

Lecture 12:

Software Packages

Caffe / Torch / Theano / TensorFlow

Administrative

Milestones were due 2/17; looking at them this week
Assignment 3 due Wednesday 2/22
If you are using Terminal: BACK UP YOUR CODE!

Caffe

<http://caffe.berkeleyvision.org>

Caffe Overview

From U.C. Berkeley

Written in C++

Has Python and MATLAB bindings

Good for training or finetuning feedforward models

Most important tip...

Don't be afraid to read the code!

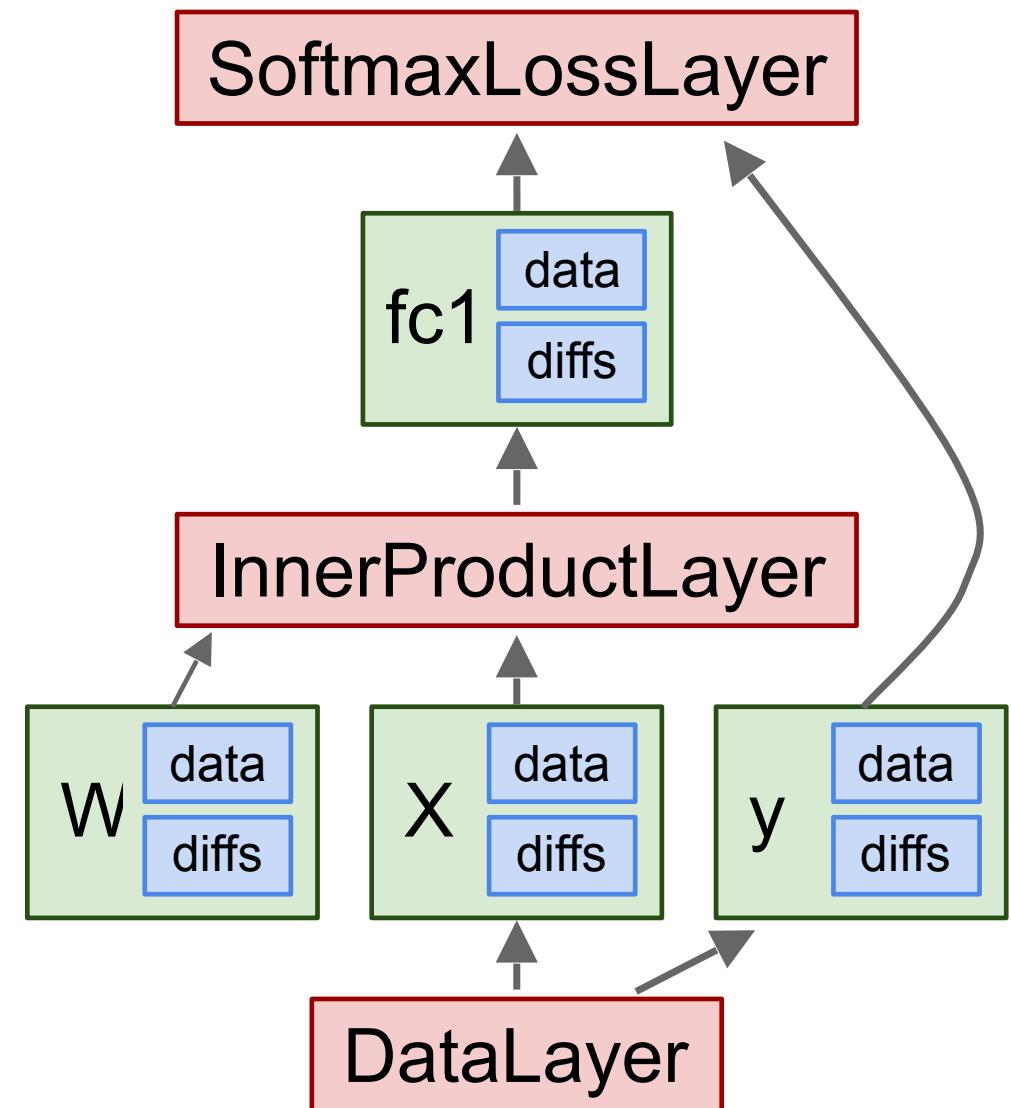
Caffe: Main classes

Blob: Stores data and derivatives ([header](#) [source](#))

Layer: Transforms bottom blobs to top blobs ([header + source](#))

Net: Many layers; computes gradients via forward / backward ([header](#) [source](#))

Solver: Uses gradients to update weights ([header](#) [source](#))



Caffe: Protocol Buffers

“Typed JSON”
from Google

Define “message types”
in .proto files

.proto file

```
message Person {  
    required string name = 1;  
    required int32 id = 2;  
    optional string email = 3;  
}
```

<https://developers.google.com/protocol-buffers/>

Caffe: Protocol Buffers

“Typed JSON”
from Google

Define “message types”
in .proto files

Serialize instances to text
files (.prototxt)

.proto file

```
message Person {  
    required string name = 1;  
    required int32 id = 2;  
    optional string email = 3;  
}
```

.prototxt file

```
name: "John Doe"  
id: 1234  
email: "jdoe@example.com"
```

<https://developers.google.com/protocol-buffers/>

Caffe: Protocol Buffers

“Typed JSON”
from Google

Define “message types”
in .proto files

Serialize instances to text
files (.prototxt)

Compile classes for
[different languages](https://developers.google.com/protocol-buffers/)

Fei-Fei Li & Andrej Karpathy & Justin Johnson

.proto file

```
message Person {  
    required string name = 1;  
    required int32 id = 2;  
    optional string email = 3;  
}
```

Java class

```
Person john = Person.newBuilder()  
.setId(1234)  
.setName("John Doe")  
.setEmail("jdoe@example.com")  
.build();  
output = new FileOutputStream(args[0]);  
john.writeTo(output);
```

.prototxt file

```
name: "John Doe"  
id: 1234  
email: "jdoe@example.com"
```

C++ class

```
Person john;  
fstream input(argv[1],  
ios::in | ios::binary);  
john.ParseFromIstream(&input);  
id = john.id();  
name = john.name();  
email = john.email();
```

Caffe: Protocol Buffers

```
64 message NetParameter {
65   optional string name = 1; // consider giving the network a name
66   // The input blobs to the network.
67   repeated string input = 3;
68   // The shape of the input blobs.
69   repeated BlobShape input_shape = 8;
70
71   // 4D input dimensions -- deprecated. Use "shape" instead.
72   // If specified, for each input blob there should be four
73   // values specifying the num, channels, height and width of the input blob.
74   // Thus, there should be a total of (4 * #input) numbers.
75   repeated int32 input_dim = 4;
76
77   // Whether the network will force every layer to carry out backward operation.
78   // If set False, then whether to carry out backward is determined
79   // automatically according to the net structure and learning rates.
80   optional bool force_backward = 5 [default = false];
81   // The current "state" of the network, including the phase, level, and stage.
82   // Some layers may be included/excluded depending on this state and the states
83   // specified in the layers' include and exclude fields.
84   optional NetState state = 6;
85
86   // Print debugging information about results while running Net::Forward,
87   // Net::Backward, and Net::Update.
88   optional bool debug_info = 7 [default = false];
89
90   // The number of iterations to run for each test net.
91   repeated int32 test_iter = 9;
92
93   // The number of iterations to run for each train net.
94   repeated int32 train_iter = 10;
95
96   // The number of iterations to run for each validation net.
97   repeated int32 val_iter = 11;
98
99   // The number of iterations to run for each test net.
100  repeated int32 test_level = 12;
101
102 message SolverParameter {
103   /////////////////////////////////
104   // Specifying the train and test networks
105   //
106   // Exactly one train net must be specified using one of the following fields:
107   //   train_net_param, train_net, net_param, net
108   // One or more test nets may be specified using any of the following fields:
109   //   test_net_param, test_net, net_param, net
110   // If more than one test net field is specified (e.g., both net and
111   // test_net are specified), they will be evaluated in the field order given
112   // above: (1) test_net_param, (2) test_net, (3) net_param/net.
113   // A test_iter must be specified for each test_net.
114   // A test_level and/or a test_stage may also be specified for each test_net.
115   /////////////////////////////////
116
117   // Proto filename for the train net, possibly combined with one or more
118   // test nets.
119   optional string net = 24;
120   // Inline train net param, possibly combined with one or more test nets.
121   optional NetParameter net_param = 25;
122
123   optional string train_net = 1; // Proto filename for the train net
```

<https://github.com/BVLC/caffe/blob/master/src/caffe/proto/caffe.proto>

<- All Caffe proto types defined here, good documentation!

Caffe: Training / Finetuning

No need to write code!

1. Convert data (run a script)
2. Define net (edit prototxt)
3. Define solver (edit prototxt)
4. Train (with pretrained weights) (run a script)

Caffe Step 1: Convert Data

DataLayer reading from LMDB is the easiest

Create LMDB using [convert_imageset](#)

Need text file where each line is

“[path/to/image.jpeg] [label]”

Create HDF5 file yourself using h5py

Caffe Step 1: Convert Data

ImageDataLayer: Read from image files

WindowDataLayer: For detection

HDF5Layer: Read from HDF5 file

From memory, using Python interface

All of these are harder to use (except Python)

Caffe Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
    top: "data"
    top: "label"
    name: "data"
    type: HDF5_DATA
    hdf5_data_param {
        source: "examples/hdf5_classification/data/train.txt"
        batch_size: 10
    }
    include {
        phase: TRAIN
    }
}
layers {
    bottom: "data"
    top: "fc1"
    name: "fc1"
    type: INNER_PRODUCT
    blobs_lr: 1
    blobs_lr: 2
    weight_decay: 1
    weight_decay: 0
}
inner_product_param {
    num_output: 2
    weight_filler {
        type: "gaussian"
        std: 0.01
    }
    bias_filler {
        type: "constant"
        value: 0
    }
}
layers {
    bottom: "fc1"
    bottom: "label"
    top: "loss"
    name: "loss"
    type: SOFTMAX_LOSS
}
```

Caffe Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
    top: "data" ← Layers and Blobs
    top: "label"
    name: "data" ← often have same
    type: HDF5_DATA           name!
    hdf5_data_param {
        source: "examples/hdf5_classification/data/train.txt"
        batch_size: 10
    }
    include {
        phase: TRAIN
    }
}
layers {
    bottom: "data"
    top: "fc1"
    name: "fc1"
    type: INNER_PRODUCT
    blobs_lr: 1
    blobs_lr: 2
    weight_decay: 1
    weight_decay: 0
}
inner_product_param {
    num_output: 2
    weight_filler {
        type: "gaussian"
        std: 0.01
    }
    bias_filler {
        type: "constant"
        value: 0
    }
}
layers {
    bottom: "fc1"
    bottom: "label"
    top: "loss"
    name: "loss"
    type: SOFTMAX_LOSS
}
```

Caffe Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
    top: "data" ← Layers and Blobs
    top: "label"
    name: "data" ← often have same
    type: HDF5_DATA
    hdf5_data_param {
        source: "examples/hdf5_classification/data/train.txt"
        batch_size: 10
    }
    include {
        phase: TRAIN
    }
}
layers {
    bottom: "data"
    top: "fc1"
    name: "fc1"
    type: INNER_PRODUCT
    blobs_lr: 1 ← Learning rates
    blobs_lr: 2 ← (weight + bias)
    weight_decay: 1 ← Regularization
    weight_decay: 0 ← (weight + bias)
}
inner_product_param {
    num_output: 2
    weight_filler {
        type: "gaussian"
        std: 0.01
    }
    bias_filler {
        type: "constant"
        value: 0
    }
}
layers {
    bottom: "fc1"
    bottom: "label"
    top: "loss"
    name: "loss"
    type: SOFTMAX_LOSS
}
```

Caffe Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
    top: "data" ← Layers and Blobs
    top: "label"
    name: "data" ← often have same
    type: HDF5_DATA
    hdf5_data_param {
        source: "examples/hdf5_classification/data/train.txt"
        batch_size: 10
    }
    include {
        phase: TRAIN
    }
}
layers {
    bottom: "data"
    top: "fc1"
    name: "fc1"
    type: INNER_PRODUCT
    blobs_lr: 1 ← Learning rates
    blobs_lr: 2 ← (weight + bias)
    weight_decay: 1 ← Regularization
    weight_decay: 0 ← (weight + bias)
```

Number of output classes

```
inner_product_param {
    num_output: 2
    weight_filler {
        type: "gaussian"
        std: 0.01
    }
    bias_filler {
        type: "constant"
        value: 0
    }
}
layers {
    bottom: "fc1"
    bottom: "label"
    top: "loss"
    name: "loss"
    type: SOFTMAX_LOSS
}
```

Caffe Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
    top: "data" ← Layers and Blobs
    top: "label"
    name: "data" ← often have same
    type: HDF5_DATA
    hdf5_data_param {
        source: "examples/hdf5_classification/data/train.txt"
        batch_size: 10
    }
    include {
        phase: TRAIN
    }
}
layers {
    bottom: "data"
    top: "fc1"
    name: "fc1"
    type: INNER_PRODUCT
    blobs_lr: 1 ← Learning rates
    blobs_lr: 2 ← (weight + bias)
    weight_decay: 1 ← Regularization
    weight_decay: 0 ← (weight + bias)
}
```

Set these to 0 to
freeze a layer

Number of output
classes

```
inner_product_param {
    num_output: 2
    weight_filler {
        type: "gaussian"
        std: 0.01
    }
    bias_filler {
        type: "constant"
        value: 0
    }
}
layers {
    bottom: "fc1"
    bottom: "label"
    top: "loss"
    name: "loss"
    type: SOFTMAX_LOSS
}
```

Caffe Step 2: Define Net

- .prototxt can get ugly for big models
- ResNet-152 prototxt is 6775 lines long!
- Not “compositional”; can’t easily define a residual block and reuse

```
1 name: "ResNet-152"
2 input: "data"
3 input_dim: 1
4 input_dim: 3
5 input_dim: 224
6 input_dim: 224
7
8 layer {
9     bottom: "data"
10    top: "conv1"
11    name: "conv1"
12    type: "Convolution"
13    convolution_param {
14        num_output: 64
15        kernel_size: 7
16        pad: 3
17        stride: 2
18        bias_term: false
19    }
20}
21
22 layer {
23     bottom: "conv1"
24     top: "conv1"
25     name: "bn_conv1"
26     type: "BatchNorm"
27     batch_norm_param {
28         use_global_stats: true
29     }
30}
```

```
6747 layer {
6748     bottom: "res5c"
6749     top: "pool5"
6750     name: "pool5"
6751     type: "Pooling"
6752     pooling_param {
6753         kernel_size: 7
6754         stride: 1
6755         pool: AVE
6756     }
6757 }
6758
6759 layer {
6760     bottom: "pool5"
6761     top: "fc1000"
6762     name: "fc1000"
6763     type: "InnerProduct"
6764     inner_product_param {
6765         num_output: 1000
6766     }
6767 }
6768
6769 layer {
6770     bottom: "fc1000"
6771     top: "prob"
6772     name: "prob"
6773     type: "Softmax"
6774 }
```

<https://github.com/KaimingHe/deep-residual-networks/blob/master/prototxt/ResNet-152-deploy.prototxt>

Caffe Step 2: Define Net (finetuning)

Original prototxt:

```
layer {
  name: "fc7"
  type: "InnerProduct"
  inner_product_param {
    num_output: 4096
  }
}
[... ReLU, Dropout]
layer {
  name: "fc8"
  type: "InnerProduct"
  inner_product_param {
    num_output: 1000
  }
}
```

Pretrained weights:

```
"fc7.weight": [values]
"fc7.bias": [values]
"fc8.weight": [values]
"fc8.bias": [values]
```

Modified prototxt:

```
layer {
  name: "fc7"
  type: "InnerProduct"
  inner_product_param {
    num_output: 4096
  }
}
[... ReLU, Dropout]
layer {
  name: "my-fc8"
  type: "InnerProduct"
  inner_product_param {
    num_output: 10
  }
}
```

Caffe Step 2: Define Net (finetuning)

Original prototxt:

```
layer {  
    name: "fc7"  
    type: "InnerProduct"  
    inner_product_param {  
        num_output: 4096  
    }  
}  
[... ReLU, Dropout]  
layer {  
    name: "fc8"  
    type: "InnerProduct"  
    inner_product_param {  
        num_output: 1000  
    }  
}
```

Same name:
weights copied

Pretrained weights:

```
"fc7.weight": [values]  
"fc7.bias": [values]  
"fc8.weight": [values]  
"fc8.bias": [values]
```

Modified prototxt:

```
layer {  
    name: "fc7"  
    type: "InnerProduct"  
    inner_product_param {  
        num_output: 4096  
    }  
}  
[... ReLU, Dropout]  
layer {  
    name: "my-fc8"  
    type: "InnerProduct"  
    inner_product_param {  
        num_output: 10  
    }  
}
```

Caffe Step 2: Define Net (finetuning)

Original prototxt:

```
layer {  
    name: "fc7"  
    type: "InnerProduct"  
    inner_product_param {  
        num_output: 4096  
    }  
}  
[... ReLU, Dropout]  
layer {  
    name: "fc8" ico  
    type: "InnerProduct"  
    inner_product_param {  
        num_output: 1000  
    }  
}
```

Same name:
weights copied

Pretrained weights:

```
"fc7.weight": [values]  
"fc7.bias": [values]  
"fc8.weight": [values] ico  
"fc8.bias": [values]
```

Different name:
weights reinitialized

Modified prototxt:

```
layer {  
    name: "fc7"  
    type: "InnerProduct"  
    inner_product_param {  
        num_output: 4096  
    }  
}  
[... ReLU, Dropout]  
layer {  
    name: "my-fc8" ico  
    type: "InnerProduct"  
    inner_product_param {  
        num_output: 10  
    }  
}
```

Caffe Step 3: Define Solver

Write a prototxt file defining a
[SolverParameter](#)

If finetuning, copy existing
solver.prototxt file

Change net to be your net

Change snapshot_prefix to your
output

Reduce base learning rate (divide
by 100)

Maybe change max_iter and
snapshot

```
1 net: "models/bvlc_alexnet/train_val.prototxt"
2 test_iter: 1000
3 test_interval: 1000
4 base_lr: 0.01
5 lr_policy: "step"
6 gamma: 0.1
7 stepsize: 100000
8 display: 20
9 max_iter: 450000
10 momentum: 0.9
11 weight_decay: 0.0005
12 snapshot: 10000
13 snapshot_prefix: "models/bvlc_alexnet/caffe_alexnet_train"
14 solver_mode: GPU
```

Caffe Step 4: Train!

```
./build/tools/caffe train \
-gpu 0 \
-model path/to/trainval.prototxt \
-solver path/to/solver.prototxt \
-weights path/to/
pretrained_weights.caffemodel
```

<https://github.com/BVLC/caffe/blob/master/tools/caffe.cpp>

Caffe Step 4: Train!

```
./build/tools/caffe train \
-gpu 0 \
-model path/to/trainval.prototxt \
-solver path/to/solver.prototxt \
-weights path/to/
pretrained_weights.caffemodel
```

-gpu -1 for CPU mode

<https://github.com/BVLC/caffe/blob/master/tools/caffe.cpp>

Caffe Step 4: Train!

```
./build/tools/caffe train \
-gpu 0 \
-model path/to/trainval.prototxt \
-solver path/to/solver.prototxt \
-weights path/to/
pretrained_weights.caffemodel
```

-gpu all for multi-GPU data parallelism

<https://github.com/BVLC/caffe/blob/master/tools/caffe.cpp>

Caffe: Model Zoo

AlexNet, VGG,
GoogLeNet, ResNet,
plus others

The screenshot shows a GitHub repository page for 'BVLC / caffe'. The top navigation bar includes links for 'Pull requests', 'Issues', and 'Gist'. Below the bar, there are buttons for 'Watch' (1,207), 'Star' (8,519), and 'Fork' (4,826). The main content area features a section titled 'Model Zoo' with a sub-section for 'Alex Kendall edited this page 13 days ago · 61 revisions'. It contains instructions for acquiring models via scripts or documentation. On the right, a sidebar lists 'Pages' (18) with links to 'Home', 'Caffe on EC2 Ubuntu 14.04 Cuda 7', 'Contributing', 'Development', and 'IDE Nvidia's Eclipse Nsight'. A 'Find a Page...' search bar is also present.

<https://github.com/BVLC/caffe/wiki/Model-Zoo>

Caffe: Python Interface

Not much documentation...

Read the code! Two most important files:

[caffe/python/caffe/_caffe.cpp](#):

Exports Blob, Layer, Net, and Solver classes

[caffe/python/caffe/pycaffe.py](#)

Adds extra methods to Net class

Caffe: Python Interface

Good for:

Interfacing with numpy

Extract features: Run net forward

Compute gradients: Run net backward (DeepDream, etc)

Define layers in Python with numpy (CPU only)

Caffe Pros / Cons

- (+) Good for feedforward networks
- (+) Good for finetuning existing networks
- (+) Train models without writing any code!
- (+) Python interface is pretty useful!
- (-) Need to write C++ / CUDA for new GPU layers
- (-) Not good for recurrent networks
- (-) Cumbersome for big networks (GoogLeNet, ResNet)

Torch

<http://torch.ch>

Torch Overview

From NYU + IDIAP

Written in C and Lua

Used a lot at Facebook, DeepMind

Torch: Lua

High level scripting language, easy to interface with C

Similar to Javascript:

One data structure:

table == JS object

Prototypical inheritance

metatable == JS prototype

First-class functions

Some gotchas:

1-indexed =(

Variables global by default =(

Small standard library

Learn Lua in 15 Minutes

more or less

For a more in-depth Lua tutorial, watch [this video](#) or check out [a transcript of the video](#).

```
-- Two dashes start a one-line comment.
```

```
--[[  
    Adding two '['s and ']'s makes it a  
    multi-line comment.  
--]]
```

```
-- 1. Variables and flow control.
```

```
num = 42 -- All numbers are doubles.  
-- Don't freak out, 64-bit doubles have 52 bits for  
-- storing exact int values; machine precision is  
-- not a problem for ints that need < 52 bits.
```

```
s = 'walternate' -- Immutable strings like Python.  
t = "double-quotes are also fine"  
u = [[ Double brackets  
        start and end  
        multi-line strings.]]  
t = nil -- Undefines t; Lua has garbage collection.
```

```
-- Blocks are denoted with keywords like do/end:  
while num < 50 do  
    num = num + 1 -- No ++ or += type operators.  
end
```

<http://tylerneylon.com/a/learn-lua/>

Torch: Tensors

Torch tensors are just like numpy arrays

Torch: Tensors

Torch tensors are just like numpy arrays

```
1 import numpy as np
2
3 # Simple feedforward network (no biases) in numpy
4
5 # Batch size, input dim, hidden dim, num classes
6 N, D, H, C = 100, 1000, 100, 10
7
8 # First and second layer weights
9 w1 = np.random.randn(D, H)
10 w2 = np.random.randn(H, C)
11
12 # Random input data
13 x = np.random.randn(N, D)
14
15 # Forward pass
16 a = x.dot(w1)          # First layer
17 a = np.maximum(a, 0)    # In-place ReLU
18 scores = a.dot(w2)     # Second layer
19
20 print scores
```

Torch: Tensors

Torch tensors are just like numpy arrays

```
1 import numpy as np
2
3 # Simple feedforward network (no biases) in numpy
4
5 # Batch size, input dim, hidden dim, num classes
6 N, D, H, C = 100, 1000, 100, 10
7
8 # First and second layer weights
9 w1 = np.random.randn(D, H)
10 w2 = np.random.randn(H, C)
11
12 # Random input data
13 x = np.random.randn(N, D)
14
15 # Forward pass
16 a = x.dot(w1)          # First layer
17 a = np.maximum(a, 0)    # In-place ReLU
18 scores = a.dot(w2)     # Second layer
19
20 print scores
```

```
1 require 'torch'
2
3 -- Simple feedforward network (no biases) in torch
4
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7
8 -- First and second layer weights
9 local w1 = torch.randn(D, H)
10 local w2 = torch.randn(H, C)
11
12 -- Random input data
13 local x = torch.randn(N, D)
14
15 -- Forward pass
16 local a = torch.mm(x, w1)           -- First layer
17 a:cmax(0)                          -- In-place ReLU
18 local scores = torch.mm(a, w2)      -- Second layer
19
20 print(scores)
```

Torch: Tensors

Like numpy, can easily change data type:

```
1 import numpy as np
2
3 |
4 # Simple feedforward network (no biases) in numpy
5
6 dtype = np.float32 # Use 32-bit floats
7
8 # Batch size, input dim, hidden dim, num classes
9 N, D, H, C = 100, 1000, 100, 10
10
11 # First and second layer weights
12 w1 = np.random.randn(D, H).astype(dtype)
13 w2 = np.random.randn(H, C).astype(dtype)
14
15 # Random input data
16 x = np.random.randn(N, D).astype(dtype)
17
18 # Forward pass
19 a = x.dot(w1) # First layer
20 a = np.maximum(a, 0) # In-place ReLU
21 scores = a.dot(w2) # Second layer
22
23 print scores
```

```
1 require 'torch'
2
3
4 -- Simple feedforward network (no biases) in torch
5
6 local dtype = 'torch.FloatTensor' -- Use 32-bit floats
7
8 -- Batch size, input dim, hidden dim, num classes
9 local N, D, H, C = 100, 1000, 100, 10
10
11 -- First and second layer weights
12 local w1 = torch.randn(D, H):type(dtype)
13 local w2 = torch.randn(H, C):type(dtype)
14
15 -- Random input data
16 local x = torch.randn(N, D):type(dtype)
17
18 -- Forward pass
19 local a = torch.mm(x, w1) -- First layer
20 a:cmax(0) -- In-place ReLU
21 local scores = torch.mm(a, w2) -- Second layer
22
23 print(scores)
```

Torch: Tensors

Unlike numpy, GPU is just a datatype away:

```
1 import numpy as np
2
3 |
4 # Simple feedforward network (no biases) in numpy
5
6 dtype = np.float32 # Use 32-bit floats
7
8 # Batch size, input dim, hidden dim, num classes
9 N, D, H, C = 100, 1000, 100, 10
10
11 # First and second layer weights
12 w1 = np.random.randn(D, H).astype(dtype)
13 w2 = np.random.randn(H, C).astype(dtype)
14
15 # Random input data
16 x = np.random.randn(N, D).astype(dtype)
17
18 # Forward pass
19 a = x.dot(w1) # First layer
20 a = np.maximum(a, 0) # In-place ReLU
21 scores = a.dot(w2) # Second layer
22
23 print scores
```

```
1 require 'torch'
2 require 'cutorch'
3
4 -- Simple feedforward network (no biases) in torch
5 |
6 local dtype = 'torch.CudaTensor' -- Use CUDA
7
8 -- Batch size, input dim, hidden dim, num classes
9 local N, D, H, C = 100, 1000, 100, 10
10
11 -- First and second layer weights
12 local w1 = torch.randn(D, H):type(dtype)
13 local w2 = torch.randn(H, C):type(dtype)
14
15 -- Random input data
16 local x = torch.randn(N, D):type(dtype)
17
18 -- Forward pass
19 local a = torch.mm(x, w1) -- First layer
20 a:cmax(0) -- In-place ReLU
21 local scores = torch.mm(a, w2) -- Second layer
22
23 print(scores)
```

Torch: Tensors

Documentation on GitHub:

The `Tensor` class is probably the most important class in `Torch`. Almost every package depends on this class. It is *the* class for handling numeric data. As with pretty much anything in `Torch7`, tensors are *serializable*.

Multi-dimensional matrix

A `Tensor` is a potentially multi-dimensional matrix. The number of dimensions is unlimited that can be created using `LongStorage` with more dimensions.

Example:

```
---- creation of a 4D-tensor 4x5x6x2
z = torch.Tensor(4,5,6,2)
--- for more dimensions, (here a 6D tensor) one can do:
s = torch.LongStorage(6)
s[1] = 4; s[2] = 5; s[3] = 6; s[4] = 2; s[5] = 7; s[6] = 3;
x = torch.Tensor(s)
```

<https://github.com/torch/torch7/blob/master/doc/tensor.md>

Torch provides MATLAB-like functions for manipulating `Tensor` objects. Functions fall into several types of categories:

- Constructors like `zeros`, `ones`;
- Extractors like `diag` and `triu`;
- Element-wise mathematical operations like `abs` and `pow`;
- BLAS operations;
- Column or row-wise operations like `sum` and `max`;
- Matrix-wide operations like `trace` and `norm`;
- Convolution and cross-correlation operations like `conv2`;
- Basic linear algebra operations like `eig`;
- Logical operations on `Tensor`s.

By default, all operations allocate a new `Tensor` to return the result. However, all functions also support passing the target `Tensor` (`s`) as the first argument(s), in which case the target `Tensor` (`s`) will be resized accordingly and filled with result. This property is especially useful when one wants have tight control over when memory is allocated.

<https://github.com/torch/torch7/blob/master/doc/math.md>

Torch: nn

nn module lets you easily
build and train neural nets

```
1 require 'torch'
2 require 'nn'
3
4
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16
17 -- Loss functions are called "criterions"
18 local crit = nn.CrossEntropyCriterion() -- Softmax loss
19
20 -- Generate some random input data
21 local x = torch.randn(N, D)
22 local y = torch.Tensor(N):random(C)
23
24 -- Forward pass: Compute scores and loss
25 local scores = net:forward(x)
26 local loss = crit:forward(scores, y)
27
28 -- Backward pass: compute gradients
29 grad_weights:zero()
30 local dscores = crit:backward(scores, y)
31 local dx = net:backward(x, dscores)
32
33 -- Make a gradient step
34 local learning_rate = 1e-3
35 weights:add(-learning_rate, grad_weights)
36 |
```

Torch: nn

nn module lets you easily build
and train neural nets

Build a two-layer ReLU net

```
1 require 'torch'
2 require 'nn'
3
4
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16
17 -- Loss functions are called "criterions"
18 local crit = nn.CrossEntropyCriterion() -- Softmax loss
19
20 -- Generate some random input data
21 local x = torch.randn(N, D)
22 local y = torch.Tensor(N):random(C)
23
24 -- Forward pass: Compute scores and loss
25 local scores = net:forward(x)
26 local loss = crit:forward(scores, y)
27
28 -- Backward pass: compute gradients
29 grad_weights:zero()
30 local dscores = crit:backward(scores, y)
31 local dx = net:backward(x, dscores)
32
33 -- Make a gradient step
34 local learning_rate = 1e-3
35 weights:add(-learning_rate, grad_weights)
36 |
```

Torch: nn

nn module lets you easily build
and train neural nets

Get weights and gradient for
entire network

```
1 require 'torch'
2 require 'nn'
3
4
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16
17 -- Loss functions are called "criterions"
18 local crit = nn.CrossEntropyCriterion() -- Softmax loss
19
20 -- Generate some random input data
21 local x = torch.randn(N, D)
22 local y = torch.Tensor(N):random(C)
23
24 -- Forward pass: Compute scores and loss
25 local scores = net:forward(x)
26 local loss = crit:forward(scores, y)
27
28 -- Backward pass: compute gradients
29 grad_weights:zero()
30 local dscores = crit:backward(scores, y)
31 local dx = net:backward(x, dscores)
32
33 -- Make a gradient step
34 local learning_rate = 1e-3
35 weights:add(-learning_rate, grad_weights)
36 |
```

Torch: nn

nn module lets you easily build
and train neural nets

Use a softmax loss function

```
1 require 'torch'
2 require 'nn'
3
4
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16
17 -- Loss functions are called "criterions"
18 local crit = nn.CrossEntropyCriterion() -- Softmax loss
19
20 -- Generate some random input data
21 local x = torch.randn(N, D)
22 local y = torch.Tensor(N):random(C)
23
24 -- Forward pass: Compute scores and loss
25 local scores = net:forward(x)
26 local loss = crit:forward(scores, y)
27
28 -- Backward pass: compute gradients
29 grad_weights:zero()
30 local dscores = crit:backward(scores, y)
31 local dx = net:backward(x, dscores)
32
33 -- Make a gradient step
34 local learning_rate = 1e-3
35 weights:add(-learning_rate, grad_weights)
36 |
```

Torch: nn

nn module lets you easily build
and train neural nets

Generate random data



```
1 require 'torch'
2 require 'nn'
3
4
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16
17 -- Loss functions are called "criterions"
18 local crit = nn.CrossEntropyCriterion() -- Softmax loss
19
20 -- Generate some random input data
21 local x = torch.randn(N, D)
22 local y = torch.Tensor(N):random(C)
23
24 -- Forward pass: Compute scores and loss
25 local scores = net:forward(x)
26 local loss = crit:forward(scores, y)
27
28 -- Backward pass: compute gradients
29 grad_weights:zero()
30 local dscores = crit:backward(scores, y)
31 local dx = net:backward(x, dscores)
32
33 -- Make a gradient step
34 local learning_rate = 1e-3
35 weights:add(-learning_rate, grad_weights)
36 |
```

Torch: nn

nn module lets you easily build and train neural nets

Forward pass: compute scores and loss



```
1 require 'torch'
2 require 'nn'
3
4
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16
17 -- Loss functions are called "criterions"
18 local crit = nn.CrossEntropyCriterion() -- Softmax loss
19
20 -- Generate some random input data
21 local x = torch.randn(N, D)
22 local y = torch.Tensor(N):random(C)
23
24 -- Forward pass: Compute scores and loss
25 local scores = net:forward(x)
26 local loss = crit:forward(scores, y)
27
28 -- Backward pass: compute gradients
29 grad_weights:zero()
30 local dscores = crit:backward(scores, y)
31 local dx = net:backward(x, dscores)
32
33 -- Make a gradient step
34 local learning_rate = 1e-3
35 weights:add(-learning_rate, grad_weights)
36 |
```

Torch: nn

nn module lets you easily build and train neural nets

Backward pass: Compute gradients. Remember to set weight gradients to zero!

```
1 require 'torch'
2 require 'nn'
3
4
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16
17 -- Loss functions are called "criterions"
18 local crit = nn.CrossEntropyCriterion() -- Softmax loss
19
20 -- Generate some random input data
21 local x = torch.randn(N, D)
22 local y = torch.Tensor(N):random(C)
23
24 -- Forward pass: Compute scores and loss
25 local scores = net:forward(x)
26 local loss = crit:forward(scores, y)
27
28 -- Backward pass: compute gradients
29 grad_weights:zero()
30 local dscores = crit:backward(scores, y)
31 local dx = net:backward(x, dscores)
32
33 -- Make a gradient step
34 local learning_rate = 1e-3
35 weights:add(-learning_rate, grad_weights)
36 |
```

Torch: nn

nn module lets you easily build and train neural nets

Update: Make a gradient descent step

```
1 require 'torch'
2 require 'nn'
3
4
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16
17 -- Loss functions are called "criterions"
18 local crit = nn.CrossEntropyCriterion() -- Softmax loss
19
20 -- Generate some random input data
21 local x = torch.randn(N, D)
22 local y = torch.Tensor(N):random(C)
23
24 -- Forward pass: Compute scores and loss
25 local scores = net:forward(x)
26 local loss = crit:forward(scores, y)
27
28 -- Backward pass: compute gradients
29 grad_weights:zero()
30 local dscores = crit:backward(scores, y)
31 local dx = net:backward(x, dscores)
32
33 -- Make a gradient step
34 local learning_rate = 1e-3
35 weights:add(-learning_rate, grad_weights)
36 !
```

Torch: cunn

Running on GPU is easy:

```
1 require 'torch'
2 require 'cutorch'
3 require 'nn'
4 require 'cunn'
5
6 -- Batch size, input dim, hidden dim, num classes
7 local N, D, H, C = 100, 1000, 100, 10
8
9 local dtype = 'torch.CudaTensor'
10
11 -- Build a one-layer ReLU network
12 local net = nn.Sequential()
13 net:add(nn.Linear(D, H))
14 net:add(nn.ReLU())
15 net:add(nn.Linear(H, C))
16 net:type(dtype)
17
18 -- Collect all weights and gradients in a single Tensor
19 local weights, grad_weights = net:getParameters()
20
21 -- Loss functions are called "criterions"
22 local crit = nn.CrossEntropyCriterion() -- Softmax loss
23 crit:type(dtype)
24
25 -- Generate some random input data
26 local x = torch.randn(N, D):type(dtype)
27 local y = torch.Tensor(N):random(C):type(dtype)
28
29 -- Forward pass: Compute scores and loss
30 local scores = net:forward(x)
31 local loss = crit:forward(scores, y)
32
33 -- Backward pass: compute gradients
34 grad_weights:zero()
35 local dscores = crit:backward(scores, y)
36 local dx = net:backward(x, dscores)
37
38 -- Make a gradient step
39 local learning_rate = 1e-3
40 weights:add(-learning_rate, grad_weights)
```

Torch: cunn

Running on GPU is easy:

Import a few new packages

```
1 require 'torch'
2 require 'cutorch'
3 require 'nn'
4 require 'cunn'
5
6 -- Batch size, input dim, hidden dim, num classes
7 local N, D, H, C = 100, 1000, 100, 10
8
9 local dtype = 'torch.CudaTensor'
10
11 -- Build a one-layer ReLU network
12 local net = nn.Sequential()
13 net:add(nn.Linear(D, H))
14 net:add(nn.ReLU())
15 net:add(nn.Linear(H, C))
16 net:type(dtype)
17
18 -- Collect all weights and gradients in a single Tensor
19 local weights, grad_weights = net:getParameters()
20
21 -- Loss functions are called "criterions"
22 local crit = nn.CrossEntropyCriterion() -- Softmax loss
23 crit:type(dtype)
24
25 -- Generate some random input data
26 local x = torch.randn(N, D):type(dtype)
27 local y = torch.Tensor(N):random(C):type(dtype)
28
29 -- Forward pass: Compute scores and loss
30 local scores = net:forward(x)
31 local loss = crit:forward(scores, y)
32
33 -- Backward pass: compute gradients
34 grad_weights:zero()
35 local dscores = crit:backward(scores, y)
36 local dx = net:backward(x, dscores)
37
38 -- Make a gradient step
39 local learning_rate = 1e-3
40 weights:add(-learning_rate, grad_weights)
```

Torch: cunn

Running on GPU is easy:

Import a few new packages

Cast network and criterion

```
1 require 'torch'
2 require 'cutorch'
3 require 'nn'
4 require 'cunn'
5
6 -- Batch size, input dim, hidden dim, num classes
7 local N, D, H, C = 100, 1000, 100, 10
8
9 local dtype = 'torch.CudaTensor'
10
11 -- Build a one-layer ReLU network
12 local net = nn.Sequential()
13 net:add(nn.Linear(D, H))
14 net:add(nn.ReLU())
15 net:add(nn.Linear(H, C))
16
17 net:type(dtype)
18
19 -- Collect all weights and gradients in a single Tensor
20 local weights, grad_weights = net:getParameters()
21
22 -- Loss functions are called "criterions"
23 local crit = nn.CrossEntropyCriterion() -- Softmax loss
24
25 crit:type(dtype)
26
27 -- Generate some random input data
28 local x = torch.randn(N, D):type(dtype)
29 local y = torch.Tensor(N):random(C):type(dtype)
30
31 -- Forward pass: Compute scores and loss
32 local scores = net:forward(x)
33 local loss = crit:forward(scores, y)
34
35 -- Backward pass: compute gradients
36 grad_weights:zero()
37 local dscores = crit:backward(scores, y)
38 local dx = net:backward(x, dscores)
39
40 -- Make a gradient step
41 local learning_rate = 1e-3
42 weights:add(-learning_rate, grad_weights)
```

Torch: cunn

Running on GPU is easy:

Import a few new packages

Cast network and criterion

Cast data and labels

```
1 require 'torch'
2 require 'cutorch'
3 require 'nn'
4 require 'cunn'
5
6 -- Batch size, input dim, hidden dim, num classes
7 local N, D, H, C = 100, 1000, 100, 10
8
9 local dtype = 'torch.CudaTensor'
10
11 -- Build a one-layer ReLU network
12 local net = nn.Sequential()
13 net:add(nn.Linear(D, H))
14 net:add(nn.ReLU())
15 net:add(nn.Linear(H, C))
16 net:type(dtype)
17
18 -- Collect all weights and gradients in a single Tensor
19 local weights, grad_weights = net:getParameters()
20
21 -- Loss functions are called "criterions"
22 local crit = nn.CrossEntropyCriterion() -- Softmax loss
23 crit:type(dtype)
24
25 -- Generate some random input data
26 local x = torch.randn(N, D):type(dtype)
27 local y = torch.FloatTensor(N):random(C):type(dtype)
28
29 -- Forward pass: Compute scores and loss
30 local scores = net:forward(x)
31 local loss = crit:forward(scores, y)
32
33 -- Backward pass: compute gradients
34 grad_weights:zero()
35 local dscores = crit:backward(scores, y)
36 local dx = net:backward(x, dscores)
37
38 -- Make a gradient step
39 local learning_rate = 1e-3
40 weights:add(-learning_rate, grad_weights)
```

Torch: optim

optim package implements different update rules: momentum, Adam, etc

```
1 require 'torch'
2 require 'nn'
3 require 'optim'
4
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16
17 -- Loss functions are called "criterions"
18 local crit = nn.CrossEntropyCriterion() -- Softmax loss
19
20 -- Callback to interface with optim methods
21 local function f(w)
22     assert(w == weights)
23
24     -- Generate some random input data
25     local x = torch.randn(N, D)
26     local y = torch.Tensor(N):random(C)
27
28     -- Forward pass: Compute scores and loss
29     local scores = net:forward(x)
30     local loss = crit:forward(scores, y)
31
32     -- Backward pass: compute gradients
33     grad_weights:zero()
34     local dscores = crit:backward(scores, y)
35     local dx = net:backward(x, dscores)
36
37     return loss, grad_weights
38 end
39
40 -- Make a step using Adam
41 local state = {learningRate=1e-3}
42 optim.adam(f, weights, state)
```

Torch: optim

optim package implements different update rules: momentum, Adam, etc

Import optim package

```
1 require 'torch'
2 require 'nn'
3 require 'optim'
4
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16
17 -- Loss functions are called "criterions"
18 local crit = nn.CrossEntropyCriterion() -- Softmax loss
19
20 -- Callback to interface with optim methods
21 local function f(w)
22     assert(w == weights)
23
24     -- Generate some random input data
25     local x = torch.randn(N, D)
26     local y = torch.Tensor(N):random(C)
27
28     -- Forward pass: Compute scores and loss
29     local scores = net:forward(x)
30     local loss = crit:forward(scores, y)
31
32     -- Backward pass: compute gradients
33     grad_weights:zero()
34     local dscores = crit:backward(scores, y)
35     local dx = net:backward(x, dscores)
36
37     return loss, grad_weights
38 end
39
40 -- Make a step using Adam
41 local state = {learningRate=1e-3}
42 optim.adam(f, weights, state)
```

Torch: optim

optim package implements different update rules: momentum, Adam, etc

Import optim package

Write a callback function that returns loss and gradients

```
1 require 'torch'
2 require 'nn'
3 require 'optim'
4
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16
17 -- Loss functions are called "criterions"
18 local crit = nn.CrossEntropyCriterion() -- Softmax loss
19
20 -- Callback to interface with optim methods
21 local function f(w)
22     assert(w == weights)
23
24     -- Generate some random input data
25     local x = torch.randn(N, D)
26     local y = torch.Tensor(N):random(C)
27
28     -- Forward pass: Compute scores and loss
29     local scores = net:forward(x)
30     local loss = crit:forward(scores, y)
31
32     -- Backward pass: compute gradients
33     grad_weights:zero()
34     local dscores = crit:backward(scores, y)
35     local dx = net:backward(x, dscores)
36
37     return loss, grad_weights
38 end
39
40 -- Make a step using Adam
41 local state = {learningRate=1e-3}
42 optim.adam(f, weights, state)
```

Torch: optim

optim package implements different update rules: momentum, Adam, etc

Import optim package

Write a callback function that returns loss and gradients

state variable holds hyperparameters, cached values, etc; pass it to adam

```
1 require 'torch'
2 require 'nn'
3 require 'optim'
4
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16
17 -- Loss functions are called "criterions"
18 local crit = nn.CrossEntropyCriterion() -- Softmax loss
19
20 -- Callback to interface with optim methods
21 local function f(w)
22     assert(w == weights)
23
24     -- Generate some random input data
25     local x = torch.randn(N, D)
26     local y = torch.Tensor(N):random(C)
27
28     -- Forward pass: Compute scores and loss
29     local scores = net:forward(x)
30     local loss = crit:forward(scores, y)
31
32     -- Backward pass: compute gradients
33     grad_weights:zero()
34     local dscores = crit:backward(scores, y)
35     local dx = net:backward(x, dscores)
36
37     return loss, grad_weights
38 end
39
40 -- Make a step using Adam
41 local state = {learningRate=1e-3}
42 optim.adam[f, weights, state]
```

Torch: Modules

Caffe has Nets and Layers;
Torch just has Modules

Torch: Modules

Caffe has Nets and Layers;
Torch just has Modules

Modules are classes written in
Lua; easy to read and write

Forward / backward written in Lua
using Tensor methods

Same code runs on CPU / GPU

```
1 local Linear, parent = torch.class('nn.Linear', 'nn.Module')
2
3 function Linear:__init__(inputSize, outputSize, bias)
4     parent.__init__(self)
5     local bias = ((bias == nil) and true) or bias
6     self.weight = torch.Tensor(outputSize, inputSize)
7     self.gradWeight = torch.Tensor(outputSize, inputSize)
8     if bias then
9         self.bias = torch.Tensor(outputSize)
10        self.gradBias = torch.Tensor(outputSize)
11    end
12    self:reset()
13
14
```

<https://github.com/torch/nn/blob/master/Linear.lua>

Torch: Modules

Caffe has Nets and Layers;
Torch just has Modules

Modules are classes written in
Lua; easy to read and write

updateOutput: Forward pass;
compute output

```
1 | local Linear, parent = torch.class('nn.Linear', 'nn.Module')
2 |
3 |
4 | function Linear:updateOutput(input)
5 |   if input:dim() == 1 then
6 |     self.output:resize(self.weight:size(1))
7 |     if self.bias then self.output:copy(self.bias) else self.output:zero() end
8 |     self.output:addmv(1, self.weight, input)
9 |   elseif input:dim() == 2 then
10 |     local nframe = input:size(1)
11 |     local nElement = self.output:nElement()
12 |     self.output:resize(nframe, self.weight:size(1))
13 |     if self.output:nElement() ~= nElement then
14 |       self.output:zero()
15 |     end
16 |     self.addBuffer = self.addBuffer or input.new()
17 |     if self.addBuffer:nElement() ~= nframe then
18 |       self.addBuffer:resize(nframe):fill(1)
19 |     end
20 |     self.output:addmm(0, self.output, 1, input, self.weight:t())
21 |     if self.bias then self.output:addr(1, self.addBuffer, self.bias) end
22 |   else
23 |     error('input must be vector or matrix')
24 |   end
25 |
26 |   return self.output
27 | end
```

<https://github.com/torch/nn/blob/master/Linear.lua>

Torch: Modules

Caffe has Nets and Layers;
Torch just has Modules

Modules are classes written in
Lua; easy to read and write

updateGradInput: Backward;
compute gradient of input

```
1 | local Linear, parent = torch.class('nn.Linear', 'nn.Module')
2 |
64 | function Linear:updateGradInput(input, gradOutput)
65 |   if self.gradInput then
66 |
67 |     local nElement = self.gradInput:nElement()
68 |     self.gradInput:resizeAs(input)
69 |     if self.gradInput:nElement() ~= nElement then
70 |       self.gradInput:zero()
71 |     end
72 |     if input:dim() == 1 then
73 |       self.gradInput:addmv(0, 1, self.weight:t(), gradOutput)
74 |     elseif input:dim() == 2 then
75 |       self.gradInput:addmm(0, 1, gradOutput, self.weight)
76 |     end
77 |
78 |     return self.gradInput
79 |   end
80 | end
```

<https://github.com/torch/nn/blob/master/Linear.lua>

Torch: Modules

Caffe has Nets and Layers;
Torch just has Modules

Modules are classes written in
Lua; easy to read and write

accGradParameters: Backward;
compute gradient of weights

```
1 | local Linear, parent = torch.class('nn.Linear', 'nn.Module')
2 |
3 |
4 |
5 |
6 |
7 |
82 function Linear:accGradParameters(input, gradOutput, scale)
83     scale = scale or 1
84     if input:dim() == 1 then
85         self.gradWeight:addr(scale, gradOutput, input)
86         if self.bias then self.gradBias:add(scale, gradOutput) end
87     elseif input:dim() == 2 then
88         self.gradWeight:addmm(scale, gradOutput:t(), input)
89         if self.bias then
90             self.gradBias:addmv(scale, gradOutput:t(), self.addBuffer)
91         end
92     end
93 end
```

<https://github.com/torch/nn/blob/master/Linear.lua>

Torch: Modules

Tons of built-in modules and loss functions

Abs.lua	TemporalConvolution.lua
AbsCriterion.lua	TemporalMaxPooling.lua
Add.lua	TemporalSubSampling.lua
AddConstant.lua	Threshold.lua
BCECriterion.lua	Transpose.lua
BatchNormalization.lua	View.lua
Bilinear.lua	VolumetricAveragePooling.lua
CAddTable.lua	VolumetricConvolution.lua
CDivTable.lua	VolumetricDropout.lua
CMakeLists.txt	VolumetricFullConvolution.lua
CMul.lua	VolumetricMaxPooling.lua
CMulTable.lua	VolumetricMaxUnpooling.lua

<https://github.com/torch/nn>

MarginCriterion.lua
MarginRankingCriterion.lua
Max.lua
Mean.lua
Min.lua
MixtureTable.lua
Module.lua
Mul.lua
MulConstant.lua
MultiCriterion.lua
MultiLabelMarginCriterion.lua
MultiLabelSoftMarginCriterion.lua
MultiMarginCriterion.lua
Narrow.lua

SparseLinear.lua
SpatialAdaptiveMaxPooling.lua
SpatialAveragePooling.lua
SpatialBatchNormalization.lua
SpatialContrastiveNormalization.lua
SpatialConvolution.lua
SpatialConvolutionLocal.lua
SpatialConvolutionMM.lua
SpatialConvolutionMap.lua
SpatialCrossMapLRN.lua
SpatialDivisiveNormalization.lua
SpatialDropout.lua
SpatialFractionalMaxPooling.lua
SpatialFullConvolution.lua
SpatialFullConvolutionMap.lua
SpatialLPPooling.lua
SpatialMaxPooling.lua
SpatialMaxUnpooling.lua
ClassSimplexCriterion.lua
Concat.lua
ConcatTable.lua
Container.lua
Contiguous.lua
Copy.lua
Cosine.lua
CosineDistance.lua
CosineEmbeddingCriterion.lua
Criterion.lua
CriterionTable.lua
CrossEntropyCriterion.lua
DepthConcat.lua
DistKLDivCriterion.lua
DotProduct.lua
Dropout.lua
ELU.lua

Torch: Modules

Tons of built-in modules and loss functions
New ones all the time:

Abs.lua	TemporalConvolution.lua
AbsCriterion.lua	TemporalMaxPooling.lua
Add.lua	TemporalSubSampling.lua
AddConstant.lua	Threshold.lua
BCECriterion.lua	Transpose.lua
BatchNormalization.lua	View.lua
Bilinear.lua	VolumetricAveragePooling.lua
CAddTable.lua	VolumetricConvolution.lua
CDivTable.lua	VolumetricDropout.lua
CMakeLists.txt	VolumetricFullConvolution.lua
CMul.lua	VolumetricMaxPooling.lua
CMulTable.lua	VolumetricMaxUnpooling.lua

<https://github.com/torch/nn>

MarginCriterion.lua
MarginRankingCriterion.lua
Max.lua
Mean.lua
Min.lua
MixtureTable.lua
Module.lua
Mul.lua
MulConstant.lua
MultiCriterion.lua
MultiLabelMarginCriterion.lua
MultiLabelSoftMarginCriterion.lua
MultiMarginCriterion.lua
Narrow.lua

Added 2/19/2016
Added 2/16/2016

SparseLinear.lua
SpatialAdaptiveMaxPooling.lua
SpatialAveragePooling.lua
SpatialBatchNormalization.lua
SpatialContrastiveNormalization.lua
SpatialConvolution.lua
SpatialConvolutionLocal.lua
SpatialConvolutionMM.lua
SpatialConvolutionMap.lua
SpatialCrossMapLRN.lua
SpatialDivisiveNormalization.lua
SpatialDropout.lua
SpatialFractionalMaxPooling.lua
SpatialFullConvolution.lua
SpatialFullConvolutionMap.lua
SpatialLPPooling.lua
SpatialMaxPooling.lua
SpatialMaxUnpooling.lua
ClassSimplexCriterion.lua
Concat.lua
ConcatTable.lua
Container.lua
Contiguous.lua
Copy.lua
Cosine.lua
CosineDistance.lua
CosineEmbeddingCriterion.lua
Criterion.lua
CriterionTable.lua
CrossEntropyCriterion.lua
DepthConcat.lua
DistKLDivCriterion.lua
DotProduct.lua
Dropout.lua
ELU.lua

Torch: Modules

Writing your own modules is easy!

TimesTwo.lua

```
1 require 'nn'  
2  
3 local times_two, parent = torch.class('nn.TimesTwo', 'nn.Module')  
4  
5 function times_two:_init()  
6     parent._init(self)  
7 end  
8  
9  
10 function times_two:updateOutput(input)  
11     self.output:mul(input, 2)  
12     return self.output  
13 end  
14  
15  
16 function times_two:updateGradInput(input, gradOutput)  
17     self.gradInput:mul(gradOutput, 2)  
18     return self.gradInput  
19 end  
20
```

times_two_example.lua

```
1 require 'nn'  
2  
3 require 'TimesTwo'  
4  
5 local times_two = nn.TimesTwo()  
6  
7 local input = torch.randn(4, 5)  
8 local output = times_two:forward(input)  
9  
10 print('here is input:')  
11 print(input)  
12  
13 print('here is output:')  
14 print(output)  
15  
16 local gradOutput = torch.randn(4, 5)  
17 local gradInput = times_two:backward(input, gradOutput)  
18  
19 print('here is gradOutput:')  
20 print(gradOutput)  
21  
22 print('here is gradInput')  
23 print(gradInput)
```

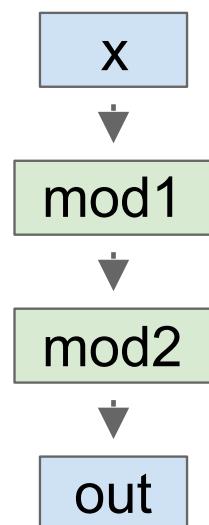
Torch: Modules

Container modules allow you to combine multiple modules

Torch: Modules

Container modules allow you to combine multiple modules

```
local seq = nn.Sequential()  
seq:add(mod1)  
seq:add(mod2)  
local out = seq:forward(x)
```

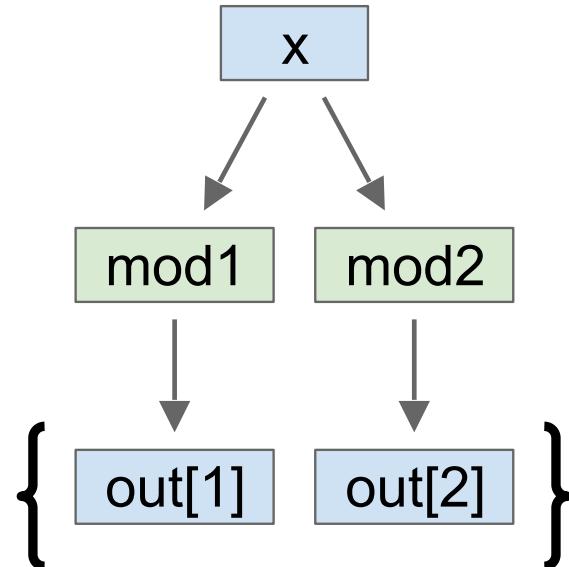
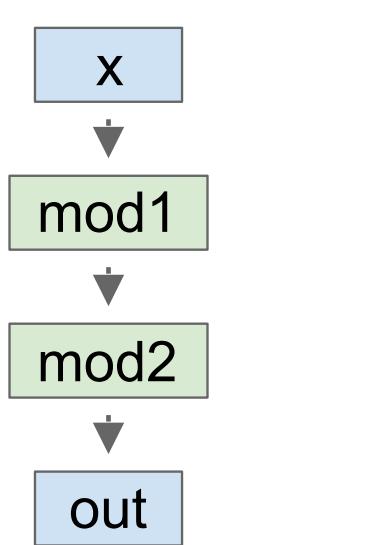


Torch: Modules

Container modules allow you to combine multiple modules

```
local seq = nn.Sequential()  
seq:add(mod1)  
seq:add(mod2)  
local out = seq:forward(x)
```

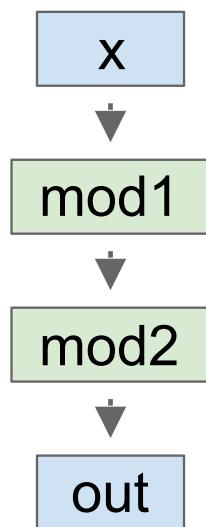
```
local concat = nn.ConcatTable()  
concat:add(mod1)  
concat:add(mod2)  
local out = concat:forward(x)
```



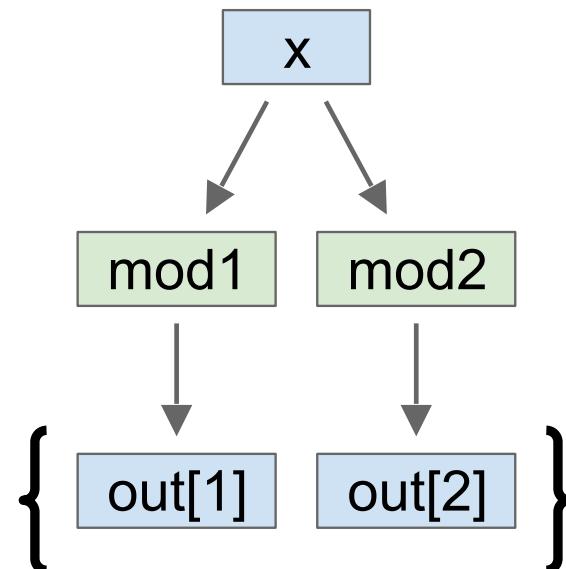
Torch: Modules

Container modules allow you to combine multiple modules

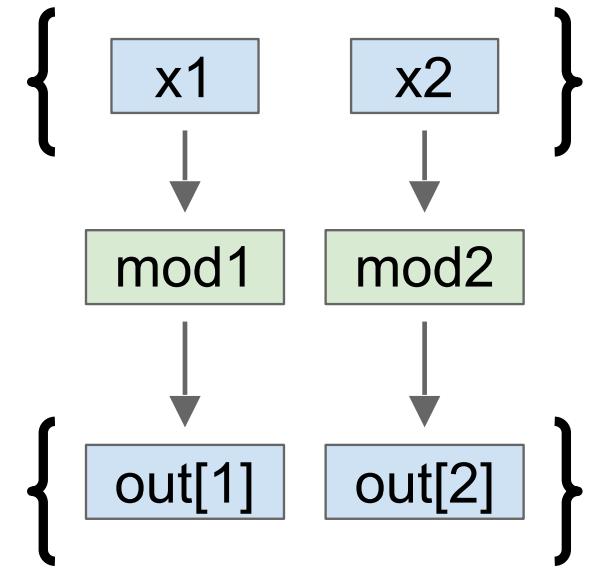
```
local seq = nn.Sequential()  
seq:add(mod1)  
seq:add(mod2)  
local out = seq:forward(x)
```



```
local concat = nn.ConcatTable()  
concat:add(mod1)  
concat:add(mod2)  
local out = concat:forward(x)
```



```
local parallel = nn.ParallelTable()  
parallel:add(mod1)  
parallel:add(mod2)  
local out = parallel:forward({x1, x2})
```



Torch: nngraph

Use nngraph to build modules
that combine their inputs in
complex ways

Inputs: x, y, z

Outputs: c

$$a = x + y$$

$$b = a \odot z$$

$$c = a + b$$

Torch: nngraph

Use nngraph to build modules that combine their inputs in complex ways

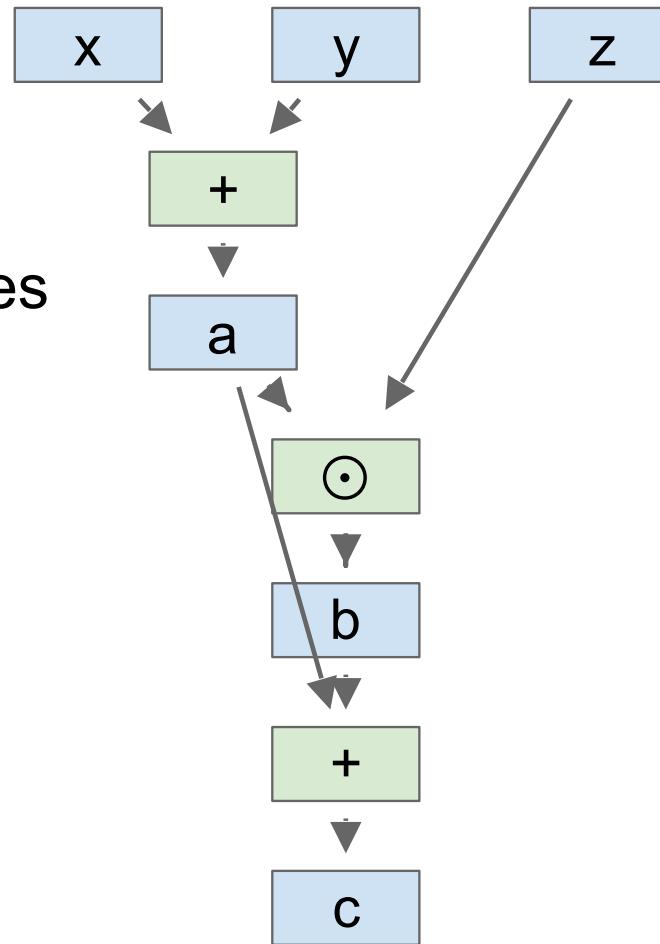
Inputs: x, y, z

Outputs: c

$$a = x + y$$

$$b = a \odot z$$

$$c = a + b$$



Torch: nngraph

Use nngraph to build modules that combine their inputs in complex ways

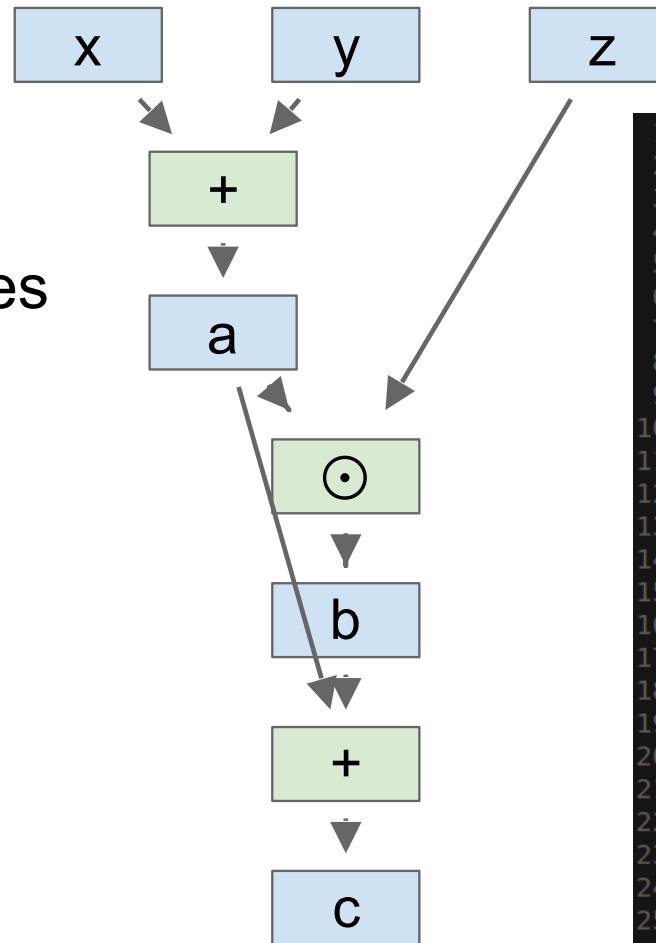
Inputs: x, y, z

Outputs: c

$$a = x + y$$

$$b = a \odot z$$

$$c = a + b$$



```
1 require 'torch'
2 require 'nn'
3 require 'nngraph'
4
5 local function build_module()
6   local x = nn.Identity(){}
7   local y = nn.Identity(){}
8   local z = nn.Identity(){}
9
10  local a = nn.CAddTable()({x, y})
11  local b = nn.CMulTable()({a, z})
12  local c = nn.CAddTable()({a, b})
13
14  local inputs = {x, y, z}
15  local outputs = {c}
16  return nn.gModule(inputs, outputs)
17 end
18
19 local mod = build_module()
20
21 local x = torch.randn(4, 5)
22 local y = torch.randn(4, 5)
23 local z = torch.randn(4, 5)
24
25 local c = mod:forward({x, y, z})
```

Torch: Pretrained Models

loadcaffe: Load pretrained Caffe models: AlexNet, VGG, some others

<https://github.com/szagoruyko/loadcaffe>

GoogLeNet v1: <https://github.com/soumith/inception.torch>

GoogLeNet v3: <https://github.com/Moodstocks/inception-v3.torch>

ResNet: <https://github.com/facebook/fb.resnet.torch>

Torch: Package Management

After installing torch, use luarocks
to install or update Lua packages

(Similar to pip install from Python)

```
luarocks install torch
luarocks install nn
luarocks install optim
luarocks install lua-cjson
```

Torch: Other useful packages

torch.cudnn: Bindings for NVIDIA cuDNN kernels

<https://github.com/soumith/cudnn.torch>

torch-hdf5: Read and write HDF5 files from Torch

<https://github.com/deepmind/torch-hdf5>

lua-cjson: Read and write JSON files from Lua

<https://luarocks.org/modules/luarocks/lua-cjson>

cltorch, clnn: OpenCL backend for Torch, and port of nn

<https://github.com/hughperkins/cltorch>, <https://github.com/hughperkins/clnn>

torch-autograd: Automatic differentiation; sort of like more powerful nngraph, similar to Theano or TensorFlow

<https://github.com/twitter/torch-autograd>

fbcunn: Facebook: FFT conv, multi-GPU (DataParallel, ModelParallel)

<https://github.com/facebook/fbcunn>

Torch: Typical Workflow

Step 1: Preprocess data; usually use a Python script to dump data to HDF5

Step 2: Train a model in Lua / Torch; read from HDF5 datafile, save trained model to disk

Step 3: Use trained model for something, often with an evaluation script

Torch: Typical Workflow

Example: <https://github.com/jcjohnson/torch-rnn>

Step 1: Preprocess data; usually use a Python script to dump data to HDF5 (<https://github.com/jcjohnson/torch-rnn/blob/master/scripts/preprocess.py>)

Step 2: Train a model in Lua / Torch; read from HDF5 datafile, save trained model to disk (<https://github.com/jcjohnson/torch-rnn/blob/master/train.lua>)

Step 3: Use trained model for something, often with an evaluation script (<https://github.com/jcjohnson/torch-rnn/blob/master/sample.lua>)

Torch: Pros / Cons

- (-) Lua
- (-) Less plug-and-play than Caffe
 - You usually write your own training code
- (+) Lots of modular pieces that are easy to combine
- (+) Easy to write your own layer types and run on GPU
- (+) Most of the library code is in Lua, easy to read
- (+) Lots of pretrained models!
- (-) Not great for RNNs

Theano

<http://deeplearning.net/software/theano/>

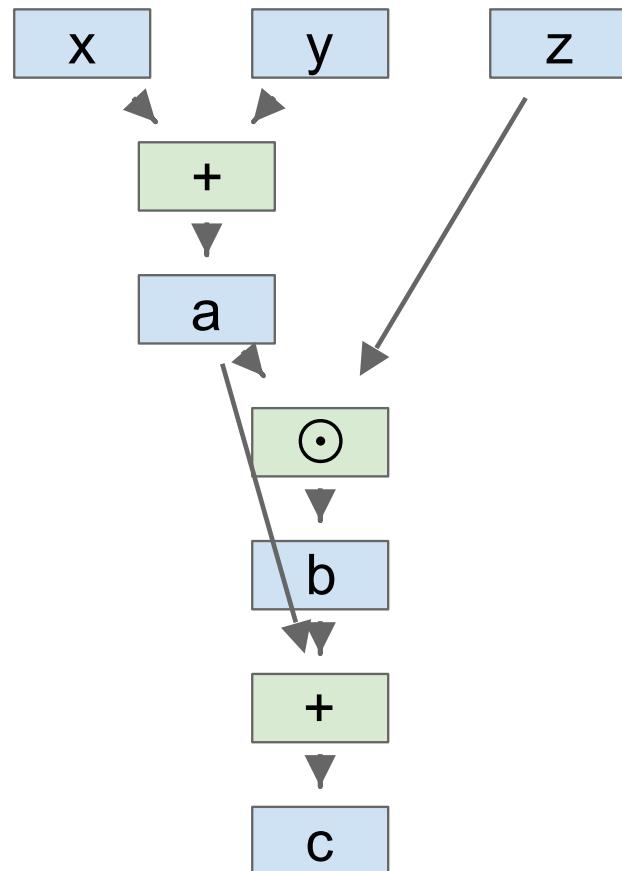
Theano Overview

From Yoshua Bengio's group at University of Montreal

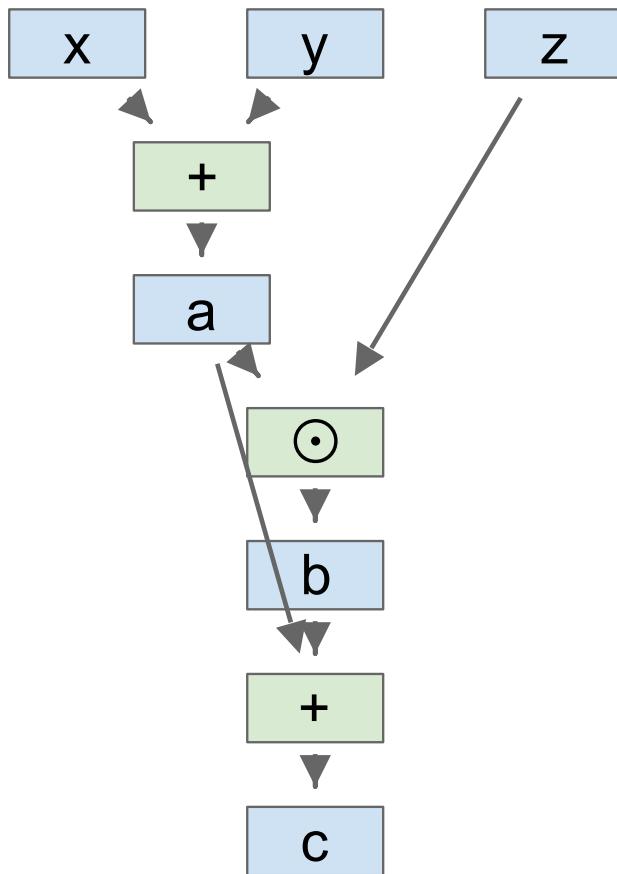
Embracing computation graphs, symbolic computation

High-level wrappers: Keras, Lasagne

Theano: Computational Graphs



Theano: Computational Graphs



```
import theano
import theano.tensor as T

# Define symbolic variables
x = T.matrix('x')
y = T.matrix('y')
z = T.matrix('z')

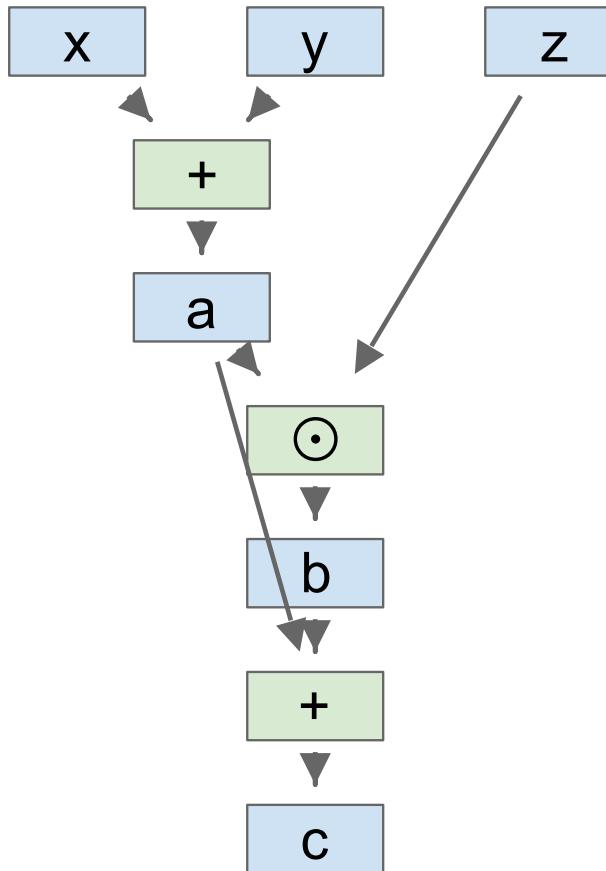
# Compute some other values symbolically
a = x + y
b = a * z
c = a + b

# Compile a function that computes c
f = theano.function(
    inputs=[x, y, z],
    outputs=c
)

# Evaluate the compiled function
# on some real values
xx = np.random.randn(4, 5)
yy = np.random.randn(4, 5)
zz = np.random.randn(4, 5)
print f(xx, yy, zz)

# Repeat the same computation
# explicitly using numpy ops
aa = xx + yy
bb = aa * zz
cc = aa + bb
print cc
```

Theano: Computational Graphs



```
import theano
import theano.tensor as T

# Define symbolic variables
x = T.matrix('x')
y = T.matrix('y')
z = T.matrix('z')

# Compute some other values symbolically
a = x + y
b = a * z
c = a + b

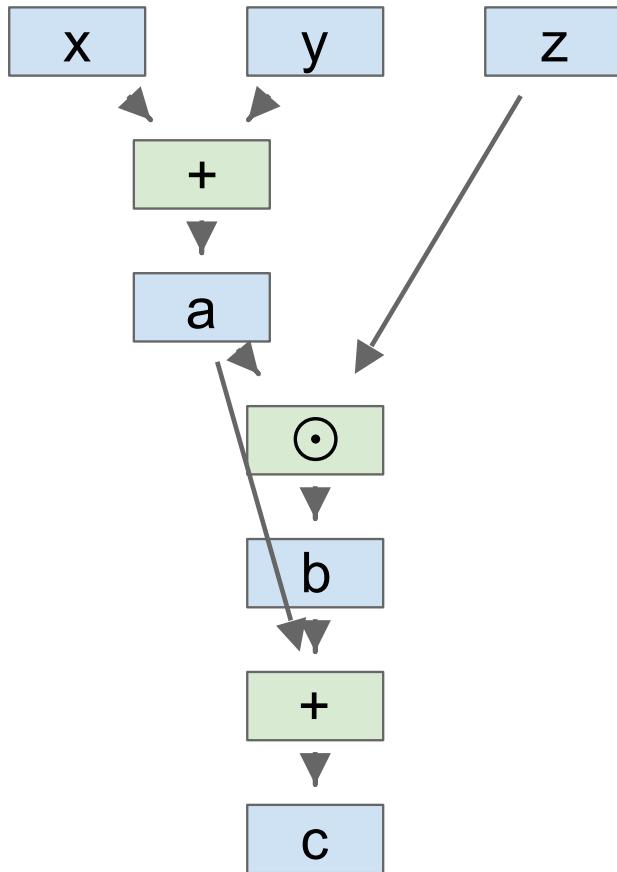
# Compile a function that computes c
f = theano.function(
    inputs=[x, y, z],
    outputs=c
)

# Evaluate the compiled function
# on some real values
xx = np.random.randn(4, 5)
yy = np.random.randn(4, 5)
zz = np.random.randn(4, 5)
print f(xx, yy, zz)

# Repeat the same computation
# explicitly using numpy ops
aa = xx + yy
bb = aa * zz
cc = aa + bb
print cc
```

Define symbolic variables;
these are inputs to the
graph

Theano: Computational Graphs



```
import theano
import theano.tensor as T

# Define symbolic variables
x = T.matrix('x')
y = T.matrix('y')
z = T.matrix('z')

# Compute some other values symbolically
a = x + y
b = a * z
c = a + b

# Compile a function that computes c
f = theano.function(
    inputs=[x, y, z],
    outputs=c
)

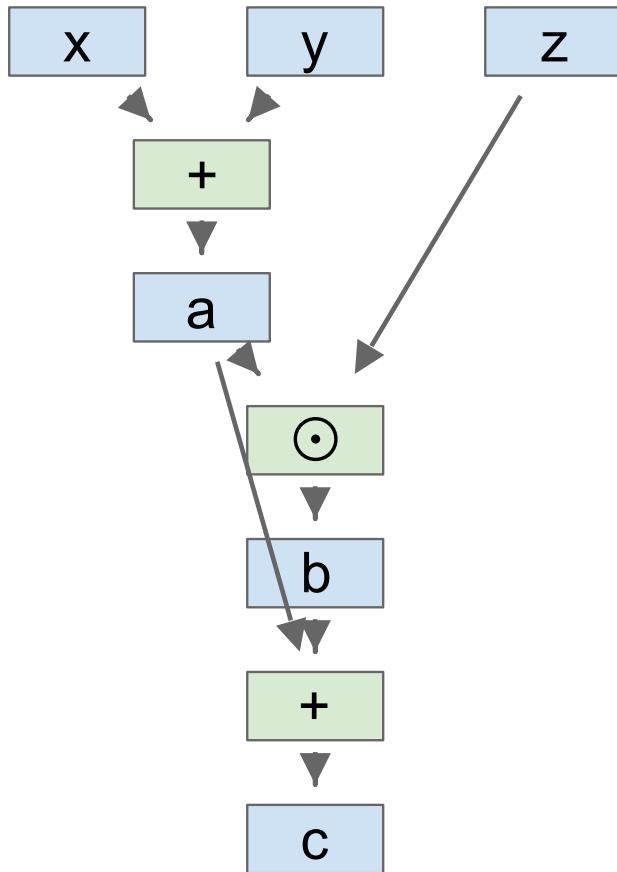
# Evaluate the compiled function
# on some real values
xx = np.random.randn(4, 5)
yy = np.random.randn(4, 5)
zz = np.random.randn(4, 5)
print f(xx, yy, zz)

# Repeat the same computation
# explicitly using numpy ops
aa = xx + yy
bb = aa * zz
cc = aa + bb
print cc
```

Compute intermediates
and outputs symbolically



Theano: Computational Graphs



```
import theano
import theano.tensor as T

# Define symbolic variables
x = T.matrix('x')
y = T.matrix('y')
z = T.matrix('z')

# Compute some other values symbolically
a = x + y
b = a * z
c = a + b

# Compile a function that computes c
f = theano.function(
    inputs=[x, y, z],
    outputs=c
)

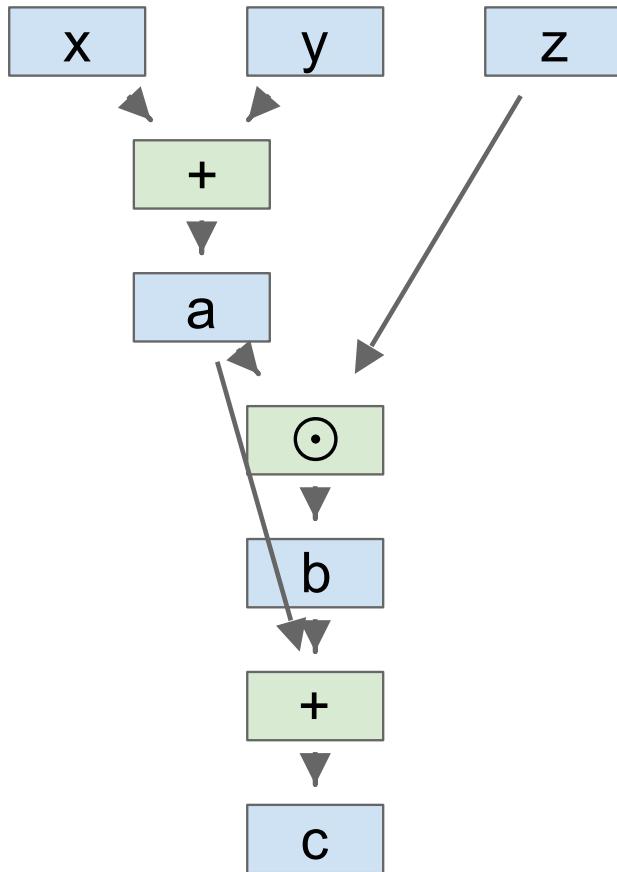
# Evaluate the compiled function
# on some real values
xx = np.random.randn(4, 5)
yy = np.random.randn(4, 5)
zz = np.random.randn(4, 5)
print f(xx, yy, zz)

# Repeat the same computation
# explicitly using numpy ops
aa = xx + yy
bb = aa * zz
cc = aa + bb
print cc
```

Compile a function that produces c from x, y, z (generates code)



Theano: Computational Graphs



```
import theano
import theano.tensor as T

# Define symbolic variables
x = T.matrix('x')
y = T.matrix('y')
z = T.matrix('z')

# Compute some other values symbolically
a = x + y
b = a * z
c = a + b

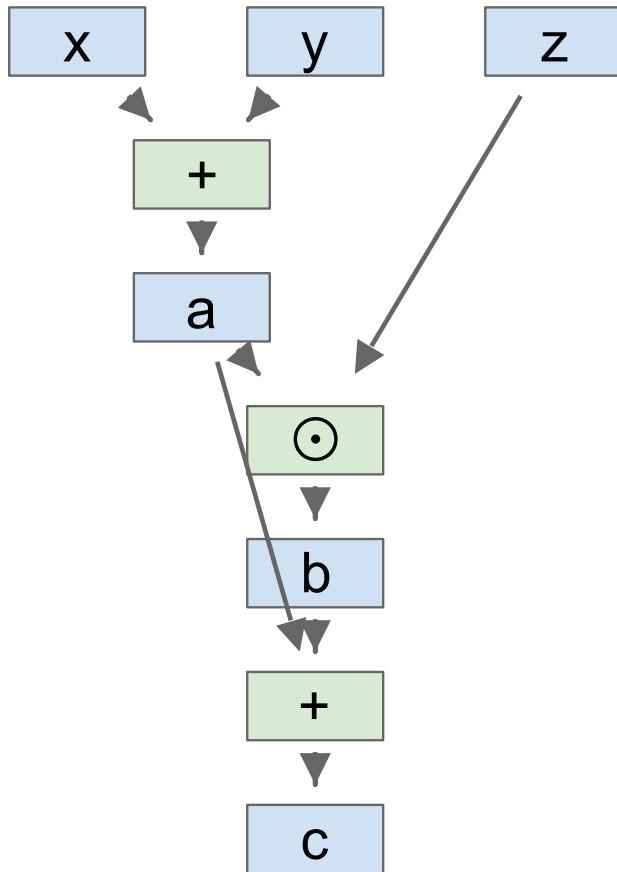
# Compile a function that computes c
f = theano.function(
    inputs=[x, y, z],
    outputs=c
)

# Evaluate the compiled function
# on some real values
xx = np.random.randn(4, 5)
yy = np.random.randn(4, 5)
zz = np.random.randn(4, 5)
print f(xx, yy, zz)

# Repeat the same computation
# explicitly using numpy ops
aa = xx + yy
bb = aa * zz
cc = aa + bb
print cc
```

Run the function, passing
some numpy arrays
(may run on GPU)

Theano: Computational Graphs



```
import theano
import theano.tensor as T

# Define symbolic variables
x = T.matrix('x')
y = T.matrix('y')
z = T.matrix('z')

# Compute some other values symbolically
a = x + y
b = a * z
c = a + b

# Compile a function that computes c
f = theano.function(
    inputs=[x, y, z],
    outputs=c
)

# Evaluate the compiled function
# on some real values
xx = np.random.randn(4, 5)
yy = np.random.randn(4, 5)
zz = np.random.randn(4, 5)
print f(xx, yy, zz)

# Repeat the same computation
# explicitly using numpy ops
aa = xx + yy
bb = aa * zz
cc = aa + bb
print cc
```

Repeat the same computation using numpy operations (runs on CPU)

Theano: Simple Neural Net

```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Compile a function to compute loss, scores
f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores],
)

# Run the function
xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N)
ww1 = 1e-3 * np.random.randn(D, H)
ww2 = 1e-3 * np.random.randn(H, C)

loss, scores = f(xx, yy, ww1, ww2)
print loss
```

Theano: Simple Neural Net

Define symbolic variables:

$x = \text{data}$
 $y = \text{labels}$
 $w1 = \text{first-layer weights}$
 $w2 = \text{second-layer weights}$



```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Compile a function to compute loss, scores
f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores],
)

# Run the function
xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N)
ww1 = 1e-3 * np.random.randn(D, H)
ww2 = 1e-3 * np.random.randn(H, C)

loss, scores = f(xx, yy, ww1, ww2)
print loss
```

Theano: Simple Neural Net

Forward: Compute scores
(symbolically)



```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Compile a function to compute loss, scores
f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores],
)

# Run the function
xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N)
ww1 = 1e-3 * np.random.randn(D, H)
ww2 = 1e-3 * np.random.randn(H, C)

loss, scores = f(xx, yy, ww1, ww2)
print loss
```

Theano: Simple Neural Net

Forward: Compute probs, loss
(symbolically)



```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Compile a function to compute loss, scores
f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores],
)

# Run the function
xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N)
ww1 = 1e-3 * np.random.randn(D, H)
ww2 = 1e-3 * np.random.randn(H, C)

loss, scores = f(xx, yy, ww1, ww2)
print loss
```

Theano: Simple Neural Net

Compile a function that computes loss, scores



```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Compile a function to compute loss, scores
f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores],
)

# Run the function
xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N)
ww1 = 1e-3 * np.random.randn(D, H)
ww2 = 1e-3 * np.random.randn(H, C)

loss, scores = f(xx, yy, ww1, ww2)
print loss
```

Theano: Simple Neural Net

Stuff actual numpy arrays into
the function



```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Compile a function to compute loss, scores
f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores],
)

# Run the function
xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N)
ww1 = 1e-3 * np.random.randn(D, H)
ww2 = 1e-3 * np.random.randn(H, C)

loss, scores = f(xx, yy, ww1, ww2)
print loss
```

Theano: Computing Gradients

```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

Theano: Computing Gradients

```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

Same as before: define variables, compute scores and loss symbolically

Theano: Computing Gradients

```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

Theano computes gradients for us symbolically!



Theano: Computing Gradients

```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

Now the function returns loss,
scores, and gradients



Theano: Computing Gradients

```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

```
# Run the function
xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N)
ww1 = 1e-2 * np.random.randn(D, H)
ww2 = 1e-2 * np.random.randn(H, C)

learning_rate = 1e-1
for t in xrange(50):
    loss, scores, dw1, dw2 = f(xx, yy, ww1, ww2)
    print loss
    ww1 -= learning_rate * dw1
    ww2 -= learning rate * dw2
```

Use the function to perform gradient descent!

Theano: Computing Gradients

```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

```
# Run the function
xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N)
ww1 = 1e-2 * np.random.randn(D, H)
ww2 = 1e-2 * np.random.randn(H, C)

learning_rate = 1e-1
for t in xrange(50):
    loss, scores, dw1, dw2 = f(xx, yy, ww1, ww2)
    print loss
    ww1 -= learning_rate * dw1
    ww2 -= learning rate * dw2
```

Problem: Shipping weights and gradients to CPU on every iteration to update...

Theano: Shared Variables

```
N, D, H, C = 64, 1000, 100, 10
x = T.matrix('x')
y = T.vector('y', dtype='int64')

w1 = theano.shared(1e-3 * np.random.randn(D, H), name='w1')
w2 = theano.shared(1e-3 * np.random.randn(H, C), name='w2')

a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()
dw1, dw2 = T.grad(loss, [w1, w2])

learning_rate = 1e-1

train = theano.function(
    inputs=[x, y],
    outputs=loss,
    updates=(
        (w1, w1 - learning_rate * dw1),
        (w2, w2 - learning_rate * dw2)
    )
)
```

Same as before: Define dimensions, define symbolic variables for x, y

Theano: Shared Variables

```
N, D, H, C = 64, 1000, 100, 10
x = T.matrix('x')
y = T.vector('y', dtype='int64')

w1 = theano.shared(1e-3 * np.random.randn(D, H), name='w1')
w2 = theano.shared(1e-3 * np.random.randn(H, C), name='w2')

a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()
dw1, dw2 = T.grad(loss, [w1, w2])

learning_rate = 1e-1

train = theano.function(
    inputs=[x, y],
    outputs=loss,
    updates=(
        (w1, w1 - learning_rate * dw1),
        (w2, w2 - learning_rate * dw2)
    )
)
```

Define weights as **shared variables** that persist in the graph between calls; initialize with numpy arrays

Theano: Shared Variables

```
N, D, H, C = 64, 1000, 100, 10
x = T.matrix('x')
y = T.vector('y', dtype='int64')

w1 = theano.shared(1e-3 * np.random.randn(D, H), name='w1')
w2 = theano.shared(1e-3 * np.random.randn(H, C), name='w2')

a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()
dw1, dw2 = T.grad(loss, [w1, w2])

learning_rate = 1e-1

train = theano.function(
    inputs=[x, y],
    outputs=loss,
    updates=(
        (w1, w1 - learning_rate * dw1),
        (w2, w2 - learning_rate * dw2)
    )
)
```

Same as before: Compute scores, loss, gradients symbolically

Theano: Shared Variables

```
N, D, H, C = 64, 1000, 100, 10  
  
x = T.matrix('x')  
y = T.vector('y', dtype='int64')  
  
w1 = theano.shared(1e-3 * np.random.randn(D, H), name='w1')  
w2 = theano.shared(1e-3 * np.random.randn(H, C), name='w2')  
  
a = x.dot(w1)  
a_relu = T.nnet.relu(a)  
scores = a_relu.dot(w2)  
probs = T.nnet.softmax(scores)  
loss = T.nnet.categorical_crossentropy(probs, y).mean()  
dw1, dw2 = T.grad(loss, [w1, w2])  
  
learning_rate = 1e-1  
  
train = theano.function(  
    inputs=[x, y],  
    outputs=loss,  
    updates=(  
        (w1, w1 - learning_rate * dw1),  
        (w2, w2 - learning_rate * dw2)  
    )  
)
```

Compiled function inputs are x and y;
weights live in the graph

Theano: Shared Variables

```
N, D, H, C = 64, 1000, 100, 10  
  
x = T.matrix('x')  
y = T.vector('y', dtype='int64')  
  
w1 = theano.shared(1e-3 * np.random.randn(D, H), name='w1')  
w2 = theano.shared(1e-3 * np.random.randn(H, C), name='w2')  
  
a = x.dot(w1)  
a_relu = T.nnet.relu(a)  
scores = a_relu.dot(w2)  
probs = T.nnet.softmax(scores)  
loss = T.nnet.categorical_crossentropy(probs, y).mean()  
dw1, dw2 = T.grad(loss, [w1, w2])  
  
learning_rate = 1e-1  
  
train = theano.function(  
    inputs=[x, y],  
    outputs=loss,  
    updates=(  
        (w1, w1 - learning_rate * dw1),  
        (w2, w2 - learning_rate * dw2)  
    )  
)
```

Function includes an **update** that updates weights on every call

Theano: Shared Variables

```
N, D, H, C = 64, 1000, 100, 10
x = T.matrix('x')
y = T.vector('y', dtype='int64')

w1 = theano.shared(1e-3 * np.random.randn(D, H), name='w1')
w2 = theano.shared(1e-3 * np.random.randn(H, C), name='w2')

a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()
dw1, dw2 = T.grad(loss, [w1, w2])

learning_rate = 1e-1

train = theano.function(
    inputs=[x, y],
    outputs=loss,
    updates=(
        (w1, w1 - learning_rate * dw1),
        (w2, w2 - learning_rate * dw2)
    )
)
```

```
xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N)

for t in xrange(100):
    loss = train(xx, yy)
    print loss
```



To train the net, just call function repeatedly!

Theano: Other Topics

Conditionals: The `ifelse` and `switch` functions allow conditional control flow in the graph

Loops: The `scan` function allows for (some types) of loops in the computational graph; good for RNNs

Derivatives: Efficient Jacobian / vector products with R and L operators, symbolic Hessians (gradient of gradient)

Sparse matrices, optimizations, etc

Theano: Multi-GPU

Experimental model parallelism:

http://deeplearning.net/software/theano/tutorial/using_multi_gpu.html

Data parallelism using platoon:

<https://github.com/mila-udem/platoon>

Lasagne: High Level Wrapper

Lasagne gives layer abstractions,
sets up weights for you, writes
update rules for you

```
1 import numpy as np
2 import theano
3 import theano.tensor as T
4 import lasagne
5
6 N, D, H, C = 64, 1000, 100, 10
7
8 x = T.matrix('x')
9 y = T.vector('y', dtype='int64')
10
11 relu = lasagne.nonlinearities.rectify
12 softmax = lasagne.nonlinearities.softmax
13 net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
14 net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
15 net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)
16
17 probs = lasagne.layers.get_output(net)
18 loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()
19
20 params = lasagne.layers.get_all_params(net, trainable=True)
21 updates = lasagne.updates.nesterov_momentum(loss, params,
22                                              learning_rate=1e-2, momentum=0.0)
23
24 train_fn = theano.function([x, y], loss, updates=updates)
25
26 xx = np.random.randn(N, D)
27 yy = np.random.randint(C, size=N).astype(np.int64)
28
29 for t in xrange(100):
30     loss_val = train_fn(xx, yy)
31     print loss_val
```

Lasagne: High Level Wrapper

Set up symbolic Theano variables
for data, labels



```
1 import numpy as np
2 import theano
3 import theano.tensor as T
4 import lasagne
5
6 N, D, H, C = 64, 1000, 100, 10
7
8 x = T.matrix('x')
9 y = T.vector('y', dtype='int64')
10
11 relu = lasagne.nonlinearities.rectify
12 softmax = lasagne.nonlinearities.softmax
13 net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
14 net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
15 net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)
16
17 probs = lasagne.layers.get_output(net)
18 loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()
19
20 params = lasagne.layers.get_all_params(net, trainable=True)
21 updates = lasagne.updates.nesterov_momentum(loss, params,
22                                              learning_rate=1e-2, momentum=0.0)
23
24 train_fn = theano.function([x, y], loss, updates=updates)
25
26 xx = np.random.randn(N, D)
27 yy = np.random.randint(C, size=N).astype(np.int64)
28
29 for t in xrange(100):
30     loss_val = train_fn(xx, yy)
31     print loss_val
```

Lasagne: High Level Wrapper

Forward: Use Lasagne layers to set up layers; don't set up weights explicitly



```
1 import numpy as np
2 import theano
3 import theano.tensor as T
4 import lasagne
5
6 N, D, H, C = 64, 1000, 100, 10
7
8 x = T.matrix('x')
9 y = T.vector('y', dtype='int64')
10
11 relu = lasagne.nonlinearities.rectify
12 softmax = lasagne.nonlinearities.softmax
13 net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
14 net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
15 net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)
16
17 probs = lasagne.layers.get_output(net)
18 loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()
19
20 params = lasagne.layers.get_all_params(net, trainable=True)
21 updates = lasagne.updates.nesterov_momentum(loss, params,
22                                              learning_rate=1e-2, momentum=0.0)
23
24 train_fn = theano.function([x, y], loss, updates=updates)
25
26 xx = np.random.randn(N, D)
27 yy = np.random.randint(C, size=N).astype(np.int64)
28
29 for t in xrange(100):
30     loss_val = train_fn(xx, yy)
31     print loss_val
```

Lasagne: High Level Wrapper

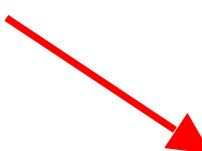
Forward: Use Lasagne layers to compute loss



```
1 import numpy as np
2 import theano
3 import theano.tensor as T
4 import lasagne
5
6 N, D, H, C = 64, 1000, 100, 10
7
8 x = T.matrix('x')
9 y = T.vector('y', dtype='int64')
10
11 relu = lasagne.nonlinearities.rectify
12 softmax = lasagne.nonlinearities.softmax
13 net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
14 net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
15 net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)
16
17 probs = lasagne.layers.get_output(net)
18 loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()
19
20 params = lasagne.layers.get_all_params(net, trainable=True)
21 updates = lasagne.updates.nesterov_momentum(loss, params,
22                                              learning_rate=1e-2, momentum=0.0)
23
24 train_fn = theano.function([x, y], loss, updates=updates)
25
26 xx = np.random.randn(N, D)
27 yy = np.random.randint(C, size=N).astype(np.int64)
28
29 for t in xrange(100):
30     loss_val = train_fn(xx, yy)
31     print loss_val
```

Lasagne: High Level Wrapper

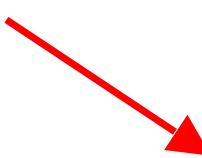
Lasagne gets parameters, and writes the update rule for you



```
1 import numpy as np
2 import theano
3 import theano.tensor as T
4 import lasagne
5
6 N, D, H, C = 64, 1000, 100, 10
7
8 x = T.matrix('x')
9 y = T.vector('y', dtype='int64')
10
11 relu = lasagne.nonlinearities.rectify
12 softmax = lasagne.nonlinearities.softmax
13 net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
14 net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
15 net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)
16
17 probs = lasagne.layers.get_output(net)
18 loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()
19
20 params = lasagne.layers.get_all_params(net, trainable=True)
21 updates = lasagne.updates.nesterov_momentum(loss, params,
22                                              learning_rate=1e-2, momentum=0.0)
23
24 train_fn = theano.function([x, y], loss, updates=updates)
25
26 xx = np.random.randn(N, D)
27 yy = np.random.randint(C, size=N).astype(np.int64)
28
29 for t in xrange(100):
30     loss_val = train_fn(xx, yy)
31     print loss_val
```

Lasagne: High Level Wrapper

Same as Theano: compile a function with updates, train model by calling function with arrays



```
1 import numpy as np
2 import theano
3 import theano.tensor as T
4 import lasagne
5
6 N, D, H, C = 64, 1000, 100, 10
7
8 x = T.matrix('x')
9 y = T.vector('y', dtype='int64')
10
11 relu = lasagne.nonlinearities.rectify
12 softmax = lasagne.nonlinearities.softmax
13 net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
14 net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
15 net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)
16
17 probs = lasagne.layers.get_output(net)
18 loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()
19
20 params = lasagne.layers.get_all_params(net, trainable=True)
21 updates = lasagne.updates.nesterov_momentum(loss, params,
22                                              learning_rate=1e-2, momentum=0.0)
23
24 train_fn = theano.function([x, y], loss, updates=updates)
25
26 xx = np.random.randn(N, D)
27 yy = np.random.randint(C, size=N).astype(np.int64)
28
29 for t in xrange(100):
30     loss_val = train_fn(xx, yy)
31     print loss_val
```

Keras: High level wrapper

keras is a layer on top of Theano;
makes common things easy to do

(Also supports TensorFlow
backend)

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

D, H, C = 1000, 100, 10

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=C))
model.add(Activation('softmax'))

sgd = SGD(lr=1e-3, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

N = 1000
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

model.fit(X, y, nb_epoch=5, batch_size=32, verbose=2)
```

Keras: High level wrapper

keras is a layer on top of Theano;
makes common things easy to do

Set up a two-layer ReLU net with softmax

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

D, H, C = 1000, 100, 10

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=C))
model.add(Activation('softmax'))

sgd = SGD(lr=1e-3, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

N = 1000
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

model.fit(X, y, nb_epoch=5, batch_size=32, verbose=2)
```

Keras: High level wrapper

keras is a layer on top of Theano;
makes common things easy to do

We will optimize the model using
SGD with Nesterov momentum

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

D, H, C = 1000, 100, 10

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=C))
model.add(Activation('softmax'))

sgd = SGD(lr=1e-3, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

N = 1000
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

model.fit(X, y, nb_epoch=5, batch_size=32, verbose=2)
```



Keras: High level wrapper

keras is a layer on top of Theano;
makes common things easy to do

Generate some random data and
train the model

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

D, H, C = 1000, 100, 10

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=C))
model.add(Activation('softmax'))

sgd = SGD(lr=1e-3, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

N = 1000
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

model.fit(X, y, nb_epoch=5, batch_size=32, verbose=2)
```



Keras: High level wrapper

Problem: It crashes, stack trace and error message not useful :(

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

D, H, C = 1000, 100, 10

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=C))
model.add(Activation('softmax'))

sgd = SGD(lr=1e-3, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

N = 1000
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

model.fit(X, y, nb_epoch=5, batch_size=32, verbose=2)
```

```
ValueError: Traceback (most recent call last)
<ipython-input-108-c68ea54ba915> in <module>()
    18 y = np.random.randint(C, size=N)
    19
--> 20 model.fit(X, y, nb_epoch=5, batch_size=32, verbose=2)
/home/justin/code/cs231n/frameworks/.env/local/lib/python2.7/site-packages/keras/models.py in fit(self, X, y, batch_size, nb_epoch, verbose, callbacks, validation_split, validation_data, shuffle, show_accuracy, class_weight, sample_weight)
    644         verbose=verbose, callbacks=callbacks,
    645         val_f=val_f, val_ins=val_ins,
--> 646         shuffle=shuffle, metrics=metrics)
    647
    648     def predict(self, X, batch_size=128, verbose=0):
/home/justin/code/cs231n/frameworks/.env/local/lib/python2.7/site-packages/keras/models.py in _fit(self, f, ins, outputs, batch_size, nb_epoch, verbose, callbacks, validation_split, validation_data, shuffle, show_accuracy, class_weight)
    78         batch_logs = [{} for _ in range(len(batch_ids))]
    79         callbacks.on_batch_begin(batch_index, batch_logs)
--> 80         outs = f(ins, batch)
    81         if type(outs) != list:
    82             outs = [outs]
/home/justin/code/cs231n/frameworks/.env/local/lib/python2.7/site-packages/keras/backend/theano_backend.py in call(self, inputs)
    382     def call(self, inputs):
    383         assert type(inputs) in {list, tuple}
--> 384         return self.function(*inputs)
    385
    386
/home/justin/code/cs231n/frameworks/.env/local/lib/python2.7/site-packages/theano/compile/function_module.py in __call__(self, *args, **kwargs)
    869         node=self.fn.nodes[self.fn.position_of_error],
    870         thunk=thunk,
--> 871         storage_map= getattr(self.fn, 'storage_map', None))
    872         else:
    873             # old-style linkers raise their own exceptions
/home/justin/code/cs231n/frameworks/.env/local/lib/python2.7/site-packages/theano/gof/link.py in raise_with_op(node, thunk, exc_info, storage_map)
    312         pass
--> 314     raise(exc_type, exc_value, exc_trace)
    315
    316
/home/justin/code/cs231n/frameworks/.env/local/lib/python2.7/site-packages/theano/compile/function_module.py in __call__(self, *args, **kwargs)
    857         t0_fn = time.time()
    858         try:
--> 859             outputs = self.fn()
    860         except Exception:
    861             if hasattr(self.fn, 'position_of_error'):
ValueError: Input dimension mis-match. (input[0].shape[1] = 10, input[3].shape[1] = 1)
Apply node that caused the error: Elemwise{Composite{((10 * 11 * 12 * 13 * 14) / (15 * 16 * 17 * 18 * 19))}}{Elemwise{Composite{((10 * 11 * 12 * 13 * 14) / (15 * 16 * 17 * 18 * 19))}}{Elementwise{AND{GE{((10, 11), LE{(10, 12))}}}, 0, InplaceDimShuffle{0,x}, 0, <TensorType{float32, matrix}>, SoftmaxWithBias, 0, InplaceDimShuffle{x,x}, 0, InplaceDimShuffle{x,x}, 0, Elementwise{Clip}{(0, 0)}}, 0, InplaceDimShuffle{x,x}, 0}}
Toposort index: 31
Inputs types: [TensorType(int8, matrix), TensorType(float32, (True, True)), TensorType(float32, col), TensorType(float32, matrix), TensorType(float32, matrix), TensorType(float32, (True, True)), TensorType(float32, col)]
Inputs shapes: [(32, 10), (1, 1), (32, 1), (32, 1), (32, 10), (1, 1), (32, 10), (32, 10), (32, 11)]
Inputs strides: [(4, 4), (4, 4), (4, 4), (4, 4), (4, 4), (4, 4), (4, 4), (4, 4), (4, 4)]
Inputs values: ['not shown', 'array([ 32.], dtype=float32)', 'not shown', 'not shown', 'not shown', 'array([ 32.], dtype=float32)', 'array([ 32.], dtype=float32)', 'not shown', 'not shown', 'array([ 32.], dtype=float32)', 'array([ 32.], dtype=float32)', 'not shown', 'not shown', 'array([ 32.], dtype=float32)', 'array([ 32.], dtype=float32)', 'not shown', 'not shown', 'array([ 32.], dtype=float32)', 'array([ 32.], dtype=float32), 0]]
Outputs clients: [[Sum{axis=[1], acc_dtype=float64}Elemwise{Composite{((10 * 11 * 12 * 13 * 14) / (15 * 16 * 17 * 18 * 19))}}.0]]]
HINT: Re-compiling with most Theano optimization disabled could give you a back-trace of when this node was created.
This can be done with by setting the Theano flag 'optimizer=fast_compile'. If that does not work, Theano optimizations can be disabled with 'optimizer=None'.
HINT: Use the Theano flag 'exception_verbosity=high' for a debugprint and storage map footprint of this apply node.
```

Keras: High level wrapper

Solution: y should be one-hot
(too much API for me ...)

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

D, H, C = 1000, 100, 10

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=C))
model.add(Activation('softmax'))

sgd = SGD(lr=1e-3, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

N = 1000
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

model.fit(X, y, nb_epoch=5, batch_size=32, verbose=2)
```

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD
from keras.utils import np_utils

D, H, C = 1000, 100, 10

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=C))
model.add(Activation('softmax'))

sgd = SGD(lr=1e-3, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

N, N_batch = 1000, 32
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)
y = np_utils.to_categorical(y)

model.fit(X, y, nb_epoch=5, batch_size=N_batch, verbose=2)
```

Theano: Pretrained Models

Lasagne Model Zoo has pretrained common architectures:

<https://github.com/Lasagne/Recipes/tree/master/modelzoo>

AlexNet with weights: https://github.com/uoguelph-mlrg/theano_alexnet

sklearn-theano: Run OverFeat and GoogLeNet forward, but no fine-tuning? <http://sklearn-theano.github.io>

caffe-theano-conversion: CS 231n project from last year: load models and weights from caffe! Not sure if full-featured <https://github.com/kitofans/caffe-theano-conversion>

Theano: Pretrained Models

Best choice

Lasagne Model Zoo has pretrained common architectures:

<https://github.com/Lasagne/Recipes/tree/master/modelzoo>

AlexNet with weights: https://github.com/uoguelph-mlrg/theano_alexnet

sklearn-theano: Run OverFeat and GoogLeNet forward, but no fine-tuning? <http://sklearn-theano.github.io>

caffe-theano-conversion: CS 231n project from last year: load models and weights from caffe! Not sure if full-featured <https://github.com/kitofans/caffe-theano-conversion>

Theano: Pros / Cons

- (+) Python + numpy
- (+) Computational graph is nice abstraction
- (+) RNNs fit nicely in computational graph
- (-) Raw Theano is somewhat low-level
- (+) High level wrappers (Keras, Lasagne) ease the pain
- (-) Error messages can be unhelpful
- (-) Large models can have long compile times
- (-) Much “fatter” than Torch; more magic
- (-) Patchy support for pretrained models

TensorFlow

<https://www.tensorflow.org>

TensorFlow

From Google

Very similar to Theano - all about computation graphs

Easy visualizations (TensorBoard)

Multi-GPU and multi-node training

TensorFlow: Two-Layer Net

```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17
18 learning_rate = 1e-2
19 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
20
21 xx = np.random.randn(N, D).astype(np.float32)
22 yy = np.zeros((N, C)).astype(np.float32)
23 yy[np.arange(N), np.random.randint(C, size=N)] = 1
24
25 with tf.Session() as sess:
26     sess.run(tf.initialize_all_variables())
27
28     for t in xrange(100):
29         _, loss_value = sess.run([train_step, loss],
30                               feed_dict={x: xx, y: yy})
31         print loss_value
32
```

TensorFlow: Two-Layer Net

Create placeholders for data and labels: These will be fed to the graph

```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17
18 learning_rate = 1e-2
19 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
20
21 xx = np.random.randn(N, D).astype(np.float32)
22 yy = np.zeros((N, C)).astype(np.float32)
23 yy[np.arange(N), np.random.randint(C, size=N)] = 1
24
25 with tf.Session() as sess:
26     sess.run(tf.initialize_all_variables())
27
28     for t in xrange(100):
29         _, loss_value = sess.run([train_step, loss],
30                               feed_dict={x: xx, y: yy})
31         print loss_value
32
```

TensorFlow: Two-Layer Net

Create Variables to hold weights; similar to Theano shared variables

Initialize variables with numpy arrays

```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17
18 learning_rate = 1e-2
19 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
20
21 xx = np.random.randn(N, D).astype(np.float32)
22 yy = np.zeros((N, C)).astype(np.float32)
23 yy[np.arange(N), np.random.randint(C, size=N)] = 1
24
25 with tf.Session() as sess:
26     sess.run(tf.initialize_all_variables())
27
28     for t in xrange(100):
29         _, loss_value = sess.run([train_step, loss],
30                               feed_dict={x: xx, y: yy})
31         print loss_value
32
```

TensorFlow: Two-Layer Net

Forward: Compute scores,
probs, loss (symbolically)

```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17
18 learning_rate = 1e-2
19 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
20
21 xx = np.random.randn(N, D).astype(np.float32)
22 yy = np.zeros((N, C)).astype(np.float32)
23 yy[np.arange(N), np.random.randint(C, size=N)] = 1
24
25 with tf.Session() as sess:
26     sess.run(tf.initialize_all_variables())
27
28     for t in xrange(100):
29         _, loss_value = sess.run([train_step, loss],
30                               feed_dict={x: xx, y: yy})
31         print loss_value
32
```

TensorFlow: Two-Layer Net

Running `train_step` will use SGD to minimize loss

```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17
18 learning_rate = 1e-2
19 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
20
21 xx = np.random.randn(N, D).astype(np.float32)
22 yy = np.zeros((N, C)).astype(np.float32)
23 yy[np.arange(N), np.random.randint(C, size=N)] = 1
24
25 with tf.Session() as sess:
26     sess.run(tf.initialize_all_variables())
27
28     for t in xrange(100):
29         _, loss_value = sess.run([train_step, loss],
30                               feed_dict={x: xx, y: yy})
31         print loss_value
32
```

TensorFlow: Two-Layer Net

Create an artificial dataset; y is one-hot like Keras

```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17
18 learning_rate = 1e-2
19 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
20
21 xx = np.random.randn(N, D).astype(np.float32)
22 yy = np.zeros((N, C)).astype(np.float32)
23 yy[np.arange(N), np.random.randint(C, size=N)] = 1
24
25 with tf.Session() as sess:
26     sess.run(tf.initialize_all_variables())
27
28     for t in xrange(100):
29         _, loss_value = sess.run([train_step, loss],
30                               feed_dict={x: xx, y: yy})
31         print loss_value
32
```

TensorFlow: Two-Layer Net

Actually train the model

```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17
18 learning_rate = 1e-2
19 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
20
21 xx = np.random.randn(N, D).astype(np.float32)
22 yy = np.zeros((N, C)).astype(np.float32)
23 yy[np.arange(N), np.random.randint(C, size=N)] = 1
24
25 with tf.Session() as sess:
26     sess.run(tf.initialize_all_variables())
27
28     for t in xrange(100):
29         _, loss_value = sess.run([train_step, loss],
30                               feed_dict={x: xx, y: yy})
31         print loss_value
```

TensorFlow: Tensorboard

Tensorboard makes it easy to visualize what's happening inside your models

```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17
18 loss_summary = tf.scalar_summary('loss', loss)
19 w1_hist = tf.histogram_summary('w1', w1)
20 w2_hist = tf.histogram_summary('w2', w2)
21
22 learning_rate = 1e-2
23 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
24
25 xx = np.random.randn(N, D).astype(np.float32)
26 yy = np.zeros((N, C)).astype(np.float32)
27 yy[np.arange(N), np.random.randint(C, size=N)] = 1
28
29 with tf.Session() as sess:
30     merged = tf.merge_all_summaries()
31     writer = tf.train.SummaryWriter('/tmp/fc_logs', sess.graph_def)
32     sess.run(tf.initialize_all_variables())
33
34 for t in xrange(100):
35     summary_str, _, loss_value = sess.run(
36         [merged, train_step, loss],
37         feed_dict={x: xx, y: yy})
38     writer.add_summary(summary_str, t)
39     print loss_value
```

TensorFlow: Tensorboard

Tensorboard makes it easy to visualize what's happening inside your models

Same as before, but now we create summaries for loss and weights



```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17
18 loss_summary = tf.scalar_summary('loss', loss)
19 w1_hist = tf.histogram_summary('w1', w1)
20 w2_hist = tf.histogram_summary('w2', w2)
21
22 learning_rate = 1e-2
23 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
24
25 xx = np.random.randn(N, D).astype(np.float32)
26 yy = np.zeros((N, C)).astype(np.float32)
27 yy[np.arange(N), np.random.randint(C, size=N)] = 1
28
29 with tf.Session() as sess:
30     merged = tf.merge_all_summaries()
31     writer = tf.train.SummaryWriter('/tmp/fc_logs', sess.graph_def)
32     sess.run(tf.initialize_all_variables())
33
34 for t in xrange(100):
35     summary_str, _, loss_value = sess.run(
36         [merged, train_step, loss],
37         feed_dict={x: xx, y: yy})
38     writer.add_summary(summary_str, t)
39     print loss_value
```

TensorFlow: Tensorboard

Tensorboard makes it easy to visualize what's happening inside your models

Create a special “merged” variable and a SummaryWriter object



```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17
18 loss_summary = tf.scalar_summary('loss', loss)
19 w1_hist = tf.histogram_summary('w1', w1)
20 w2_hist = tf.histogram_summary('w2', w2)
21
22 learning_rate = 1e-2
23 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
24
25 xx = np.random.randn(N, D).astype(np.float32)
26 yy = np.zeros((N, C)).astype(np.float32)
27 yy[np.arange(N), np.random.randint(C, size=N)] = 1
28
29 with tf.Session() as sess:
30     merged = tf.merge_all_summaries()
31     writer = tf.train.SummaryWriter('/tmp/fc_logs', sess.graph_def)
32     sess.run(tf.initialize_all_variables())
33
34     for t in xrange(100):
35         summary_str, _, loss_value = sess.run(
36             [merged, train_step, loss],
37             feed_dict={x: xx, y: yy})
38         writer.add_summary(summary_str, t)
39         print loss_value
```

TensorFlow: Tensorboard

Tensorboard makes it easy to visualize what's happening inside your models

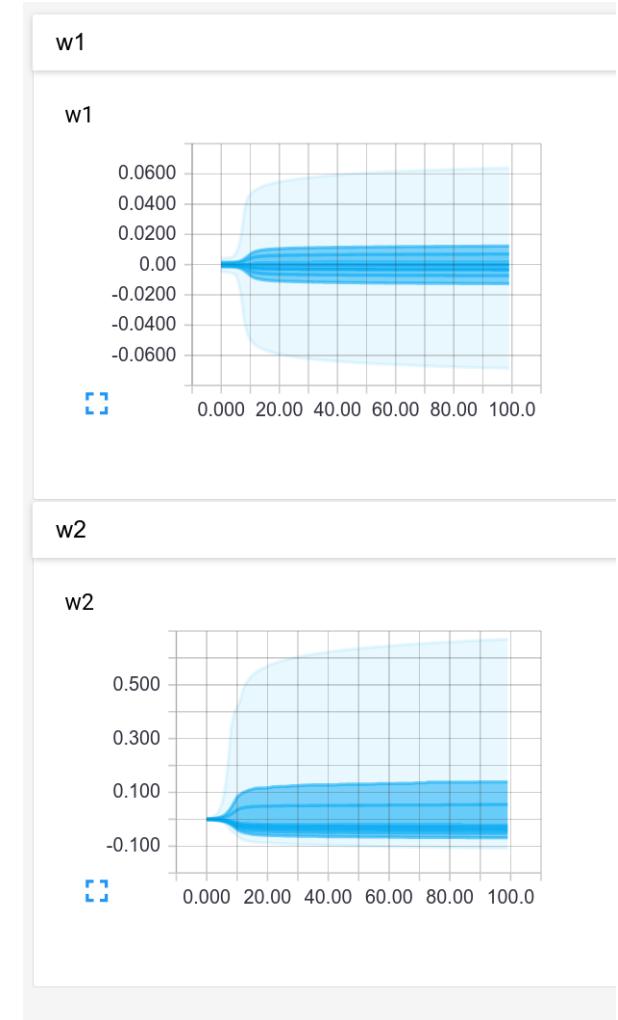
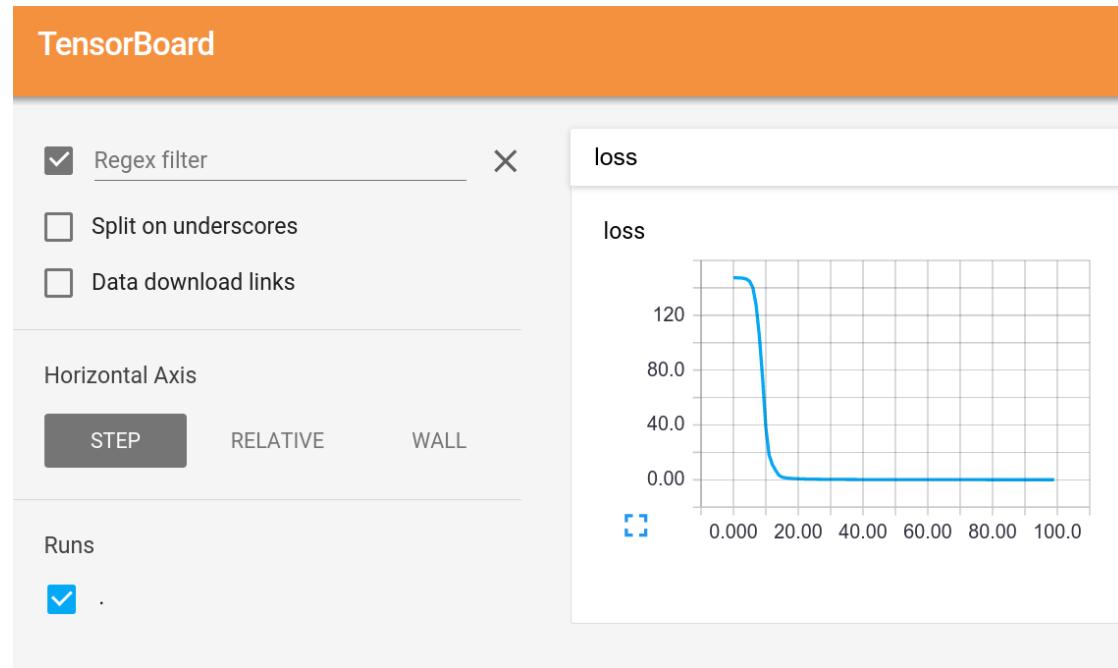
In the training loop, also run merged and pass its value to the writer

```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17
18 loss_summary = tf.scalar_summary('loss', loss)
19 w1_hist = tf.histogram_summary('w1', w1)
20 w2_hist = tf.histogram_summary('w2', w2)
21
22 learning_rate = 1e-2
23 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
24
25 xx = np.random.randn(N, D).astype(np.float32)
26 yy = np.zeros((N, C)).astype(np.float32)
27 yy[np.arange(N), np.random.randint(C, size=N)] = 1
28
29 with tf.Session() as sess:
30     merged = tf.merge_all_summaries()
31     writer = tf.train.SummaryWriter('/tmp/fc_logs', sess.graph_def)
32     sess.run(tf.initialize_all_variables())
33
34 for t in xrange(100):
35     summary_str, _, loss_value = sess.run(
36         [merged, train_step, loss],
37         feed_dict={x: xx, y: yy})
38     writer.add_summary(summary_str, t)
39     print loss_value
```



TensorFlow: Tensorboard

Start Tensorboard server, and we get graphs!



TensorFlow: TensorBoard

```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D], name='x')
7 y = tf.placeholder(tf.float32, shape=[None, C], name='y')
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32), name='w1')
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32), name='w2')
11
12 with tf.name_scope('scores') as scope:
13     a = tf.matmul(x, w1)
14     a_relu = tf.nn.relu(a)
15     scores = tf.matmul(a_relu, w2)
16 with tf.name_scope('loss') as scope:
17     probs = tf.nn.softmax(scores)
18     loss = -tf.reduce_sum(y * tf.log(probs))
19
20 loss_summary = tf.scalar_summary('loss', loss)
21 w1_hist = tf.histogram_summary('w1', w1)
22 w2_hist = tf.histogram_summary('w2', w2)
23
24 learning_rate = 1e-2
25 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
26
27 xx = np.random.randn(N, D).astype(np.float32)
28 yy = np.zeros((N, C)).astype(np.float32)
29 yy[np.arange(N), np.random.randint(C, size=N)] = 1
30
```

TensorFlow: TensorBoard

Add names to placeholders and variables

```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D], name='x')
7 y = tf.placeholder(tf.float32, shape=[None, C], name='y')
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32), name='w1')
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32), name='w2')
11
12 with tf.name_scope('scores') as scope:
13     a = tf.matmul(x, w1)
14     a_relu = tf.nn.relu(a)
15     scores = tf.matmul(a_relu, w2)
16 with tf.name_scope('loss') as scope:
17     probs = tf.nn.softmax(scores)
18     loss = -tf.reduce_sum(y * tf.log(probs))
19
20 loss_summary = tf.scalar_summary('loss', loss)
21 w1_hist = tf.histogram_summary('w1', w1)
22 w2_hist = tf.histogram_summary('w2', w2)
23
24 learning_rate = 1e-2
25 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
26
27 xx = np.random.randn(N, D).astype(np.float32)
28 yy = np.zeros((N, C)).astype(np.float32)
29 yy[np.arange(N), np.random.randint(C, size=N)] = 1
30
```

TensorFlow: TensorBoard

Add names to placeholders and variables

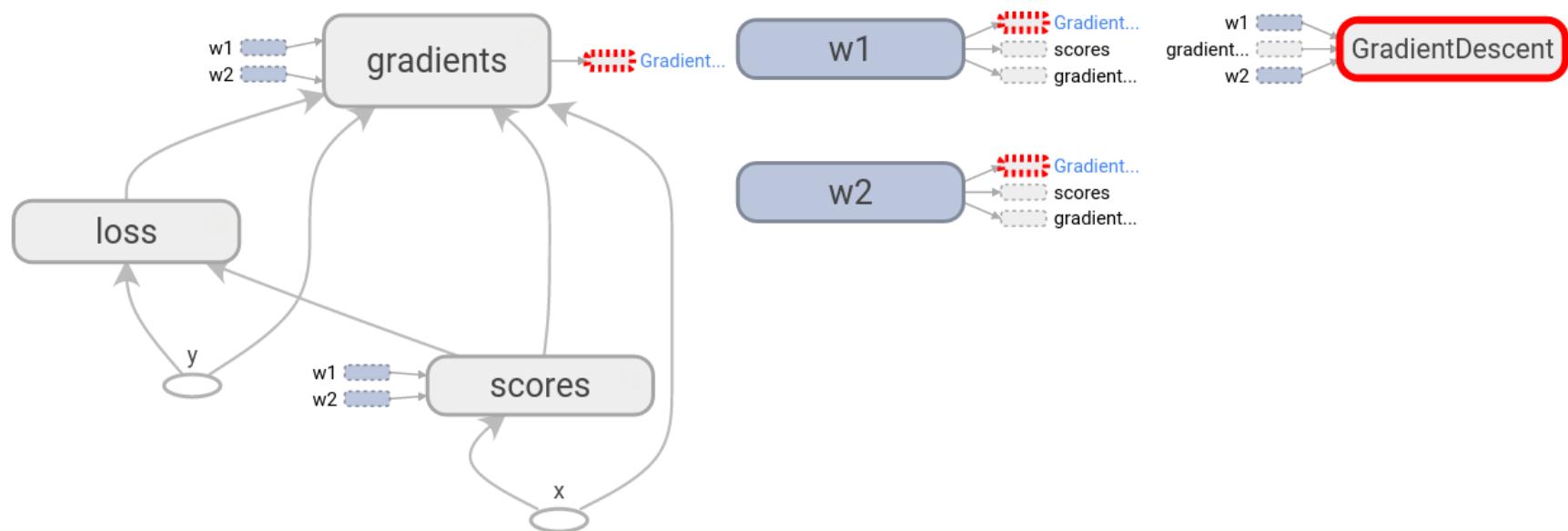
Break up the forward pass with name scoping



```
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5
6 x = tf.placeholder(tf.float32, shape=[None, D], name='x')
7 y = tf.placeholder(tf.float32, shape=[None, C], name='y')
8
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32), name='w1')
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32), name='w2')
11
12 with tf.name_scope('scores') as scope:
13     a = tf.matmul(x, w1)
14     a_relu = tf.nn.relu(a)
15     scores = tf.matmul(a_relu, w2)
16 with tf.name_scope('loss') as scope:
17     probs = tf.nn.softmax(scores)
18     loss = -tf.reduce_sum(y * tf.log(probs))
19
20 loss_summary = tf.scalar_summary('loss', loss)
21 w1_hist = tf.histogram_summary('w1', w1)
22 w2_hist = tf.histogram_summary('w2', w2)
23
24 learning_rate = 1e-2
25 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
26
27 xx = np.random.randn(N, D).astype(np.float32)
28 yy = np.zeros((N, C)).astype(np.float32)
29 yy[np.arange(N), np.random.randint(C, size=N)] = 1
30
```

TensorFlow: TensorBoard

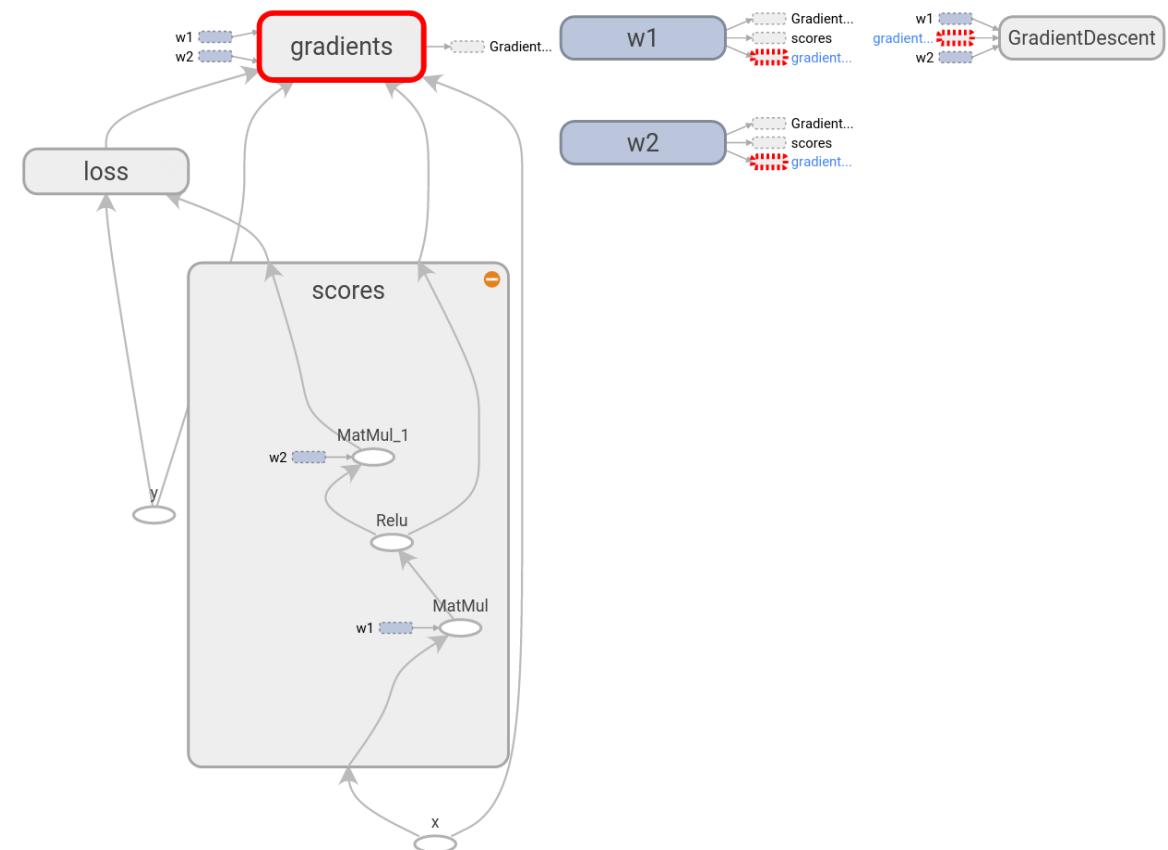
Tensorboard shows the graph!



TensorFlow: TensorBoard

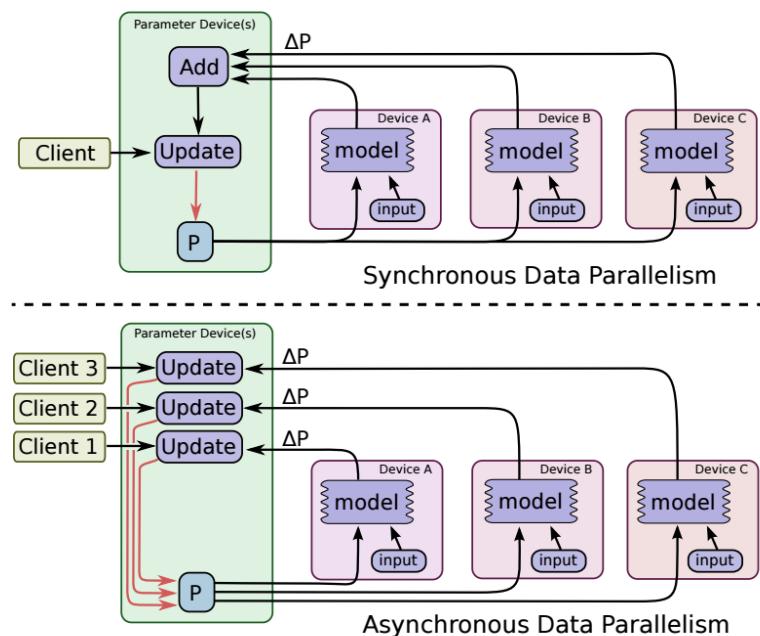
Tensorboard shows the graph!

Name scopes expand to show individual operations



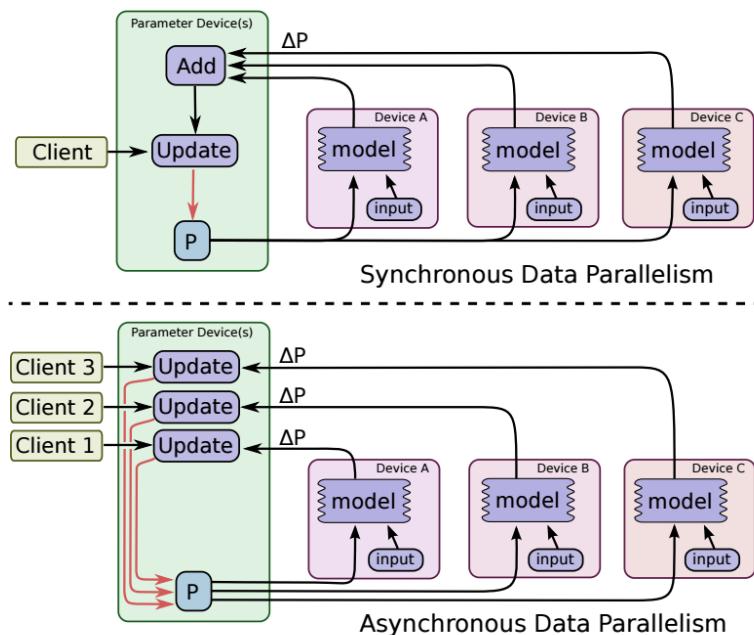
TensorFlow: Multi-GPU

Data parallelism:
synchronous or asynchronous

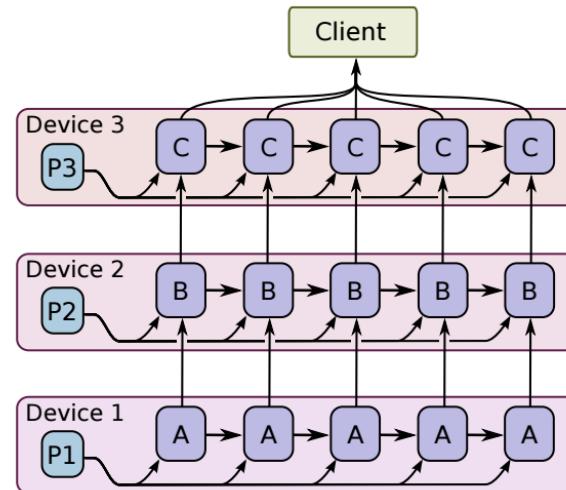


TensorFlow: Multi-GPU

Data parallelism:
synchronous or asynchronous

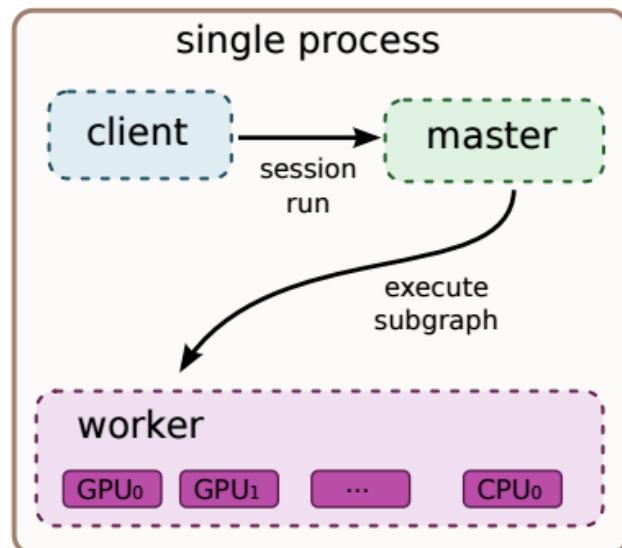


Model parallelism:
Split model across GPUs

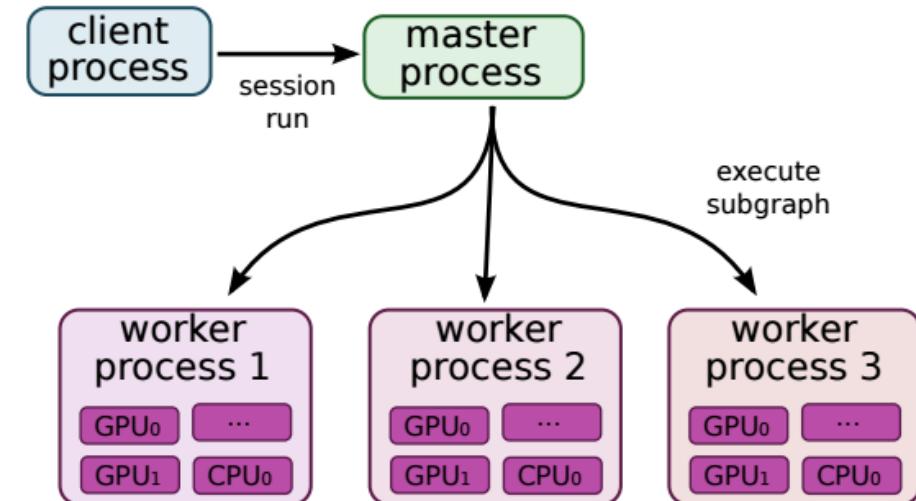


TensorFlow: Distributed

Single machine:
Like other frameworks



Many machines:
Not open source (yet) =(



TensorFlow: Pretrained Models

You can get a pretrained version of Inception here:

<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/android/README.md>

(In an Android example?? Very well-hidden)

The only one I could find =(

TensorFlow: Pros / Cons

- (+) Python + numpy
- (+) Computational graph abstraction, like Theano; great for RNNs
- (+) Much faster compile times than Theano
- (+) Slightly more convenient than raw Theano?
- (+) TensorBoard for visualization
- (+) Data AND model parallelism; best of all frameworks
- (+/-) Distributed models, but not open-source yet
- (-) Slower than other frameworks right now
- (-) Much “fatter” than Torch; more magic
- (-) Not many pretrained models

Overview

	Caffe	Torch	Theano	TensorFlow
Language	C++, Python	Lua	Python	Python
Pretrained	Yes ++	Yes ++	Yes (Lasagne)	Inception
Multi-GPU: Data parallel	Yes	Yes cunn.DataParallelTable	Yes platoon	Yes
Multi-GPU: Model parallel	No	Yes fbcunn.ModelParallel	Experimental	Yes (best)
Readable source code	Yes (C++)	Yes (Lua)	No	No
Good at RNN	No	Mediocre	Yes	Yes (best)

Use Cases

Extract AlexNet or VGG features?

Use Cases

Extract AlexNet or VGG features? **Use Caffe**

Use Cases

Fine-tune AlexNet for new classes?

Use Cases

Fine-tune AlexNet for new classes? **Use Caffe**

Use Cases

Image Captioning with finetuning?

Use Cases

Image Captioning with finetuning?

- > Need pretrained models (Caffe, Torch, Lasagne)
- > Need RNNs (Torch or Lasagne)
- > **Use Torch or Lasagna**

Use Cases

Segmentation? (Classify every pixel)

Use Cases

Segmentation? (Classify every pixel)

- > Need pretrained model (Caffe, Torch, Lasagna)
- > Need funny loss function
- > If loss function exists in Caffe: **Use Caffe**
- > If you want to write your own loss: **Use Torch**

Use Cases

Object Detection?

Use Cases

Object Detection?

- > Need pretrained model (Torch, Caffe, Lasagne)
- > Need lots of custom imperative code (NOT Lasagne)
- > **Use Caffe + Python or Torch**

Use Cases

Language modeling with new RNN structure?

Use Cases

Language modeling with new RNN structure?

- > Need easy recurrent nets (NOT Caffe, Torch)
- > No need for pretrained models
- > **Use Theano or TensorFlow**

Use Cases

Implement BatchNorm?

- > Don't want to derive gradient? **Theano** or **TensorFlow**
- > Implement efficient backward pass? **Use Torch**

My Recommendation

Feature extraction / finetuning existing models: Use Caffe

Complex uses of pretrained models: Use Lasagne or Torch

Write your own layers: Use Torch

Crazy RNNs: Use Theano or Tensorflow

Huge model, need model parallelism: Use TensorFlow

Caffe: Blobs

```
23 template <typename Dtype>
24 class Blob {
25 public:
26     Blob()
27         : data_(), diff_(), count_(0), capacity_(0) {}

28     /// @brief Deprecated; use <code>Blob(const vector<int>& shape)</code>.
29     explicit Blob(const int num, const int channels, const int height,
30                   const int width);
31     explicit Blob(const vector<int>& shape);

32     const Dtype* cpu_data() const;
33     void set_cpu_data(Dtype* data);
34     const int* gpu_shape() const;
35     const Dtype* gpu_data() const;
36     const Dtype* cpu_diff() const;
37     const Dtype* gpu_diff() const;
38     Dtype* mutable_cpu_data();
39     Dtype* mutable_gpu_data();
40     Dtype* mutable_cpu_diff();
41     Dtype* mutable_gpu_diff();

42     protected:
43         shared_ptr<SyncedMemory> data_;
44         shared_ptr<SyncedMemory> diff_;
45         shared_ptr<SyncedMemory> shape_data_;
46         vector<int> shape_;
47         int count_;
48         int capacity_;
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/blob.hpp>

Caffe: Blobs

N-dimensional array for
storing activations and
weights

```
23 template <typename Dtype>
24 class Blob {
25 public:
26     Blob()
27         : data_(), diff_(), count_(0), capacity_(0) {}
28
29     /// @brief Deprecated; use <code>Blob(const vector<int>& shape)</code>.
30     explicit Blob(const int num, const int channels, const int height,
31                   const int width);
32     explicit Blob(const vector<int>& shape);
33
34     const Dtype* cpu_data() const;
35     void set_cpu_data(Dtype* data);
36     const int* gpu_shape() const;
37     const Dtype* gpu_data() const;
38     const Dtype* cpu_diff() const;
39     const Dtype* gpu_diff() const;
40     Dtype* mutable_cpu_data();
41     Dtype* mutable_gpu_data();
42     Dtype* mutable_cpu_diff();
43     Dtype* mutable_gpu_diff();
44
45     protected:
46         shared_ptr<SyncedMemory> data_;
47         shared_ptr<SyncedMemory> diff_;
48         shared_ptr<SyncedMemory> shape_data_;
49         vector<int> shape_;
50         int count_;
51         int capacity_;
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/blob.hpp>

Caffe: Blobs

N-dimensional array for
storing activations and
weights

Template over datatype

```
23 template <typename Dtype>
24 class Blob {
25 public:
26     Blob()
27         : data_(), diff_(), count_(0), capacity_(0) {}
28
29     /// @brief Deprecated; use <code>Blob(const vector<int>& shape)</code>.
30     explicit Blob(const int num, const int channels, const int height,
31                 const int width);
32     explicit Blob(const vector<int>& shape);
33
34
35     const Dtype* cpu_data() const;
36     void set_cpu_data(Dtype* data);
37     const int* gpu_shape() const;
38     const Dtype* gpu_data() const;
39     const Dtype* cpu_diff() const;
40     const Dtype* gpu_diff() const;
41     Dtype* mutable_cpu_data();
42     Dtype* mutable_gpu_data();
43     Dtype* mutable_cpu_diff();
44     Dtype* mutable_gpu_diff();
45
46
47 protected:
48     shared_ptr<SyncedMemory> data_;
49     shared_ptr<SyncedMemory> diff_;
50     shared_ptr<SyncedMemory> shape_data_;
51     vector<int> shape_;
52     int count_;
53     int capacity ;
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/blob.hpp>

Caffe: Blobs

N-dimensional array for
storing activations and
weights

Template over datatype

Two parallel tensors:
data: values
diffs: gradients

```
23  template <typename Dtype>
24  class Blob {
25  public:
26      Blob()
27          : data_(), diff_(), count_(0), capacity_(0) {}

28
29      /// @brief Deprecated; use <code>Blob(const vector<int>& shape)</code>.
30      explicit Blob(const int num, const int channels, const int height,
31                     const int width);
32      explicit Blob(const vector<int>& shape);

33
34      const Dtype* cpu_data() const;
35      void set_cpu_data(Dtype* data);
36      const int* gpu_shape() const;
37      const Dtype* gpu_data() const;
38      const Dtype* cpu_diff() const;
39      const Dtype* gpu_diff() const;
40      Dtype* mutable_cpu_data();
41      Dtype* mutable_gpu_data();
42      Dtype* mutable_cpu_diff();
43      Dtype* mutable_gpu_diff();

44
45  protected:
46      shared_ptr<SyncedMemory> data_;
47      shared_ptr<SyncedMemory> diff_;
48      shared_ptr<SyncedMemory> shape_data_;
49      vector<int> shape_;
50      int count_;
51      int capacity_;
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/blob.hpp>

Caffe: Blobs

N-dimensional array for
storing activations and
weights

Template over datatype

Two parallel tensors:
data: values
diffs: gradients

Stores CPU / GPU versions
of each tensor

```
23  template <typename Dtype>
24  class Blob {
25  public:
26      Blob()
27          : data_(), diff_(), count_(0), capacity_(0) {}

28
29      /// @brief Deprecated; use <code>Blob(const vector<int>& shape)</code>.
30      explicit Blob(const int num, const int channels, const int height,
31                     const int width);
32      explicit Blob(const vector<int>& shape);

33
34      const Dtype* cpu_data() const;
35      void set_cpu_data(Dtype* data);
36      const int* gpu_shape() const;
37      const Dtype* gpu_data() const;
38      const Dtype* cpu_diff() const;
39      const Dtype* gpu_diff() const;
40      Dtype* mutable_cpu_data();
41      Dtype* mutable_gpu_data();
42      Dtype* mutable_cpu_diff();
43      Dtype* mutable_gpu_diff();

44
45      protected:
46
47          shared_ptr<SyncedMemory> data_;
48          shared_ptr<SyncedMemory> diff_;
49          shared_ptr<SyncedMemory> shape_data_;
50          vector<int> shape_;
51          int count_;
52          int capacity_;

53
54 }
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/blob.hpp>

Caffe: Layer

A small unit of computation

```
32 template <typename Dtype>
33 class Layer {
34 public:
35     /** @brief Using the CPU device, compute the layer output. */
36     virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
37                             const vector<Blob<Dtype>*>& top) = 0;
38     /**
39      * @brief Using the GPU device, compute the layer output.
40      *        Fall back to Forward_cpu() if unavailable.
41      */
42     virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
43                             const vector<Blob<Dtype>*>& top) {
44         // LOG(WARNING) << "Using CPU code as backup.";
45         return Forward_cpu(bottom, top);
46     }
47
48     /**
49      * @brief Using the CPU device, compute the gradients for any parameters and
50      *        for the bottom blobs if propagate_down is true.
51      */
52     virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
53                             const vector<bool>& propagate_down,
54                             const vector<Blob<Dtype>*>& bottom) = 0;
55     /**
56      * @brief Using the GPU device, compute the gradients for any parameters and
57      *        for the bottom blobs if propagate_down is true.
58      *        Fall back to Backward_cpu() if unavailable.
59      */
60     virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
61                             const vector<bool>& propagate_down,
62                             const vector<Blob<Dtype>*>& bottom) {
63         // LOG(WARNING) << "Using CPU code as backup.";
64         Backward_cpu(top, propagate_down, bottom);
65     }
66 }
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/layer.hpp>

Caffe: Layer

A small unit of computation

Forward: Use “bottom” data to compute “top” data

```
32 template <typename Dtype>
33 class Layer {
34 public:
35     /**
36      * @brief Using the CPU device, compute the layer output.
37      */
38     virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
39                             const vector<Blob<Dtype>*>& top) = 0;
40 /**
41  * @brief Using the GPU device, compute the layer output.
42  *        Fall back to Forward_cpu() if unavailable.
43  */
44     virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
45                             const vector<Blob<Dtype>*>& top) {
46         // LOG(WARNING) << "Using CPU code as backup.";
47         return Forward_cpu(bottom, top);
48     }
49 /**
50  * @brief Using the CPU device, compute the gradients for any parameters and
51  *        for the bottom blobs if propagate_down is true.
52  */
53     virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
54                               const vector<bool>& propagate_down,
55                               const vector<Blob<Dtype>*>& bottom) = 0;
56 /**
57  * @brief Using the GPU device, compute the gradients for any parameters and
58  *        for the bottom blobs if propagate_down is true.
59  *        Fall back to Backward_cpu() if unavailable.
60  */
61     virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
62                             const vector<bool>& propagate_down,
63                             const vector<Blob<Dtype>*>& bottom) {
64         // LOG(WARNING) << "Using CPU code as backup.";
65         Backward_cpu(top, propagate_down, bottom);
66     }
67 }
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/layer.hpp>

Caffe: Layer

A small unit of computation

Forward: Use “bottom” data to compute “top” data

Backward: Use “top” diffs to compute “bottom” diffs

```
32 template <typename Dtype>
33 class Layer {
34 public:
35     /** @brief Using the CPU device, compute the layer output. */
36     virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
37                             const vector<Blob<Dtype>*>& top) = 0;
38     /**
39      * @brief Using the GPU device, compute the layer output.
40      *        Fall back to Forward_cpu() if unavailable.
41      */
42     virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
43                             const vector<Blob<Dtype>*>& top) {
44         // LOG(WARNING) << "Using CPU code as backup.";
45         return Forward_cpu(bottom, top);
46     }
47
48     /**
49      * @brief Using the CPU device, compute the gradients for any parameters and
50      *        for the bottom blobs if propagate_down is true.
51      */
52     virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
53                             const vector<bool>& propagate_down,
54                             const vector<Blob<Dtype>*>& bottom) = 0;
55     /**
56      * @brief Using the GPU device, compute the gradients for any parameters and
57      *        for the bottom blobs if propagate_down is true.
58      *        Fall back to Backward_cpu() if unavailable.
59      */
60     virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
61                             const vector<bool>& propagate_down,
62                             const vector<Blob<Dtype>*>& bottom) {
63         // LOG(WARNING) << "Using CPU code as backup.";
64         Backward_cpu(top, propagate_down, bottom);
65     }
66 }
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/layer.hpp>

Caffe: Layer

A small unit of computation

Forward: Use “bottom” data to compute “top” data

Backward: Use “top” diffs to compute “bottom” diffs

Separate **CPU / GPU** implementations

```
32 template <typename Dtype>
33 class Layer {
34 public:
35     /** @brief Using the CPU device, compute the layer output. */
36     virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
37                             const vector<Blob<Dtype>*>& top) = 0;
38
39     * @brief Using the GPU device, compute the layer output.
40     *       Fall back to Forward_cpu() if unavailable.
41     */
42     virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
43                             const vector<Blob<Dtype>*>& top) {
44         // LOG(WARNING) << "Using CPU code as backup.";
45         return Forward_cpu(bottom, top);
46     }
47
48     /**
49      * @brief Using the CPU device, compute the gradients for any parameters and
50      *        for the bottom blobs if propagate_down is true.
51      */
52     virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
53                               const vector<bool>& propagate_down,
54                               const vector<Blob<Dtype>*>& bottom) = 0;
55
56     * @brief Using the GPU device, compute the gradients for any parameters and
57     *        for the bottom blobs if propagate_down is true.
58     *        Fall back to Backward_cpu() if unavailable.
59     */
60     virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
61                               const vector<bool>& propagate_down,
62                               const vector<Blob<Dtype>*>& bottom) {
63         // LOG(WARNING) << "Using CPU code as backup.";
64         Backward_cpu(top, propagate_down, bottom);
65     }
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/layer.hpp>

Caffe: Layer

Tons of different layer types:

Branch: master ▾		caffe / src / caffe / layers /
 jeffdonahue Remove incorrect cast of gemm int arg to Dtype in BiasLayer		
..		
 absval_layer.cpp		dismantle layer headers
 absval_layer.cu		dismantle layer headers
 accuracy_layer.cpp		dismantle layer headers
 argmax_layer.cpp		dismantle layer headers
 base_conv_layer.cpp		enable dilated deconvolution
 base_data_layer.cpp		dismantle layer headers
 base_data_layer.cu		dismantle layer headers
 batch_norm_layer.cpp		dismantle layer headers
 batch_norm_layer.cu		dismantle layer headers
■ ■ ■		
 conv_layer.cpp		add support for 2D dilated convolution
 conv_layer.cu		dismantle layer headers
 cudnn_conv_layer.cpp		dismantle layer headers
 cudnn_conv_layer.cu		Fix CuDNNConvolutionLayer for cuDNN v4

<https://github.com/BVLC/caffe/tree/master/src/caffe/layers>

Caffe: Layer

Tons of different layer types:

batch norm

convolution

cuDNN convolution

.cpp: CPU implementation

.cu: GPU implementation

Branch: master [master](#) | [cafe](#) / [src](#) / [cafe](#) / [layers](#) /

 jeffdonahue Remove incorrect cast of gemm int arg to Dtype in BiasLayer

..

[absval_layer.cpp](#) dismantle layer headers

[absval_layer.cu](#) dismantle layer headers

[accuracy_layer.cpp](#) dismantle layer headers

[argmax_layer.cpp](#) dismantle layer headers

[base_conv_layer.cpp](#) enable dilated deconvolution

[base_data_layer.cpp](#) dismantle layer headers

[base_data_layer.cu](#) dismantle layer headers

[batch_norm_layer.cpp](#) dismantle layer headers

[batch_norm_layer.cu](#) dismantle layer headers

..

[conv_layer.cpp](#) add support for 2D dilated convolution

[conv_layer.cu](#) dismantle layer headers

[cudnn_conv_layer.cpp](#) dismantle layer headers

[cudnn_conv_layer.cu](#) Fix CuDNNConvolutionLayer for cuDNN v4

<https://github.com/BVLC/caffe/tree/master/src/caffe/layers>

Caffe: Net

Collects layers into a DAG

Run all or part of the net
forward and **backward**

```
23 template <typename Dtype>
24 class Net {
25 public:
26     explicit Net(const NetParameter& param, const Net* root_net = NULL);
27     explicit Net(const string& param_file, Phase phase,
28                  const Net* root_net = NULL);
29     virtual ~Net() {}

41 /**
42 * The From and To variants of Forward and Backward operate on the
43 * (topological) ordering by which the net is specified. For general DAG
44 * networks, note that (1) computing from one layer to another might entail
45 * extra computation on unrelated branches, and (2) computation starting in
46 * the middle may be incorrect if all of the layers of a fan-in are not
47 * included.
48 */
49 Dtype ForwardFromTo(int start, int end);
50 Dtype ForwardFrom(int start);
51 Dtype ForwardTo(int end);
52 /// @brief Run forward using a set of bottom blobs, and return the result.
53 const vector<Blob<Dtype>*>& Forward(const vector<Blob<Dtype>*> & bottom,
54                                         Dtype* loss = NULL);

67 /**
68 * The network backward should take no input and output, since it solely
69 * computes the gradient w.r.t the parameters, and the data has already been
70 * provided during the forward pass.
71 */
72 void Backward();
73 void BackwardFromTo(int start, int end);
74 void BackwardFrom(int start);
75 void BackwardTo(int end);
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/net.hpp>

Caffe: Solver

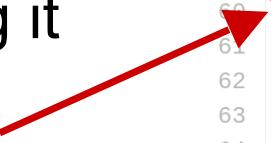
```
40 | template <typename Dtype>
41 | class Solver {
42 | public:
43 |
44 |     // The main entry of the solver function. In default, iter will be zero. Pass
45 |     // in a non-zero iter number to resume training for a pre-trained net.
46 |     virtual void Solve(const char* resume_file = NULL);
47 |     inline void Solve(const string resume_file) { Solve(resume_file.c_str()); }
48 |     void Step(int iters);
49 |
50 |     // The Restore method simply dispatches to one of the
51 |     // RestoreSolverStateFrom____ protected methods. You should implement these
52 |     // methods to restore the state from the appropriate snapshot type.
53 |     void Restore(const char* resume_file);
54 |
55 |     // The Solver::Snapshot function implements the basic snapshotting utility
56 |     // that stores the learned net. You should implement the SnapshotSolverState()
57 |     // function that produces a SolverState protocol buffer that needs to be
58 |     // written to disk together with the learned net.
59 |     void Snapshot();
```

<https://github.com/BVLC/caffe/blob/master/include/caffe/solver.hpp>

Caffe: Solver

Trains a Net by running it
forward / backward,
updating weights

```
40 template <typename Dtype>
41 class Solver {
42 public:
43     // The main entry of the solver function. In default, iter will be zero. Pass
44     // in a non-zero iter number to resume training for a pre-trained net.
45     virtual void Solve(const char* resume_file = NULL);
46     inline void Solve(const string resume_file) { Solve(resume_file.c_str()); }
47     void Step(int iters);
48
49     // The Restore method simply dispatches to one of the
50     // RestoreSolverStateFrom____ protected methods. You should implement these
51     // methods to restore the state from the appropriate snapshot type.
52     void Restore(const char* resume_file);
53
54     // The Solver::Snapshot function implements the basic snapshotting utility
55     // that stores the learned net. You should implement the SnapshotSolverState()
56     // function that produces a SolverState protocol buffer that needs to be
57     // written to disk together with the learned net.
58     void Snapshot();
```



<https://github.com/BVLC/caffe/blob/master/include/caffe/solver.hpp>

Caffe: Solver

Trains a Net by running it forward / backward, updating weights

Handles snapshotting, restoring from snapshots

```
40 template <typename Dtype>
41 class Solver {
42 public:
43
44     // The main entry of the solver function. In default, iter will be zero. Pass
45     // in a non-zero iter number to resume training for a pre-trained net.
46     virtual void Solve(const char* resume_file = NULL);
47     inline void Solve(const string resume_file) { Solve(resume_file.c_str()); }
48     void Step(int iter);
49
50     // The Restore method simply dispatches to one of the
51     // RestoreSolverStateFrom____ protected methods. You should implement these
52     // methods to restore the state from the appropriate snapshot type.
53     void Restore(const char* resume_file);
54
55     // The Solver::Snapshot function implements the basic snapshotting utility
56     // that stores the learned net. You should implement the SnapshotSolverState()
57     // function that produces a SolverState protocol buffer that needs to be
58     // written to disk together with the learned net.
59     void Snapshot();
60
61 }
```



<https://github.com/BVLC/caffe/blob/master/include/caffe/solver.hpp>

Caffe: Solver

Trains a Net by running it forward / backward, updating weights

Handles snapshotting, restoring from snapshots

Subclasses implement different update rules

```
40 | template <typename Dtype>
41 | class Solver {
42 | public:
43 |
44 |     // The main entry of the solver function. In default, iter will be zero. Pass
45 |     // in a non-zero iter number to resume training for a pre-trained net.
46 |     virtual void Solve(const char* resume_file = NULL);
47 |     inline void Solve(const string resume_file) { Solve(resume_file.c_str()); }
48 |     void Step(int iters);
49 |
50 |     // The Restore method simply dispatches to one of the
51 |     // RestoreSolverStateFrom____ protected methods. You should implement these
52 |     // methods to restore the state from the appropriate snapshot type.
53 |     void Restore(const char* resume_file);
54 |
55 |     // The Solver::Snapshot function implements the basic snapshotting utility
56 |     // that stores the learned net. You should implement the SnapshotSolverState()
57 |     // function that produces a SolverState protocol buffer that needs to be
58 |     // written to disk together with the learned net.
59 |     void Snapshot();
```

```
15 | template <typename Dtype>
16 | class SGDSolver : public Solver<Dtype> {
17 |
18 |     template <typename Dtype>
19 |     class RMSPropSolver : public SGDSolver<Dtype> {
20 |
21 |         template <typename Dtype>
22 |         class AdamSolver : public SGDSolver<Dtype> {
```

https://github.com/BVLC/caffe/blob/master/include/caffe/sgd_solvers.hpp