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!

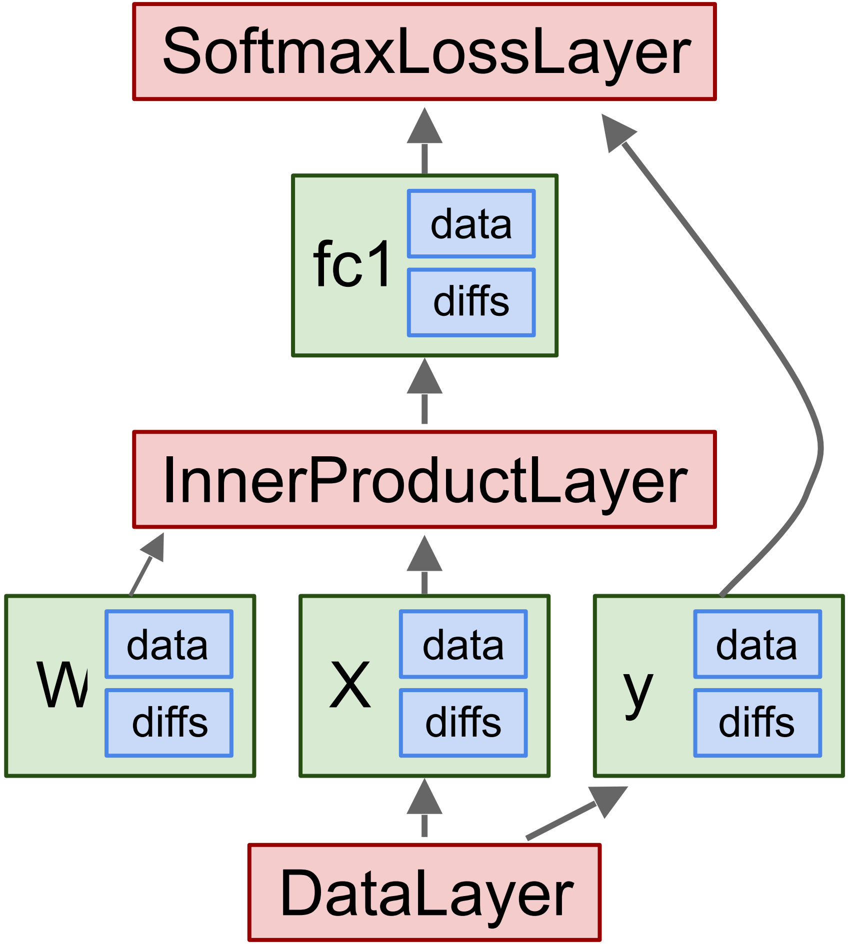
Caffe: Main classes

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

.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"
```

Java class

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

C++ class

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

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

Caffe: Protocol Buffers

```
message NetParameter {
  optional string name = 1; // consider giving the network a name
  // The input blobs to the network.
  repeated string input = 3;
  // The shape of the input blobs.
  repeated BlobShape input_shape = 8;

  // 4D input dimensions -- deprecated.  Use "shape" instead.
  // If specified, for each input blob there should be four
  // values specifying the num, channels, height and width of the input blob.
  // Thus, there should be a total of (4 * #input) numbers.
  repeated int32 input_dim = 4;

  // Whether the network will force every layer to carry out backward operation.
  // If set False, then whether to carry out backward is determined
  // automatically according to the net structure and learning rates.
  optional bool force_backward = 5 [default = false];
  // The current "state" of the network, including the phase, level, and stage.
  // Some layers may be included/excluded depending on this state and the states
  // specified in the layers' include and exclude fields.
  optional NetState state = 6;

  // Print debugging information about results while running Net::Forward,
  // Net::Backward, and Net::Update.
  optional bool debug_info = 7 [default = false];
```

```
message SolverParameter {
  //////////////////////////////////////////////////////////////////////////////
  // Specifying the train and test networks
  //
  // Exactly one train net must be specified using one of the following fields:
  //     train_net_param, train_net, net_param, net
  // One or more test nets may be specified using any of the following fields:
  //     test_net_param, test_net, net_param, net
  // If more than one test net field is specified (e.g., both net and
  // test_net are specified), they will be evaluated in the field order given
  // above: (1) test_net_param, (2) test_net, (3) net_param/net.
  // A test_iter must be specified for each test_net.
  // A test_level and/or a test_stage may also be specified for each test_net.
  //////////////////////////////////////////////////////////////////////////////

  // Proto filename for the train net, possibly combined with one or more
  // test nets.
  optional string net = 24;
  // Inline train net param, possibly combined with one or more test nets.
  optional NetParameter net_param = 25;

  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"
  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
```

Layers and Blobs often have same name!

```
  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"
  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
}
```

Layers and Blobs often have same name!

Learning rates (weight + bias)

Regularization (weight + bias)

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
```

Layers and Blobs often have same name!

Learning rates (weight + bias)

Regularization (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"
  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
}
```

Layers and Blobs often have same name!

Set these to 0 to freeze a layer

Learning rates (weight + bias)

Regularization (weight + bias)

Number of output classes

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

```
name: "ResNet-152"
input: "data"
input_dim: 1
input_dim: 3
input_dim: 224
input_dim: 224

layer {
        bottom: "data"
        top: "conv1"
        name: "conv1"
        type: "Convolution"
        convolution_param {
                num_output: 64
                kernel_size: 7
                pad: 3
                stride: 2
                bias_term: false
        }
}

layer {
        bottom: "conv1"
        top: "conv1"
        name: "bn_conv1"
        type: "BatchNorm"
        batch_norm_param {
                use_global_stats: true
        }
}
```

```
layer {
        bottom: "res5c"
        top: "pool5"
        name: "pool5"
        type: "Pooling"
        pooling_param {
                kernel_size: 7
                stride: 1
                pool: AVE
        }
}

layer {
        bottom: "pool5"
        top: "fc1000"
        name: "fc1000"
        type: "InnerProduct"
        inner_product_param {
                num_output: 1000
        }
}

layer {
        bottom: "fc1000"
        top: "prob"
        name: "prob"
        type: "Softmax"
}
```

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"
  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]
```

Different name: weights reinitialized

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 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

```
net: "models/bvlc_alexnet/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "models/bvlc_alexnet/caffe_alexnet_train"
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

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 a 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.

```lua
-- 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

```python
import numpy as np

# Simple feedforward network (no biases) in numpy

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

# First and second layer weights
w1 = np.random.randn(D, H)
w2 = np.random.randn(H, C)

# Random input data
x = np.random.randn(N, D)

# Forward pass
a = x.dot(w1)            # First layer
a = np.maximum(a, 0)  # In-place ReLU
scores = a.dot(w2)     # Second layer

print scores
```

Torch: Tensors

Torch tensors are just like numpy arrays

```python
import numpy as np

# Simple feedforward network (no biases) in numpy

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

# First and second layer weights
w1 = np.random.randn(D, H)
w2 = np.random.randn(H, C)

# Random input data
x = np.random.randn(N, D)

# Forward pass
a = x.dot(w1)             # First layer
a = np.maximum(a, 0)  # In-place ReLU
scores = a.dot(w2)     # Second layer

print scores
```

```lua
require 'torch'

-- Simple feedforward network (no biases) in torch

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

-- First and second layer weights
local w1 = torch.randn(D, H)
local w2 = torch.randn(H, C)

-- Random input data
local x = torch.randn(N, D)

-- Forward pass
local a = torch.mm(x, w1)          -- First layer
a:cmax(0)                          -- In-place ReLU
local scores = torch.mm(a, w2)     -- Second layer

print(scores)
```

Torch: Tensors

Like numpy, can easily change data type:

```python
import numpy as np


# Simple feedforward network (no biases) in numpy

dtype = np.float32  # Use 32-bit floats

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

# First and second layer weights
w1 = np.random.randn(D, H).astype(dtype)
w2 = np.random.randn(H, C).astype(dtype)

# Random input data
x = np.random.randn(N, D).astype(dtype)

# Forward pass
a = x.dot(w1)          # First layer
a = np.maximum(a, 0)  # In-place ReLU
scores = a.dot(w2)    # Second layer

print scores
```

```lua
require 'torch'


-- Simple feedforward network (no biases) in torch

local dtype = 'torch.FloatTensor'  -- Use 32-bit floats

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

-- First and second layer weights
local w1 = torch.randn(D, H):type(dtype)
local w2 = torch.randn(H, C):type(dtype)

-- Random input data
local x = torch.randn(N, D):type(dtype)

-- Forward pass
local a = torch.mm(x, w1)        -- First layer
a:cmax(0)                        -- In-place ReLU
local scores = torch.mm(a, w2)  -- Second layer

print(scores)
```

Torch: Tensors

Unlike numpy, GPU is just a datatype away:

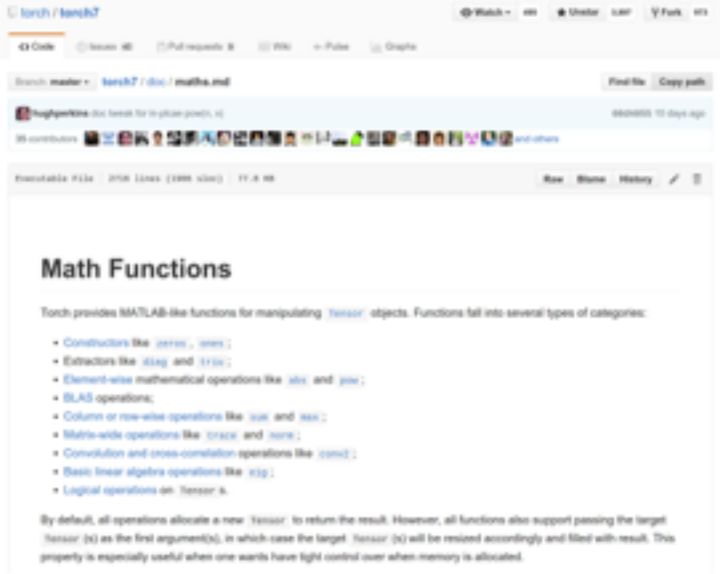
```python
import numpy as np


# Simple feedforward network (no biases) in numpy

dtype = np.float32  # Use 32-bit floats

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

# First and second layer weights
w1 = np.random.randn(D, H).astype(dtype)
w2 = np.random.randn(H, C).astype(dtype)

# Random input data
x = np.random.randn(N, D).astype(dtype)

# Forward pass
a = x.dot(w1)              # First layer
a = np.maximum(a, 0)  # In-place ReLU
scores = a.dot(w2)     # Second layer

print scores
```

```lua
require 'torch'
require 'cutorch'

-- Simple feedforward network (no biases) in torch

local dtype = 'torch.CudaTensor'   -- Use CUDA

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

-- First and second layer weights
local w1 = torch.randn(D, H):type(dtype)
local w2 = torch.randn(H, C):type(dtype)

-- Random input data
local x = torch.randn(N, D):type(dtype)

-- Forward pass
local a = torch.mm(x, w1)        -- First layer
a:cmax(0)                        -- In-place ReLU
local scores = torch.mm(a, w2)  -- Second layer

print(scores)
```

Torch: Tensors

Documentation on GitHub:

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



https://github.com/torch/torch7/blob/master/doc/maths.md

Torch: nn

nn module lets you easily build and train neural nets

```lua
require 'torch'
require 'nn'


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

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss

-- Generate some random input data
local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)

-- Forward pass: Compute scores and loss
local scores = net:forward(x)
local loss = crit:forward(scores, y)

-- Backward pass: compute gradients
grad_weights:zero()
local dscores = crit:backward(scores, y)
local dx = net:backward(x, dscores)

-- Make a gradient step
local learning_rate = 1e-3
weights:add(-learning_rate, grad_weights)
```

Torch: nn

nn module lets you easily build and train neural nets

Build a two-layer ReLU net

```
require 'torch'
require 'nn'


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

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss

-- Generate some random input data
local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)

-- Forward pass: Compute scores and loss
local scores = net:forward(x)
local loss = crit:forward(scores, y)

-- Backward pass: compute gradients
grad_weights:zero()
local dscores = crit:backward(scores, y)
local dx = net:backward(x, dscores)

-- Make a gradient step
local learning_rate = 1e-3
weights:add(-learning_rate, grad_weights)
```

Torch: nn

nn module lets you easily build and train neural nets

Get weights and gradient for entire network

```lua
require 'torch'
require 'nn'


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

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss

-- Generate some random input data
local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)

-- Forward pass: Compute scores and loss
local scores = net:forward(x)
local loss = crit:forward(scores, y)

-- Backward pass: compute gradients
grad_weights:zero()
local dscores = crit:backward(scores, y)
local dx = net:backward(x, dscores)

-- Make a gradient step
local learning_rate = 1e-3
weights:add(-learning_rate, grad_weights)
```

Torch: nn

nn module lets you easily build and train neural nets

Use a softmax loss function

```lua
require 'torch'
require 'nn'


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

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss

-- Generate some random input data
local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)

-- Forward pass: Compute scores and loss
local scores = net:forward(x)
local loss = crit:forward(scores, y)

-- Backward pass: compute gradients
grad_weights:zero()
local dscores = crit:backward(scores, y)
local dx = net:backward(x, dscores)

-- Make a gradient step
local learning_rate = 1e-3
weights:add(-learning_rate, grad_weights)
```

Torch: nn

nn module lets you easily build and train neural nets

Generate random data

```lua
require 'torch'
require 'nn'


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

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss

-- Generate some random input data
local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)

-- Forward pass: Compute scores and loss
local scores = net:forward(x)
local loss = crit:forward(scores, y)

-- Backward pass: compute gradients
grad_weights:zero()
local dscores = crit:backward(scores, y)
local dx = net:backward(x, dscores)

-- Make a gradient step
local learning_rate = 1e-3
weights:add(-learning_rate, grad_weights)
```

Torch: nn

nn module lets you easily build and train neural nets

Forward pass: compute scores and loss

```lua
require 'torch'
require 'nn'


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

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss

-- Generate some random input data
local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)

-- Forward pass: Compute scores and loss
local scores = net:forward(x)
local loss = crit:forward(scores, y)

-- Backward pass: compute gradients
grad_weights:zero()
local dscores = crit:backward(scores, y)
local dx = net:backward(x, dscores)

-- Make a gradient step
local learning_rate = 1e-3
weights:add(-learning_rate, grad_weights)
```

Torch: nn

nn module lets you easily build and train neural nets

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

```lua
require 'torch'
require 'nn'


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

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss

-- Generate some random input data
local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)

-- Forward pass: Compute scores and loss
local scores = net:forward(x)
local loss = crit:forward(scores, y)

-- Backward pass: compute gradients
grad_weights:zero()
local dscores = crit:backward(scores, y)
local dx = net:backward(x, dscores)

-- Make a gradient step
local learning_rate = 1e-3
weights:add(-learning_rate, grad_weights)
```

Torch: nn

nn module lets you easily build and train neural nets

Update: Make a gradient descent step

```
require 'torch'
require 'nn'


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

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss

-- Generate some random input data
local x = torch.randn(N, D)
local y = torch.Tensor(N):random(C)

-- Forward pass: Compute scores and loss
local scores = net:forward(x)
local loss = crit:forward(scores, y)

-- Backward pass: compute gradients
grad_weights:zero()
local dscores = crit:backward(scores, y)
local dx = net:backward(x, dscores)

-- Make a gradient step
local learning_rate = 1e-3
weights:add(-learning_rate, grad_weights)
```

Torch: cunn

Running on GPU is easy:

```
require 'torch'
require 'cutorch'
require 'nn'
require 'cunn'

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

local dtype = 'torch.CudaTensor'

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))
net:type(dtype)

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss
crit:type(dtype)

-- Generate some random input data
local x = torch.randn(N, D):type(dtype)
local y = torch.Tensor(N):random(C):type(dtype)

-- Forward pass: Compute scores and loss
local scores = net:forward(x)
local loss = crit:forward(scores, y)

-- Backward pass: compute gradients
grad_weights:zero()
local dscores = crit:backward(scores, y)
local dx = net:backward(x, dscores)

-- Make a gradient step
local learning_rate = 1e-3
weights:add(-learning_rate, grad_weights)
```

Torch: cunn

Running on GPU is easy:

Import a few new packages

```lua
require 'torch'
require 'cutorch'
require 'nn'
require 'cunn'

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

local dtype = 'torch.CudaTensor'

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))
net:type(dtype)

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss
crit:type(dtype)

-- Generate some random input data
local x = torch.randn(N, D):type(dtype)
local y = torch.Tensor(N):random(C):type(dtype)

-- Forward pass: Compute scores and loss
local scores = net:forward(x)
local loss = crit:forward(scores, y)

-- Backward pass: compute gradients
grad_weights:zero()
local dscores = crit:backward(scores, y)
local dx = net:backward(x, dscores)

-- Make a gradient step
local learning_rate = 1e-3
weights:add(-learning_rate, grad_weights)
```

Torch: cunn

Running on GPU is easy:

Import a few new packages

Cast network and criterion

```
require 'torch'
require 'cutorch'
require 'nn'
require 'cunn'

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

local dtype = 'torch.CudaTensor'

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))
net:type(dtype)

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss
crit:type(dtype)

-- Generate some random input data
local x = torch.randn(N, D):type(dtype)
local y = torch.Tensor(N):random(C):type(dtype)

-- Forward pass: Compute scores and loss
local scores = net:forward(x)
local loss = crit:forward(scores, y)

-- Backward pass: compute gradients
grad_weights:zero()
local dscores = crit:backward(scores, y)
local dx = net:backward(x, dscores)

-- Make a gradient step
local learning_rate = 1e-3
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

```lua
require 'torch'
require 'cutorch'
require 'nn'
require 'cunn'

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

local dtype = 'torch.CudaTensor'

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))
net:type(dtype)

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss
crit:type(dtype)

-- Generate some random input data
local x = torch.randn(N, D):type(dtype)
local y = torch.Tensor(N):random(C):type(dtype)

-- Forward pass: Compute scores and loss
local scores = net:forward(x)
local loss = crit:forward(scores, y)

-- Backward pass: compute gradients
grad_weights:zero()
local dscores = crit:backward(scores, y)
local dx = net:backward(x, dscores)

-- Make a gradient step
local learning_rate = 1e-3
weights:add(-learning_rate, grad_weights)
```

Torch: optim

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

```
require 'torch'
require 'nn'
require 'optim'

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

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss

-- Calback to interface with optim methods
local function f(w)
  assert(w == weights)

  -- Generate some random input data
  local x = torch.randn(N, D)
  local y = torch.Tensor(N):random(C)

  -- Forward pass: Compute scores and loss
  local scores = net:forward(x)
  local loss = crit:forward(scores, y)

  -- Backward pass: compute gradients
  grad_weights:zero()
  local dscores = crit:backward(scores, y)
  local dx = net:backward(x, dscores)

  return loss, grad_weights
end

-- Make a step using Adam
local state = {learningRate=1e-3}
optim.adam(f, weights, state)
```

Torch: optim

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

Import optim package

```lua
require 'torch'
require 'nn'
require 'optim'

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

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss

-- Calback to interface with optim methods
local function f(w)
  assert(w == weights)

  -- Generate some random input data
  local x = torch.randn(N, D)
  local y = torch.Tensor(N):random(C)

  -- Forward pass: Compute scores and loss
  local scores = net:forward(x)
  local loss = crit:forward(scores, y)

  -- Backward pass: compute gradients
  grad_weights:zero()
  local dscores = crit:backward(scores, y)
  local dx = net:backward(x, dscores)

  return loss, grad_weights
end

-- Make a step using Adam
local state = {learningRate=1e-3}
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

```lua
require 'torch'
require 'nn'
require 'optim'

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

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss

-- Calback to interface with optim methods
local function f(w)
  assert(w == weights)

  -- Generate some random input data
  local x = torch.randn(N, D)
  local y = torch.Tensor(N):random(C)

  -- Forward pass: Compute scores and loss
  local scores = net:forward(x)
  local loss = crit:forward(scores, y)

  -- Backward pass: compute gradients
  grad_weights:zero()
  local dscores = crit:backward(scores, y)
  local dx = net:backward(x, dscores)

  return loss, grad_weights
end

-- Make a step using Adam
local state = {learningRate=1e-3}
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

```lua
require 'torch'
require 'nn'
require 'optim'

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

-- Build a one-layer ReLU network
local net = nn.Sequential()
net:add(nn.Linear(D, H))
net:add(nn.ReLU())
net:add(nn.Linear(H, C))

-- Collect all weights and gradients in a single Tensor
local weights, grad_weights = net:getParameters()

-- Loss functions are called "criterions"
local crit = nn.CrossEntropyCriterion()  -- Softmax loss

-- Callback to interface with optim methods
local function f(w)
  assert(w == weights)

  -- Generate some random input data
  local x = torch.randn(N, D)
  local y = torch.Tensor(N):random(C)

  -- Forward pass: Compute scores and loss
  local scores = net:forward(x)
  local loss = crit:forward(scores, y)

  -- Backward pass: compute gradients
  grad_weights:zero()
  local dscores = crit:backward(scores, y)
  local dx = net:backward(x, dscores)

  return loss, grad_weights
end

-- Make a step using Adam
local state = {learningRate=1e-3}
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

```lua
local Linear, parent = torch.class('nn.Linear', 'nn.Module')

function Linear:__init(inputSize, outputSize, bias)
   parent.__init(self)
   local bias = ((bias == nil) and true) or bias
   self.weight = torch.Tensor(outputSize, inputSize)
   self.gradWeight = torch.Tensor(outputSize, inputSize)
   if bias then
      self.bias = torch.Tensor(outputSize)
      self.gradBias = torch.Tensor(outputSize)
   end
   self:reset()
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

updateOutput: Forward pass; compute output

```lua
local Linear, parent = torch.class('nn.Linear', 'nn.Module')
```

```lua
function Linear:updateOutput(input)
   if input:dim() == 1 then
      self.output:resize(self.weight:size(1))
      if self.bias then self.output:copy(self.bias) else self.output:zero() end
      self.output:addmv(1, self.weight, input)
   elseif input:dim() == 2 then
      local nframe = input:size(1)
      local nElement = self.output:nElement()
      self.output:resize(nframe, self.weight:size(1))
      if self.output:nElement() ~= nElement then
         self.output:zero()
      end
      self.addBuffer = self.addBuffer or input.new()
      if self.addBuffer:nElement() ~= nframe then
         self.addBuffer:resize(nframe):fill(1)
      end
      self.output:addmm(0, self.output, 1, input, self.weight:t())
      if self.bias then self.output:addr(1, self.addBuffer, self.bias) end
   else
      error('input must be vector or matrix')
   end

   return self.output
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

```lua
local Linear, parent = torch.class('nn.Linear', 'nn.Module')
```

```lua
function Linear:updateGradInput(input, gradOutput)
   if self.gradInput then

      local nElement = self.gradInput:nElement()
      self.gradInput:resizeAs(input)
      if self.gradInput:nElement() ~= nElement then
         self.gradInput:zero()
      end
      if input:dim() == 1 then
         self.gradInput:addmv(0, 1, self.weight:t(), gradOutput)
      elseif input:dim() == 2 then
         self.gradInput:addmm(0, 1, gradOutput, self.weight)
      end

      return self.gradInput
   end
end
```

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

Torch: Modules

```lua
local Linear, parent = torch.class('nn.Linear', 'nn.Module')
```

Caffe has Nets and Layers;
Torch just has Modules

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

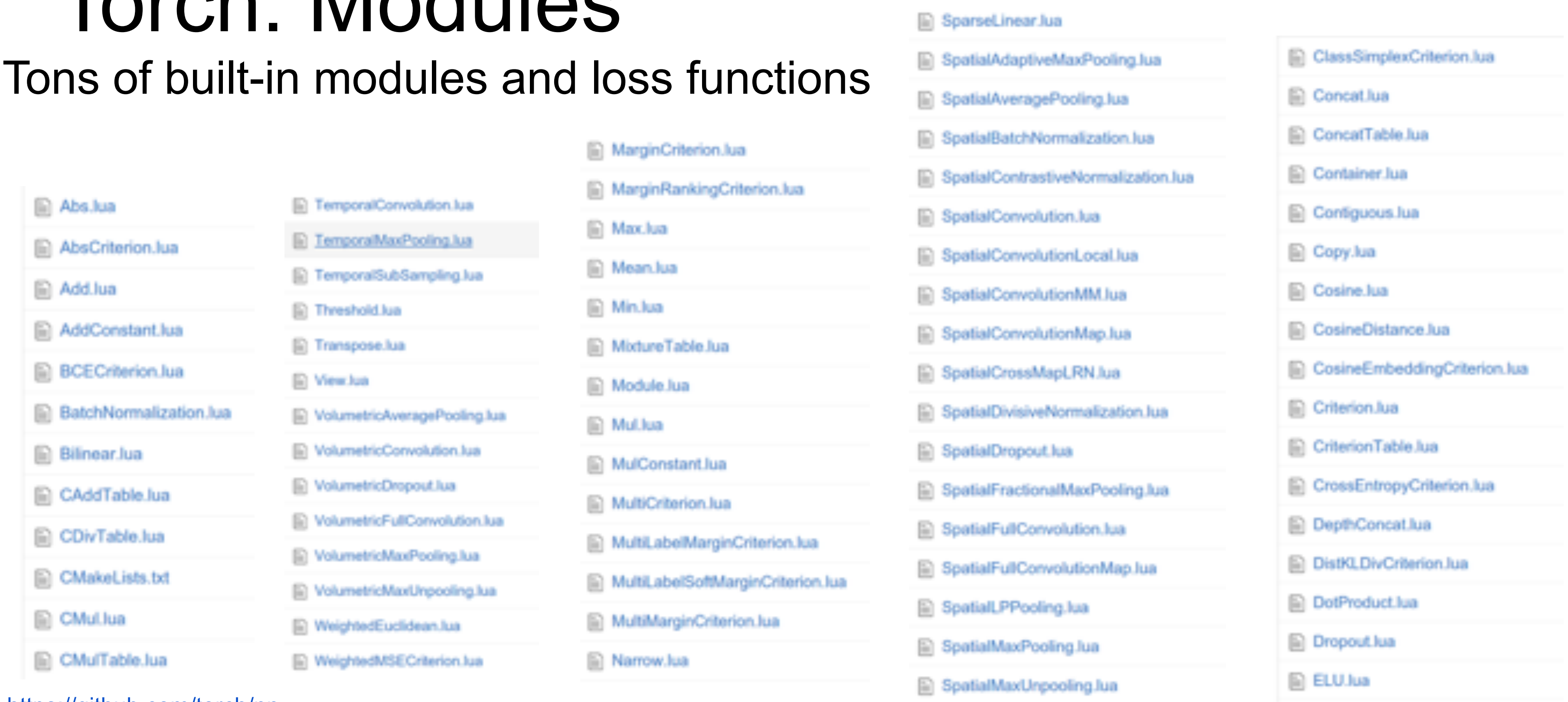
accGradParameters: Backward; compute gradient of weights

```lua
function Linear:accGradParameters(input, gradOutput, scale)
   scale = scale or 1
   if input:dim() == 1 then
      self.gradWeight:addr(scale, gradOutput, input)
      if self.bias then self.gradBias:add(scale, gradOutput) end
   elseif input:dim() == 2 then
      self.gradWeight:addmm(scale, gradOutput:t(), input)
      if self.bias then
         self.gradBias:addmv(scale, gradOutput:t(), self.addBuffer)
      end
   end
end
```

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

Torch: Modules

Tons of built-in modules and loss functions



https://github.com/torch/nn

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

Torch: Modules

Tons of built-in modules and loss functions

New ones all the time:

- Abs.lua
- AbsCriterion.lua
- Add.lua
- AddConstant.lua
- BCECriterion.lua
- BatchNormalization.lua
- Bilinear.lua
- CAddTable.lua
- CDivTable.lua
- CMakeLists.txt
- CMul.lua
- CMulTable.lua

- TemporalConvolution.lua
- TemporalMaxPooling.lua
- TemporalSubSampling.lua
- Threshold.lua
- Transpose.lua
- View.lua
- VolumetricAveragePooling.lua
- VolumetricConvolution.lua
- VolumetricDropout.lua
- VolumetricFullConvolution.lua
- VolumetricMaxPooling.lua
- VolumetricMaxUnpooling.lua
- WeightedEuclidean.lua
- WeightedMSECriterion.lua

- 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

https://github.com/torch/nn

Torch: Modules

Writing your own modules is easy!

TimesTwo.lua

```lua
require 'nn'

local times_two, parent = torch.class('nn.TimesTwo', 'nn.Module')


function times_two:__init()
  parent.__init(self)
end


function times_two:updateOutput(input)
  self.output:mul(input, 2)
  return self.output
end


function times_two:updateGradInput(input, gradOutput)
  self.gradInput:mul(gradOutput, 2)
  return self.gradInput
end
```

times_two_example.lua

```lua
require 'nn'

require 'TimesTwo'

local times_two = nn.TimesTwo()

local input = torch.randn(4, 5)
local output = times_two:forward(input)

print('here is input:')
print(input)

print('here is output:')
print(output)

local gradOutput = torch.randn(4, 5)
local gradInput = times_two:backward(input, gradOutput)

print('here is gradOutput:')
print(gradOutput)

print('here is gradInput')
print(gradInput)
```

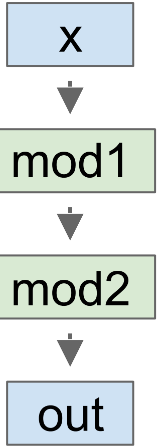
Torch: Modules

Container modules allow you to combine multiple modules

Torch: Modules

Container modules allow you to combine multiple modules

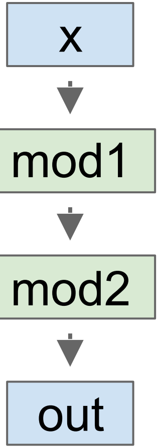
```
local seq = nn.Sequential()
seq:add(mod1)
seq:add(mod2)
local out = seq:forward(x)
```



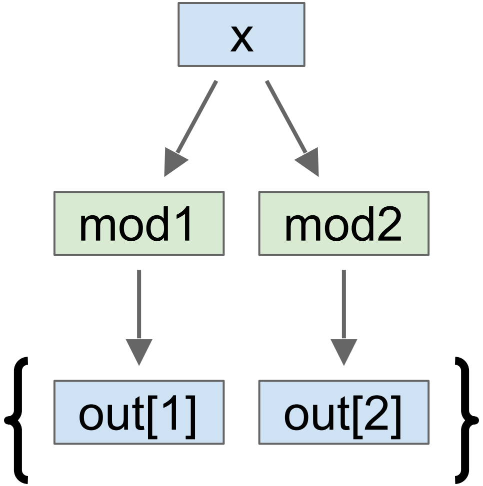
Torch: Modules

Container modules allow you to combine multiple modules

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



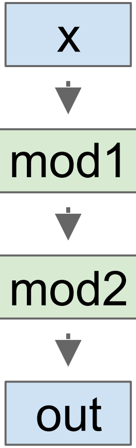

```
local concat = nn.ConcatTable()
concat:add(mod1)
concat:add(mod2)
local out = concat:forward(x)
```



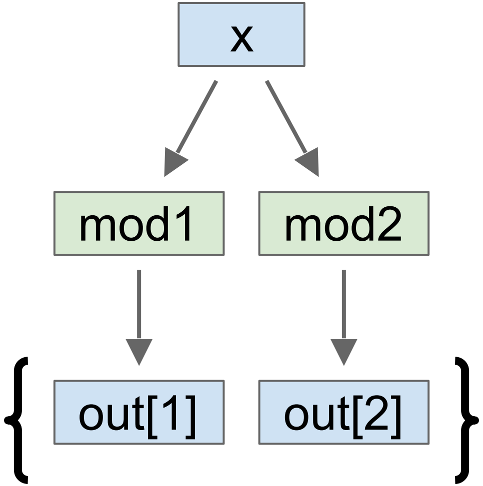
Torch: Modules

Container modules allow you to combine multiple modules

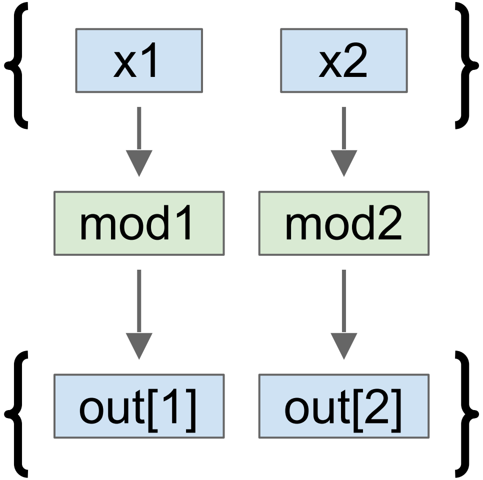
```lua
local seq = nn.Sequential()
seq:add(mod1)
seq:add(mod2)
local out = seq:forward(x)
```




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




```lua
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

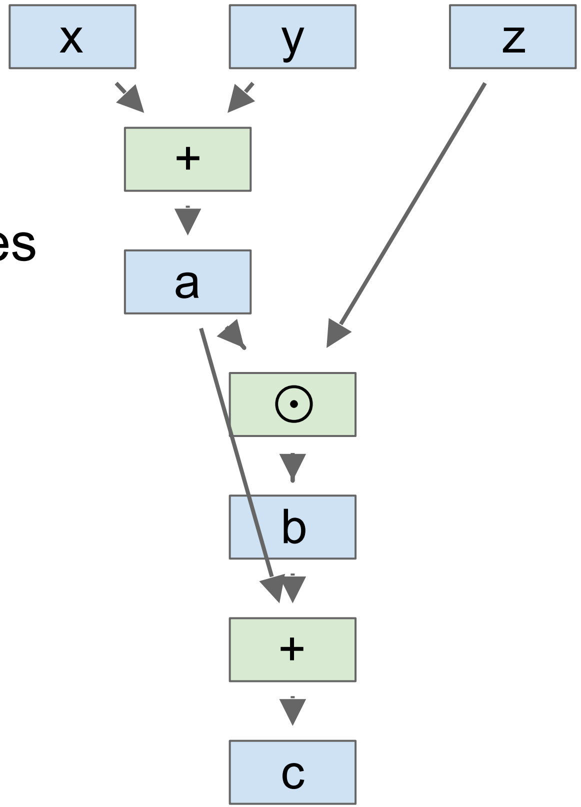
Use nngraph to build modules that combine their inputs in complex ways

Inputs: x, y, z
Outputs: c

$a = x + y$

$b = a \odot z$

$c = a + b$



Torch: nngraph

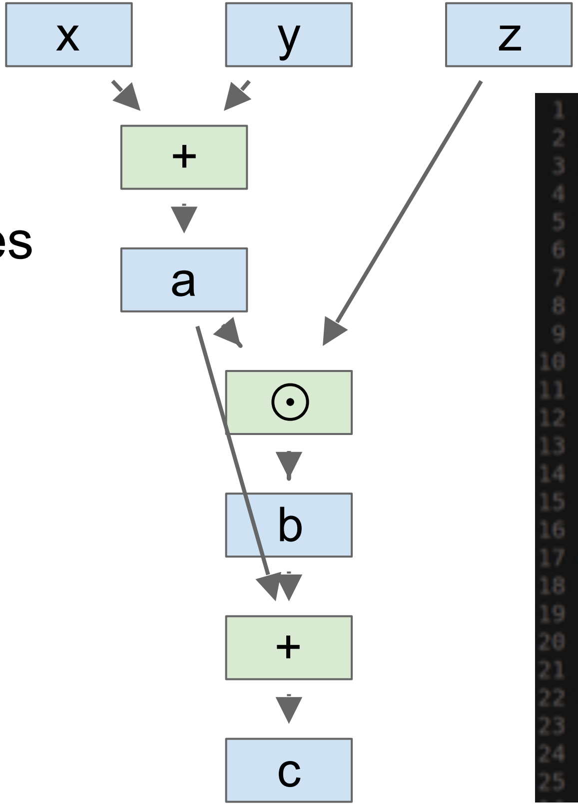
Use nngraph to build modules that combine their inputs in complex ways

Inputs: x, y, z
Outputs: c

a = x + y

b = a ⊙ z

c = a + b




```lua
require 'torch'
require 'nn'
require 'nngraph'

local function build_module()
  local x = nn.Identity()()
  local y = nn.Identity()()
  local z = nn.Identity()()

  local a = nn.CAddTable()({x, y})
  local b = nn.CMulTable()({a, z})
  local c = nn.CAddTable()({a, b})

  local inputs = {x, y, z}
  local outputs = {c}
  return nn.gModule(inputs, outputs)
end

local mod = build_module()

local x = torch.randn(4, 5)
local y = torch.randn(4, 5)
local z = torch.randn(4, 5)

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/

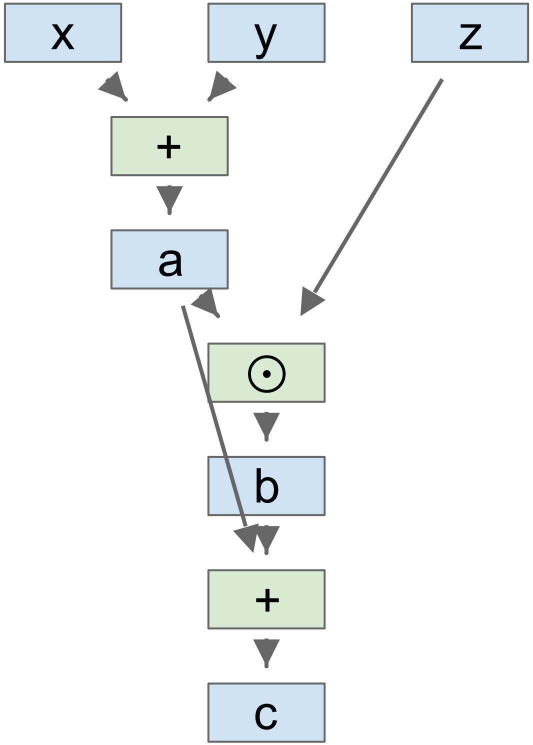
Theano Overview

From Yoshua Bengio’s group at University of Montreal

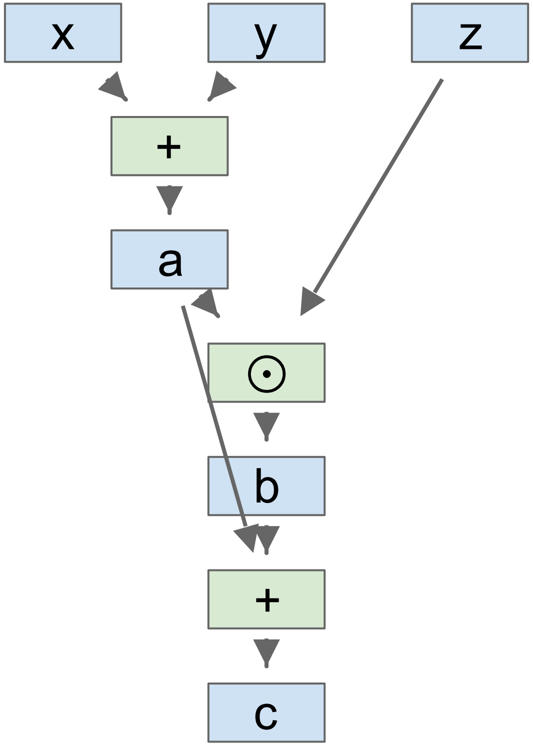
Embracing computation graphs, symbolic computation

High-level wrappers: Keras, Lasagne

Theano: Computational Graphs

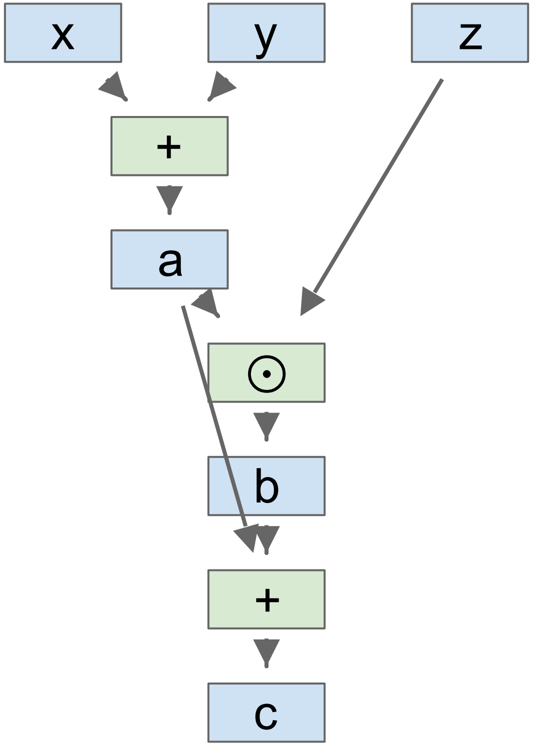


Theano: Computational Graphs




```python
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
```

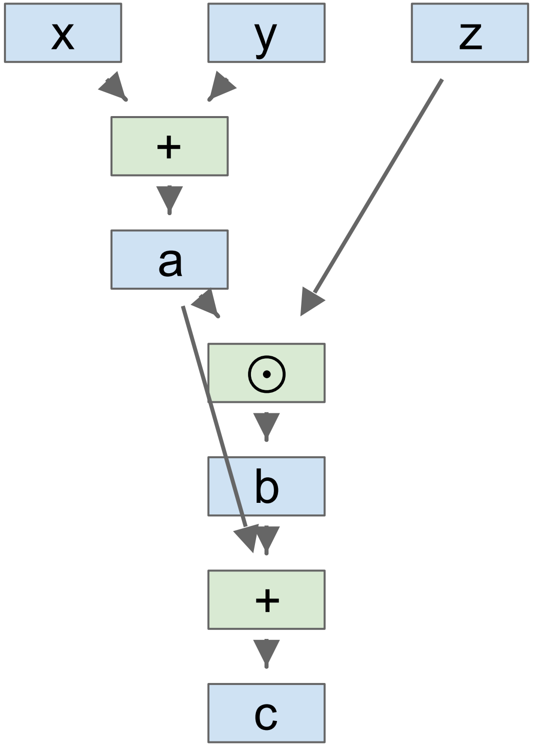
Theano: Computational Graphs




```
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

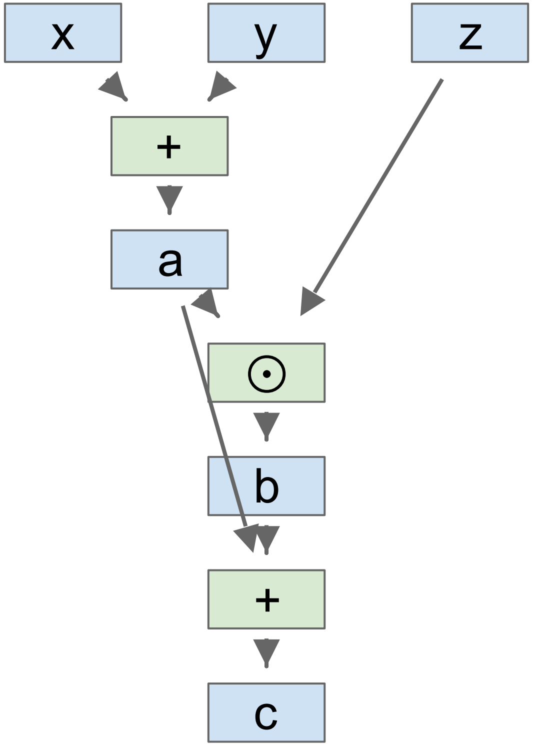
Theano: Computational Graphs




```python
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

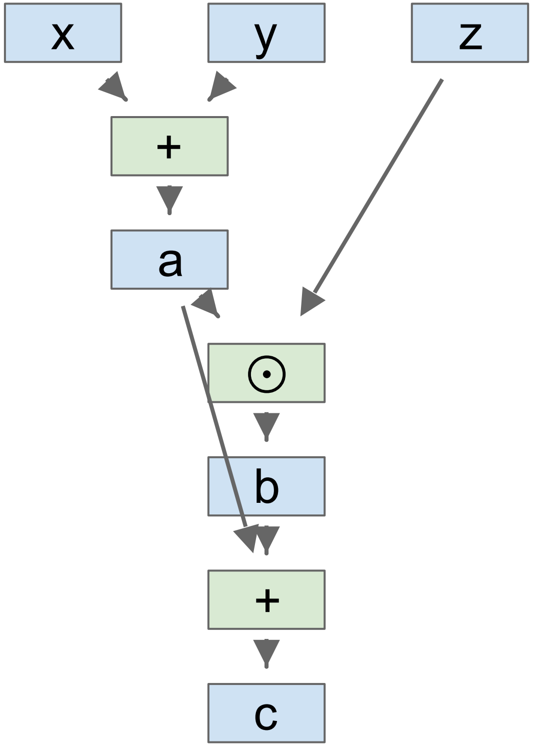
Theano: Computational Graphs




```python
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)

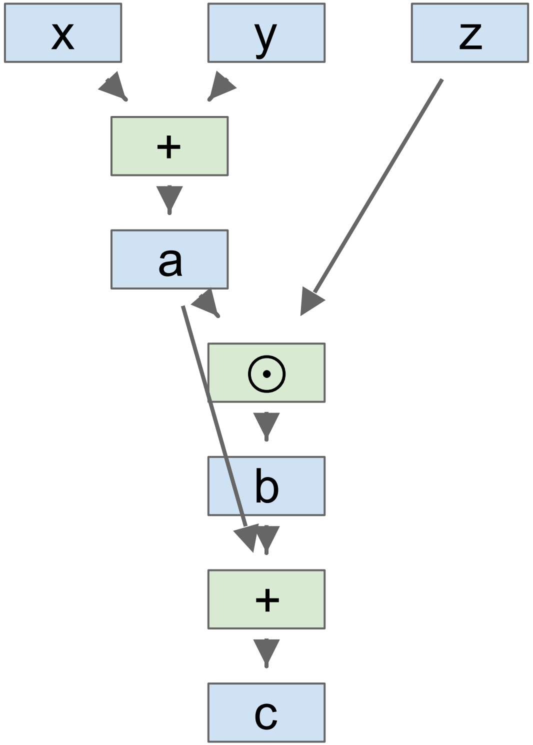
Theano: Computational Graphs




```python
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




```python
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

```python
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 = data
y = labels
w1 = first-layer weights
w2 = second-layer weights

```python
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)

```python
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

```python
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

```python
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],
    )
```

```python
# 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, dww1, dww2 = f(xx, yy, ww1, ww2)
  print loss
  ww1 -= learning_rate * dww1
  ww2 -= learning_rate * dww2
```

Use the function to perform gradient descent!

Theano: Computing Gradients

```python
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],
    )
```

```python
# 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, dww1, dww2 = f(xx, yy, ww1, ww2)
  print loss
  ww1 -= learning_rate * dww1
  ww2 -= learning_rate * dww2
```

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

```python
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

```
import numpy as np
import theano
import theano.tensor as T
import lasagne

N, D, H, C = 64, 1000, 100, 10

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

relu = lasagne.nonlinearities.rectify
softmax = lasagne.nonlinearities.softmax
net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)

probs = lasagne.layers.get_output(net)
loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()

params = lasagne.layers.get_all_params(net, trainable=True)
updates = lasagne.updates.nesterov_momentum(loss, params,
                    learning_rate=1e-2, momentum=0.0)

train_fn = theano.function([x, y], loss, updates=updates)

xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N).astype(np.int64)

for t in xrange(100):
  loss_val = train_fn(xx, yy)
  print loss_val
```

Lasagne: High Level Wrapper

Set up symbolic Theano variables for data, labels

```
import numpy as np
import theano
import theano.tensor as T
import lasagne

N, D, H, C = 64, 1000, 100, 10

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

relu = lasagne.nonlinearities.rectify
softmax = lasagne.nonlinearities.softmax
net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)

probs = lasagne.layers.get_output(net)
loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()

params = lasagne.layers.get_all_params(net, trainable=True)
updates = lasagne.updates.nesterov_momentum(loss, params,
                    learning_rate=1e-2, momentum=0.0)

train_fn = theano.function([x, y], loss, updates=updates)

xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N).astype(np.int64)

for t in xrange(100):
  loss_val = train_fn(xx, yy)
  print loss_val
```

Lasagne: High Level Wrapper

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

```python
import numpy as np
import theano
import theano.tensor as T
import lasagne

N, D, H, C = 64, 1000, 100, 10

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

relu = lasagne.nonlinearities.rectify
softmax = lasagne.nonlinearities.softmax
net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)

probs = lasagne.layers.get_output(net)
loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()

params = lasagne.layers.get_all_params(net, trainable=True)
updates = lasagne.updates.nesterov_momentum(loss, params,
                    learning_rate=1e-2, momentum=0.0)

train_fn = theano.function([x, y], loss, updates=updates)

xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N).astype(np.int64)

for t in xrange(100):
  loss_val = train_fn(xx, yy)
  print loss_val
```

Lasagne: High Level Wrapper

Forward: Use Lasagne layers to compute loss

```python
import numpy as np
import theano
import theano.tensor as T
import lasagne

N, D, H, C = 64, 1000, 100, 10

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

relu = lasagne.nonlinearities.rectify
softmax = lasagne.nonlinearities.softmax
net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)

probs = lasagne.layers.get_output(net)
loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()

params = lasagne.layers.get_all_params(net, trainable=True)
updates = lasagne.updates.nesterov_momentum(loss, params,
                learning_rate=1e-2, momentum=0.0)

train_fn = theano.function([x, y], loss, updates=updates)

xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N).astype(np.int64)

for t in xrange(100):
  loss_val = train_fn(xx, yy)
  print loss_val
```

Lasagne: High Level Wrapper

Lasagne gets parameters, and writes the update rule for you

```
import numpy as np
import theano
import theano.tensor as T
import lasagne

N, D, H, C = 64, 1000, 100, 10

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

relu = lasagne.nonlinearities.rectify
softmax = lasagne.nonlinearities.softmax
net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)

probs = lasagne.layers.get_output(net)
loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()

params = lasagne.layers.get_all_params(net, trainable=True)
updates = lasagne.updates.nesterov_momentum(loss, params,
                    learning_rate=1e-2, momentum=0.0)

train_fn = theano.function([x, y], loss, updates=updates)

xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N).astype(np.int64)

for t in xrange(100):
  loss_val = train_fn(xx, yy)
  print loss_val
```

Lasagne: High Level Wrapper

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

```
import numpy as np
import theano
import theano.tensor as T
import lasagne

N, D, H, C = 64, 1000, 100, 10

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

relu = lasagne.nonlinearities.rectify
softmax = lasagne.nonlinearities.softmax
net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)

probs = lasagne.layers.get_output(net)
loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()

params = lasagne.layers.get_all_params(net, trainable=True)
updates = lasagne.updates.nesterov_momentum(loss, params,
                          learning_rate=1e-2, momentum=0.0)

train_fn = theano.function([x, y], loss, updates=updates)

xx = np.random.randn(N, D)
yy = np.random.randint(C, size=N).astype(np.int64)

for t in xrange(100):
  loss_val = train_fn(xx, yy)
  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 :(

```python
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

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

```python
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, C])

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))

a = tf.matmul(x, w1)
a_relu = tf.nn.relu(a)
scores = tf.matmul(a_relu, w2)
probs = tf.nn.softmax(scores)
loss = -tf.reduce_sum(y * tf.log(probs))

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1

with tf.Session() as sess:
  sess.run(tf.initialize_all_variables())

  for t in xrange(100):
    _, loss_value = sess.run([train_step, loss],
                             feed_dict={x: xx, y: yy})
    print loss_value
```

TensorFlow: Two-Layer Net

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

```python
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, C])

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))

a = tf.matmul(x, w1)
a_relu = tf.nn.relu(a)
scores = tf.matmul(a_relu, w2)
probs = tf.nn.softmax(scores)
loss = -tf.reduce_sum(y * tf.log(probs))

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1

with tf.Session() as sess:
  sess.run(tf.initialize_all_variables())

  for t in xrange(100):
    _, loss_value = sess.run([train_step, loss],
                             feed_dict={x: xx, y: yy})
    print loss_value
```

TensorFlow: Two-Layer Net

Create Variables to hold weights; similar to Theano shared variables

Initialize variables with numpy arrays

```
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, C])

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))

a = tf.matmul(x, w1)
a_relu = tf.nn.relu(a)
scores = tf.matmul(a_relu, w2)
probs = tf.nn.softmax(scores)
loss = -tf.reduce_sum(y * tf.log(probs))

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1

with tf.Session() as sess:
  sess.run(tf.initialize_all_variables())

  for t in xrange(100):
    _, loss_value = sess.run([train_step, loss],
                             feed_dict={x: xx, y: yy})
    print loss_value
```

TensorFlow: Two-Layer Net

Forward: Compute scores, probs, loss (symbolically)

```
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, C])

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))

a = tf.matmul(x, w1)
a_relu = tf.nn.relu(a)
scores = tf.matmul(a_relu, w2)
probs = tf.nn.softmax(scores)
loss = -tf.reduce_sum(y * tf.log(probs))

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1

with tf.Session() as sess:
  sess.run(tf.initialize_all_variables())

  for t in xrange(100):
    _, loss_value = sess.run([train_step, loss],
                             feed_dict={x: xx, y: yy})
    print loss_value
```

TensorFlow: Two-Layer Net

Running train_step will use SGD to minimize loss

```python
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, C])

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))

a = tf.matmul(x, w1)
a_relu = tf.nn.relu(a)
scores = tf.matmul(a_relu, w2)
probs = tf.nn.softmax(scores)
loss = -tf.reduce_sum(y * tf.log(probs))

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1

with tf.Session() as sess:
  sess.run(tf.initialize_all_variables())

  for t in xrange(100):
    _, loss_value = sess.run([train_step, loss],
                             feed_dict={x: xx, y: yy})
    print loss_value
```

TensorFlow: Two-Layer Net

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

```python
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, C])

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))

a = tf.matmul(x, w1)
a_relu = tf.nn.relu(a)
scores = tf.matmul(a_relu, w2)
probs = tf.nn.softmax(scores)
loss = -tf.reduce_sum(y * tf.log(probs))

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1

with tf.Session() as sess:
  sess.run(tf.initialize_all_variables())

  for t in xrange(100):
    _, loss_value = sess.run([train_step, loss],
                             feed_dict={x: xx, y: yy})
    print loss_value
```

TensorFlow: Two-Layer Net

Actually train the model

```python
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, C])

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))

a = tf.matmul(x, w1)
a_relu = tf.nn.relu(a)
scores = tf.matmul(a_relu, w2)
probs = tf.nn.softmax(scores)
loss = -tf.reduce_sum(y * tf.log(probs))

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1

with tf.Session() as sess:
  sess.run(tf.initialize_all_variables())

  for t in xrange(100):
    _, loss_value = sess.run([train_step, loss],
                             feed_dict={x: xx, y: yy})
    print loss_value
```

TensorFlow: Tensorboard

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

```python
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, C])

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))

a = tf.matmul(x, w1)
a_relu = tf.nn.relu(a)
scores = tf.matmul(a_relu, w2)
probs = tf.nn.softmax(scores)
loss = -tf.reduce_sum(y * tf.log(probs))

loss_summary = tf.scalar_summary('loss', loss)
w1_hist = tf.histogram_summary('w1', w1)
w2_hist = tf.histogram_summary('w2', w2)

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1

with tf.Session() as sess:
  merged = tf.merge_all_summaries()
  writer = tf.train.SummaryWriter('/tmp/fc_logs', sess.graph_def)
  sess.run(tf.initialize_all_variables())

  for t in xrange(100):
    summary_str, _, loss_value = sess.run(
                                   [merged, train_step, loss],
                                   feed_dict={x: xx, y: yy})
    writer.add_summary(summary_str, t)
    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

```
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, C])

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))

a = tf.matmul(x, w1)
a_relu = tf.nn.relu(a)
scores = tf.matmul(a_relu, w2)
probs = tf.nn.softmax(scores)
loss = -tf.reduce_sum(y * tf.log(probs))

loss_summary = tf.scalar_summary('loss', loss)
w1_hist = tf.histogram_summary('w1', w1)
w2_hist = tf.histogram_summary('w2', w2)

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1

with tf.Session() as sess:
  merged = tf.merge_all_summaries()
  writer = tf.train.SummaryWriter('/tmp/fc_logs', sess.graph_def)
  sess.run(tf.initialize_all_variables())

  for t in xrange(100):
    summary_str, _, loss_value = sess.run(
                                [merged, train_step, loss],
                                feed_dict={x: xx, y: yy})
    writer.add_summary(summary_str, t)
    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

```python
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, C])

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))

a = tf.matmul(x, w1)
a_relu = tf.nn.relu(a)
scores = tf.matmul(a_relu, w2)
probs = tf.nn.softmax(scores)
loss = -tf.reduce_sum(y * tf.log(probs))

loss_summary = tf.scalar_summary('loss', loss)
w1_hist = tf.histogram_summary('w1', w1)
w2_hist = tf.histogram_summary('w2', w2)

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1

with tf.Session() as sess:
  merged = tf.merge_all_summaries()
  writer = tf.train.SummaryWriter('/tmp/fc_logs', sess.graph_def)
  sess.run(tf.initialize_all_variables())

  for t in xrange(100):
    summary_str, _, loss_value = sess.run(
                                  [merged, train_step, loss],
                                  feed_dict={x: xx, y: yy})
    writer.add_summary(summary_str, t)
    print loss_value
```

TensorFlow: Tensorboard

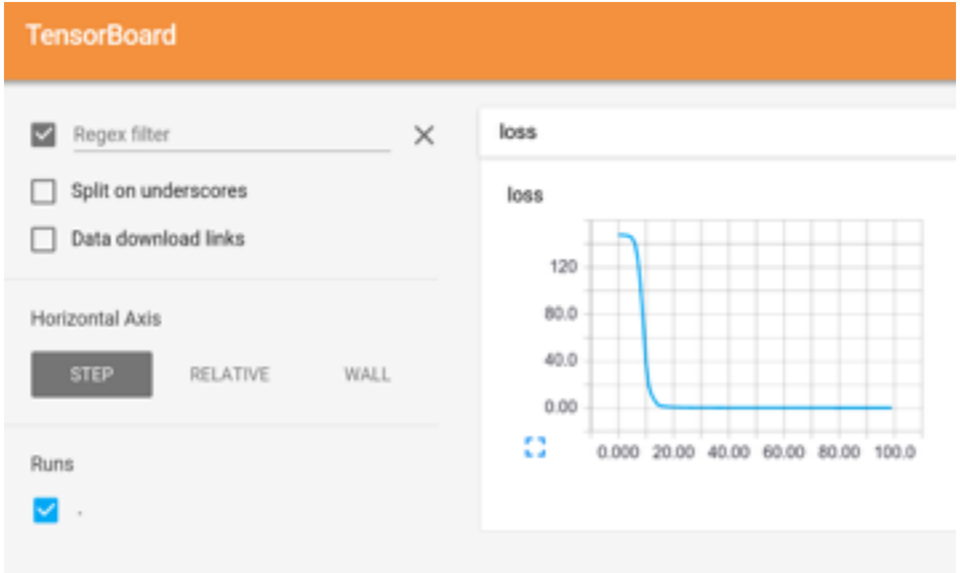
Tensorboard makes it easy to visualize what's happening inside your models

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

```
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D])
y = tf.placeholder(tf.float32, shape=[None, C])

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))

a = tf.matmul(x, w1)
a_relu = tf.nn.relu(a)
scores = tf.matmul(a_relu, w2)
probs = tf.nn.softmax(scores)
loss = -tf.reduce_sum(y * tf.log(probs))

loss_summary = tf.scalar_summary('loss', loss)
w1_hist = tf.histogram_summary('w1', w1)
w2_hist = tf.histogram_summary('w2', w2)

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1

with tf.Session() as sess:
  merged = tf.merge_all_summaries()
  writer = tf.train.SummaryWriter('/tmp/fc_logs', sess.graph_def)
  sess.run(tf.initialize_all_variables())

  for t in xrange(100):
    summary_str, _, loss_value = sess.run(
                                  [merged, train_step, loss],
                                  feed_dict={x: xx, y: yy})
    writer.add_summary(summary_str, t)
    print loss_value
```

TensorFlow: Tensorboard

Start Tensorboard server, and we get graphs!



TensorFlow: TensorBoard

```python
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D], name='x')
y = tf.placeholder(tf.float32, shape=[None, C], name='y')

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32), name='w1')
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32), name='w2')

with tf.name_scope('scores') as scope:
  a = tf.matmul(x, w1)
  a_relu = tf.nn.relu(a)
  scores = tf.matmul(a_relu, w2)
with tf.name_scope('loss') as scope:
  probs = tf.nn.softmax(scores)
  loss = -tf.reduce_sum(y * tf.log(probs))

loss_summary = tf.scalar_summary('loss', loss)
w1_hist = tf.histogram_summary('w1', w1)
w2_hist = tf.histogram_summary('w2', w2)

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1
```

TensorFlow: TensorBoard

Add names to placeholders and variables

```python
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D], name='x')
y = tf.placeholder(tf.float32, shape=[None, C], name='y')

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32), name='w1')
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32), name='w2')

with tf.name_scope('scores') as scope:
  a = tf.matmul(x, w1)
  a_relu = tf.nn.relu(a)
  scores = tf.matmul(a_relu, w2)
with tf.name_scope('loss') as scope:
  probs = tf.nn.softmax(scores)
  loss = -tf.reduce_sum(y * tf.log(probs))

loss_summary = tf.scalar_summary('loss', loss)
w1_hist = tf.histogram_summary('w1', w1)
w2_hist = tf.histogram_summary('w2', w2)

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1
```

TensorFlow: TensorBoard

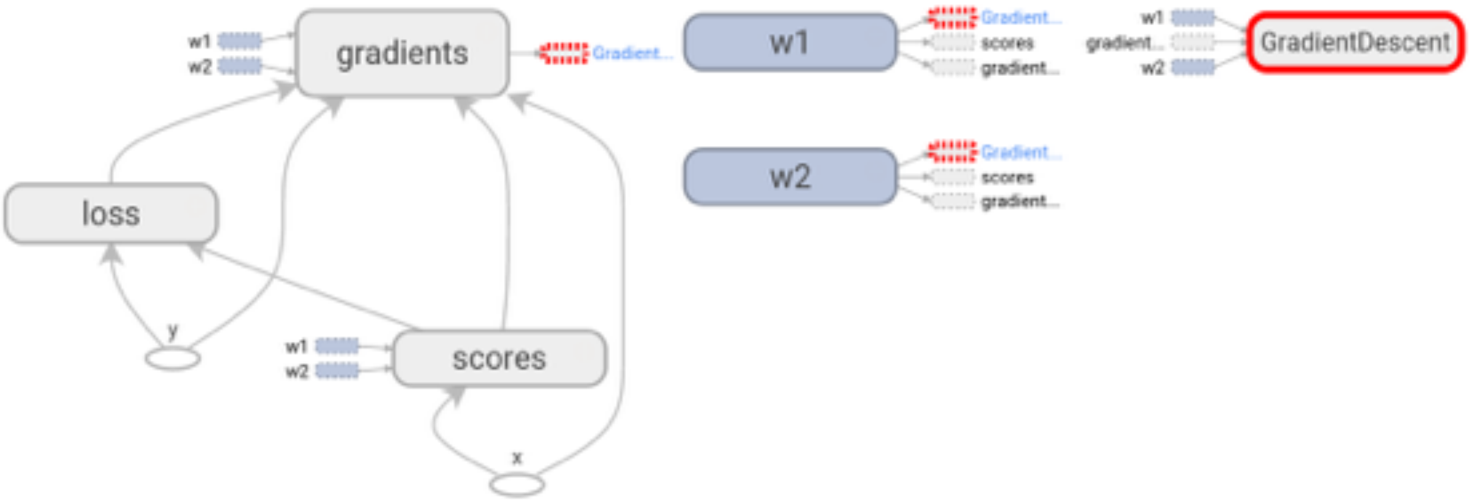
Add names to placeholders and variables

Break up the forward pass with name scoping

```python
import tensorflow as tf
import numpy as np

N, D, H, C = 64, 1000, 100, 10

x = tf.placeholder(tf.float32, shape=[None, D], name='x')
y = tf.placeholder(tf.float32, shape=[None, C], name='y')

w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32), name='w1')
w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32), name='w2')

with tf.name_scope('scores') as scope:
  a = tf.matmul(x, w1)
  a_relu = tf.nn.relu(a)
  scores = tf.matmul(a_relu, w2)
with tf.name_scope('loss') as scope:
  probs = tf.nn.softmax(scores)
  loss = -tf.reduce_sum(y * tf.log(probs))

loss_summary = tf.scalar_summary('loss', loss)
w1_hist = tf.histogram_summary('w1', w1)
w2_hist = tf.histogram_summary('w2', w2)

learning_rate = 1e-2
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

xx = np.random.randn(N, D).astype(np.float32)
yy = np.zeros((N, C)).astype(np.float32)
yy[np.arange(N), np.random.randint(C, size=N)] = 1
```

TensorFlow: TensorBoard

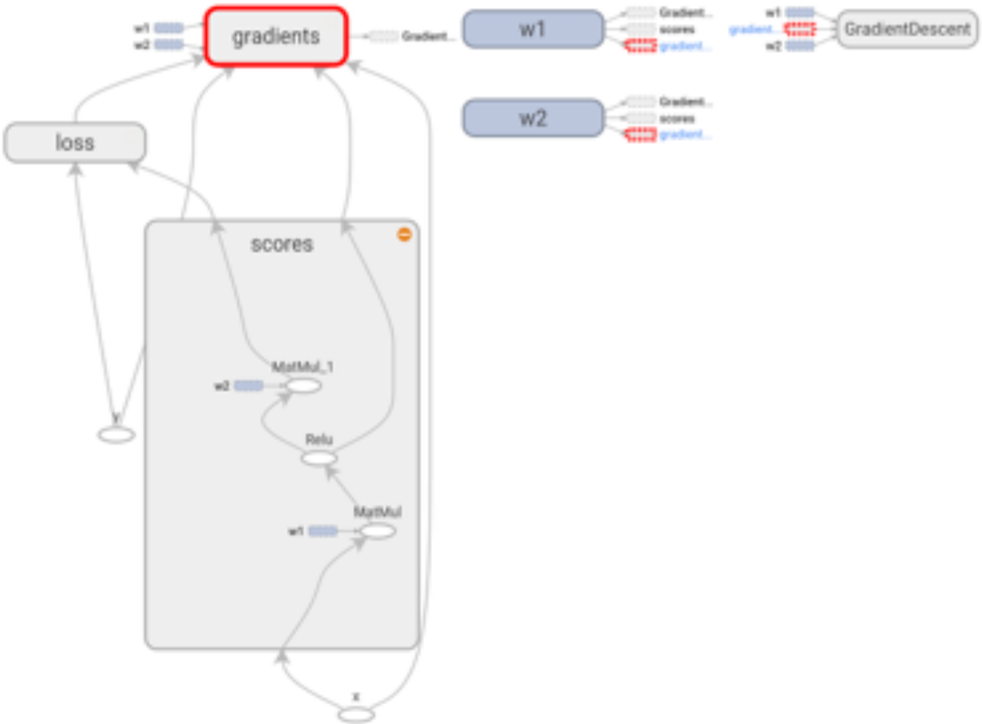
Tensorboard shows the graph!



TensorFlow: TensorBoard

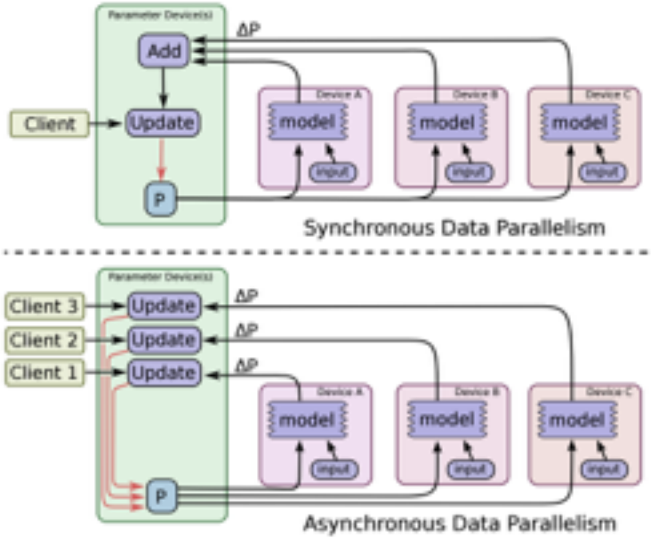
Tensorboard shows the graph!

Name scopes expand to show individual operations



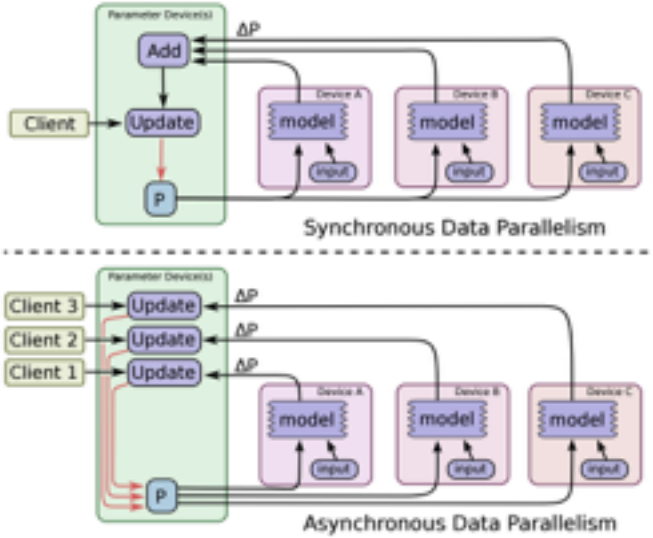
TensorFlow: Multi-GPU

Data parallelism:
synchronous or asynchronous

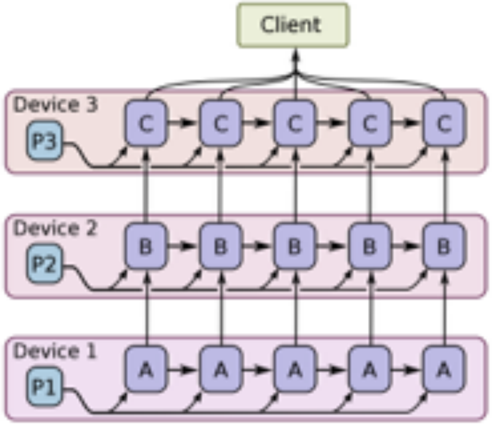


TensorFlow: Multi-GPU

Data parallelism:
synchronous or asynchronous

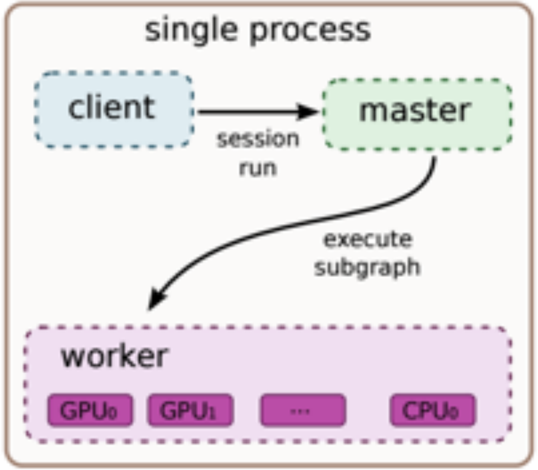


Model parallelism:
Split model across GPUs

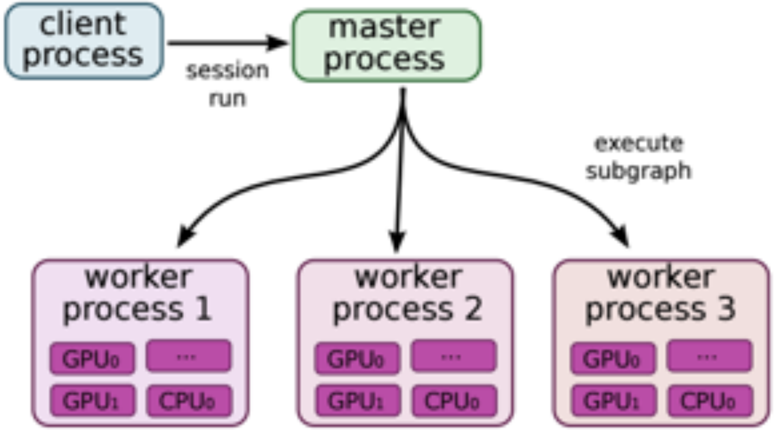


TensorFlow: Distributed

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

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

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

```
  const Dtype* cpu_data() const;
  void set_cpu_data(Dtype* data);
  const int* gpu_shape() const;
  const Dtype* gpu_data() const;
  const Dtype* cpu_diff() const;
  const Dtype* gpu_diff() const;
  Dtype* mutable_cpu_data();
  Dtype* mutable_gpu_data();
  Dtype* mutable_cpu_diff();
  Dtype* mutable_gpu_diff();
```

```
 protected:
  shared_ptr<SyncedMemory> data_;
  shared_ptr<SyncedMemory> diff_;
  shared_ptr<SyncedMemory> shape_data_;
  vector<int> shape_;
  int count_;
  int capacity_;
```

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

Caffe: Blobs

N-dimensional array for storing activations and weights

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

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

```
  const Dtype* cpu_data() const;
  void set_cpu_data(Dtype* data);
  const int* gpu_shape() const;
  const Dtype* gpu_data() const;
  const Dtype* cpu_diff() const;
  const Dtype* gpu_diff() const;
  Dtype* mutable_cpu_data();
  Dtype* mutable_gpu_data();
  Dtype* mutable_cpu_diff();
  Dtype* mutable_gpu_diff();
```

```
 protected:
  shared_ptr<SyncedMemory> data_;
  shared_ptr<SyncedMemory> diff_;
  shared_ptr<SyncedMemory> shape_data_;
  vector<int> shape_;
  int count_;
  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

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

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

  const Dtype* cpu_data() const;
  void set_cpu_data(Dtype* data);
  const int* gpu_shape() const;
  const Dtype* gpu_data() const;
  const Dtype* cpu_diff() const;
  const Dtype* gpu_diff() const;
  Dtype* mutable_cpu_data();
  Dtype* mutable_gpu_data();
  Dtype* mutable_cpu_diff();
  Dtype* mutable_gpu_diff();

 protected:
  shared_ptr<SyncedMemory> data_;
  shared_ptr<SyncedMemory> diff_;
  shared_ptr<SyncedMemory> shape_data_;
  vector<int> shape_;
  int count_;
  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

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

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

```
  const Dtype* cpu_data() const;
  void set_cpu_data(Dtype* data);
  const int* gpu_shape() const;
  const Dtype* gpu_data() const;
  const Dtype* cpu_diff() const;
  const Dtype* gpu_diff() const;
  Dtype* mutable_cpu_data();
  Dtype* mutable_gpu_data();
  Dtype* mutable_cpu_diff();
  Dtype* mutable_gpu_diff();
```

```
 protected:
  shared_ptr<SyncedMemory> data_;
  shared_ptr<SyncedMemory> diff_;
  shared_ptr<SyncedMemory> shape_data_;
  vector<int> shape_;
  int count_;
  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

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

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

```
  const Dtype* cpu_data() const;
  void set_cpu_data(Dtype* data);
  const int* gpu_shape() const;
  const Dtype* gpu_data() const;
  const Dtype* cpu_diff() const;
  const Dtype* gpu_diff() const;
  Dtype* mutable_cpu_data();
  Dtype* mutable_gpu_data();
  Dtype* mutable_cpu_diff();
  Dtype* mutable_gpu_diff();
```

```
 protected:
  shared_ptr<SyncedMemory> data_;
  shared_ptr<SyncedMemory> diff_;
  shared_ptr<SyncedMemory> shape_data_;
  vector<int> shape_;
  int count_;
  int capacity_;
```

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

Caffe: Layer

A small unit of computation

```
template <typename Dtype>
class Layer {
 public:

  /** @brief Using the CPU device, compute the layer output. */
  virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) = 0;
  /**
   * @brief Using the GPU device, compute the layer output.
   *        Fall back to Forward_cpu() if unavailable.
   */
  virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
    // LOG(WARNING) << "Using CPU code as backup.";
    return Forward_cpu(bottom, top);
  }

  /**
   * @brief Using the CPU device, compute the gradients for any parameters and
   *        for the bottom blobs if propagate_down is true.
   */
  virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down,
      const vector<Blob<Dtype>*>& bottom) = 0;
  /**
   * @brief Using the GPU device, compute the gradients for any parameters and
   *        for the bottom blobs if propagate_down is true.
   *        Fall back to Backward_cpu() if unavailable.
   */
  virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down,
      const vector<Blob<Dtype>*>& bottom) {
    // LOG(WARNING) << "Using CPU code as backup.";
    Backward_cpu(top, propagate_down, bottom);
  }
```

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

```
template <typename Dtype>
class Layer {
 public:

  /** @brief Using the CPU device, compute the layer output. */
  virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) = 0;
  /**
   * @brief Using the GPU device, compute the layer output.
   *        Fall back to Forward_cpu() if unavailable.
   */
  virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
    // LOG(WARNING) << "Using CPU code as backup.";
    return Forward_cpu(bottom, top);
  }

  /**
   * @brief Using the CPU device, compute the gradients for any parameters and
   *        for the bottom blobs if propagate_down is true.
   */
  virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down,
      const vector<Blob<Dtype>*>& bottom) = 0;
  /**
   * @brief Using the GPU device, compute the gradients for any parameters and
   *        for the bottom blobs if propagate_down is true.
   *        Fall back to Backward_cpu() if unavailable.
   */
  virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down,
      const vector<Blob<Dtype>*>& bottom) {
    // LOG(WARNING) << "Using CPU code as backup.";
    Backward_cpu(top, propagate_down, bottom);
  }
```

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

```
template <typename Dtype>
class Layer {
 public:

  /** @brief Using the CPU device, compute the layer output. */
  virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) = 0;
  /**
   * @brief Using the GPU device, compute the layer output.
   *        Fall back to Forward_cpu() if unavailable.
   */
  virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
    // LOG(WARNING) << "Using CPU code as backup.";
    return Forward_cpu(bottom, top);
  }

  /**
   * @brief Using the CPU device, compute the gradients for any parameters and
   *        for the bottom blobs if propagate_down is true.
   */
  virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down,
      const vector<Blob<Dtype>*>& bottom) = 0;
  /**
   * @brief Using the GPU device, compute the gradients for any parameters and
   *        for the bottom blobs if propagate_down is true.
   *        Fall back to Backward_cpu() if unavailable.
   */
  virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down,
      const vector<Blob<Dtype>*>& bottom) {
    // LOG(WARNING) << "Using CPU code as backup.";
    Backward_cpu(top, propagate_down, bottom);
  }
```

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

```
template <typename Dtype>
class Layer {
 public:

  /** @brief Using the CPU device, compute the layer output. */
  virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) = 0;
  /**
   * @brief Using the GPU device, compute the layer output.
   *        Fall back to Forward_cpu() if unavailable.
   */
  virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
    // LOG(WARNING) << "Using CPU code as backup.";
    return Forward_cpu(bottom, top);
  }

  /**
   * @brief Using the CPU device, compute the gradients for any parameters and
   *        for the bottom blobs if propagate_down is true.
   */
  virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down,
      const vector<Blob<Dtype>*>& bottom) = 0;
  /**
   * @brief Using the GPU device, compute the gradients for any parameters and
   *        for the bottom blobs if propagate_down is true.
   *        Fall back to Backward_cpu() if unavailable.
   */
  virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down,
      const vector<Blob<Dtype>*>& bottom) {
    // LOG(WARNING) << "Using CPU code as backup.";
    Backward_cpu(top, propagate_down, bottom);
  }
```

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

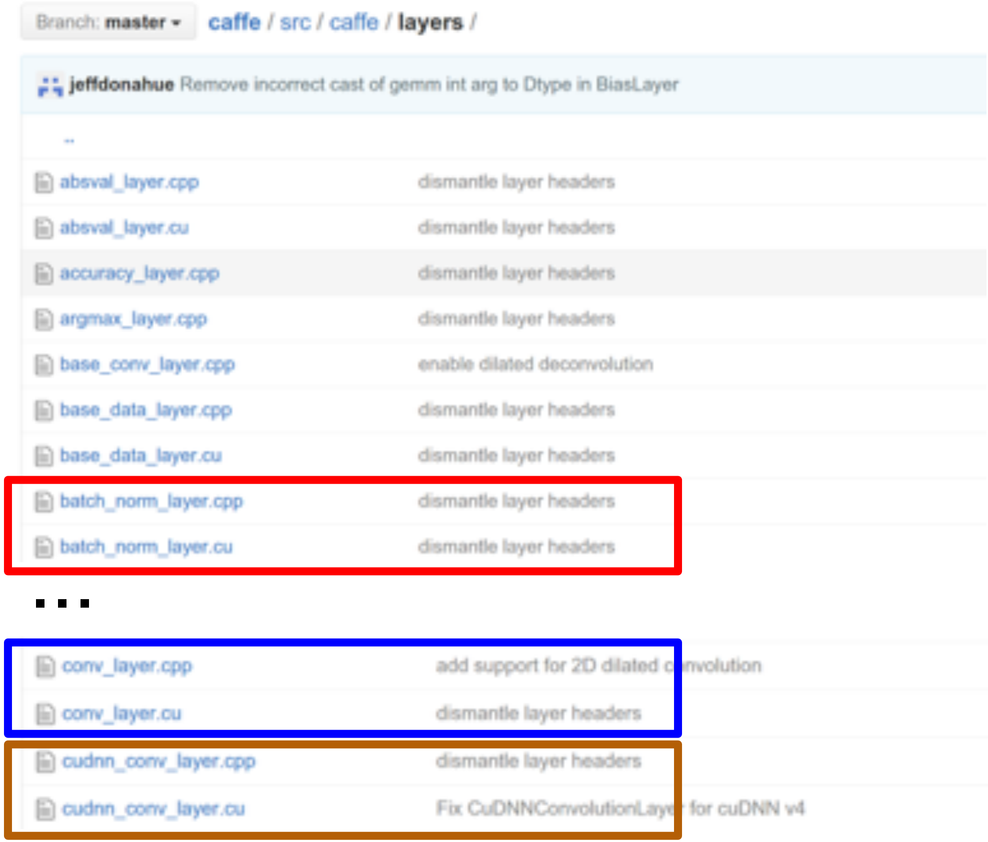
https://github.com/BVLC/caffe/tree/master/src/caffe/layers

Caffe: Layer

Tons of different layer types:

- **batch norm**
- **convolution**
- **cuDNN convolution**

.cpp: CPU implementation
.cu: GPU implementation



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**

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

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

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

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

Caffe: Solver

```
template <typename Dtype>
class Solver {
 public:

  // The main entry of the solver function. In default, iter will be zero. Pass
  // in a non-zero iter number to resume training for a pre-trained net.
  virtual void Solve(const char* resume_file = NULL);
  inline void Solve(const string resume_file) { Solve(resume_file.c_str()); }
  void Step(int iters);
  // The Restore method simply dispatches to one of the
  // RestoreSolverStateFrom___ protected methods. You should implement these
  // methods to restore the state from the appropriate snapshot type.
  void Restore(const char* resume_file);
  // The Solver::Snapshot function implements the basic snapshotting utility
  // that stores the learned net. You should implement the SnapshotSolverState()
  // function that produces a SolverState protocol buffer that needs to be
  // written to disk together with the learned net.
  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

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

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

Subclasses implement different update rules

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

```
template <typename Dtype>
class SGDSolver : public Solver<Dtype> {
template <typename Dtype>
class RMSPropSolver : public SGDSolver<Dtype> {
template <typename Dtype>
class AdamSolver : public SGDSolver<Dtype> {
```

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