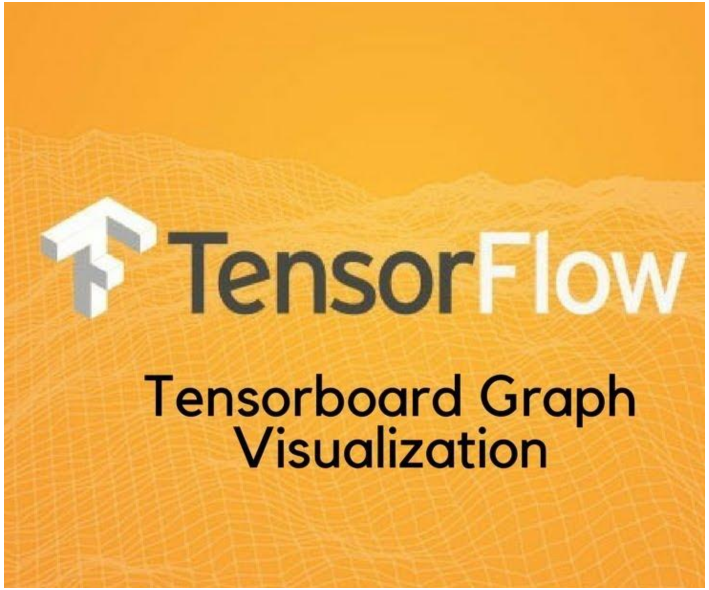
Torch vs PyTorch

	Pros	Cons
Torch	More stable Lots of existing code	Who codes in Lua? No autograd
PyTorch	Python Autograd	Newer and still changing Less existing code (but catching up)

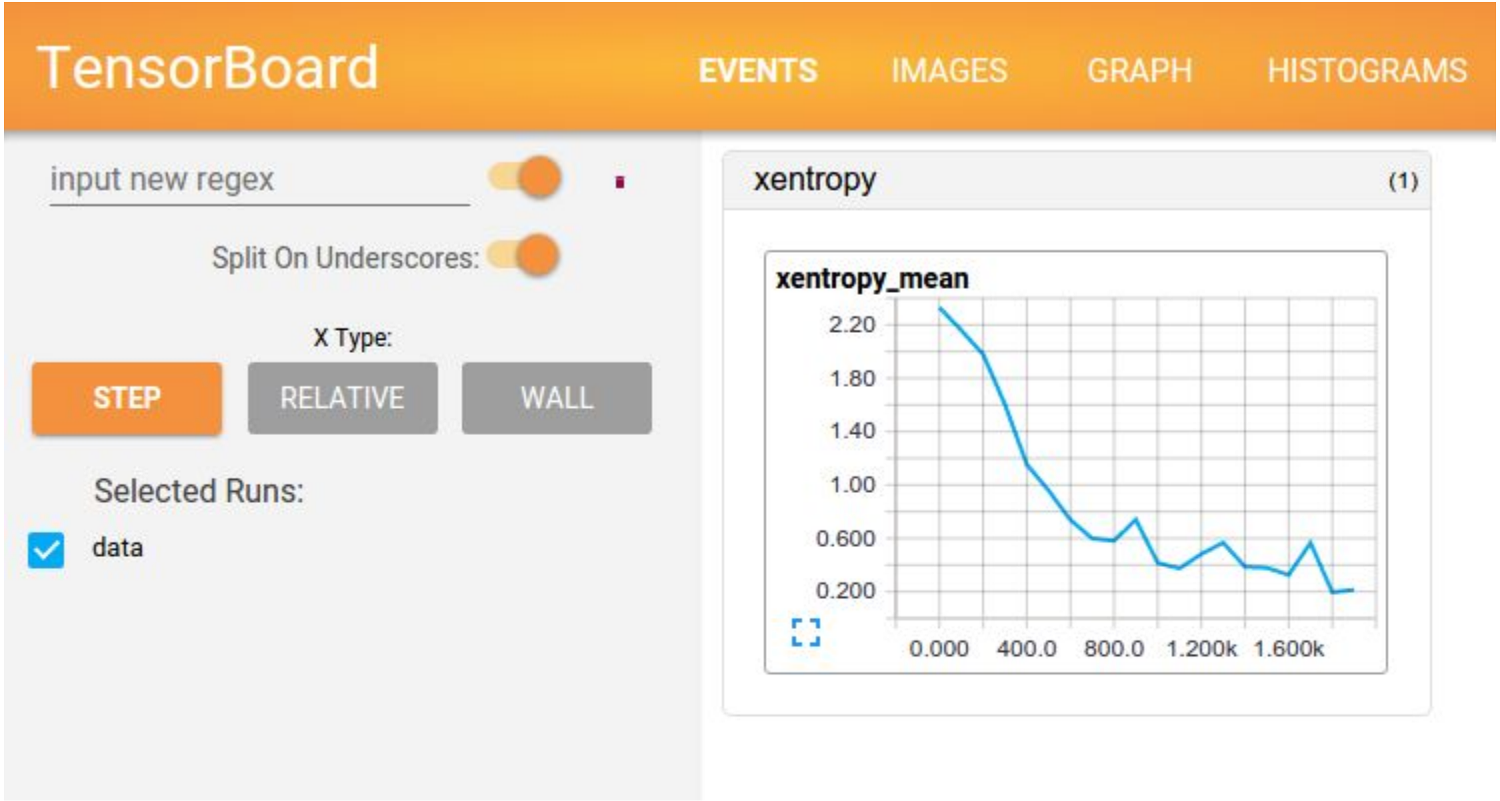
- Probably use PyTorch for new projects.

Choosing tips

- **Tensorflow** is a safe bet for most projects. Not perfect but has a huge community and wide usage. You can pair it with a high-level wrapper. Usable among many machines with a single graph.
- **Pytorch** is one of the best for research, very easy to use. However, it is still a bit new.
- Consider **Caffe** or **Caffe2** for production deployment and if you are familiar with C++.
- Consider **Tensorflow** or **Caffe2** for mobile devices.
- **MatConvNet** is also an option if you are used to it and want a starting point. It allows to load any Caffe model.


TensorFlow
Tensorboard Graph
Visualization

What is Tensorboard?



How to use TensorBoard

1. Write a Log File (**SumaryWriter**). We add the following line before our train loop:

```
writer = tf.train.SummaryWriter(logs_path, graph=tf.get_default_graph())
```

   This will create the log folder and save the graph structure.

2. **Run** TensorBoard:

```
tensorboard --logdir=path/to/log-directory --port #PORT
```

3. Make your **Tensorflow Graph readable**. Add the scope of our variables and a name for our placeholders and variables to clean up the visualization of our model (use *with* statement).

```
with tf.name_scope('input'):
    x = tf.placeholder(tf.float32, shape=[None, 784], name="x-input")
    y_ = tf.placeholder(tf.float32, shape=[None, 10], name="y-input")
```

How to use TensorBoard

4. Log **dynamic values** (1). Add summaries of specific variables like train error. List of summary operators here. All summary operations can be merged.

```
tf.scalar_summary("cost", cross_entropy)
tf.scalar_summary("accuracy", accuracy)
summary_op = tf.merge_all_summaries()
```

5. Log **dynamic values** (2). Once we define them, the summary operations have to be executed inside our train cycle to write the values into the SummaryWriter.

```
_, summary = sess.run([train_op, summary_op], feed_dict={x: batch_x, y_: batch_y})
writer.add_summary(summary, step)
```

6. Remember to **restart** TensorBoard if you make changes and they don't show.

Slides credit and other tutorials

- A Practical Introduction to Deep Learning with Caffe, Peter Anderson
- Stanford Vision course CS231n
- Brewing Deep Networks With Caffe, Rohit Girdhar
- Caffe Tutorial, Princeton University course COS598
- Caffe presentations by Evan Shelhamer, Jeff Donahue, Jon Long, Yangqing Jia, and Ross Girshick
- Comparing deep learning frameworks from Imaging Hub
- Tensorflow in a Nutshell by Camron Godbout
- Main pages and tutorial pages of the DL frameworks mentioned during this lecture.
- How to use TensorBoard by Imanol Schlag.