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

```
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

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

.prototxt file

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

https://developers.google.com/protocol-buffers/
"Typed JSON" from Google

Define "message types" in .proto files

Serialize instances to text files (.prototxt)

Compile classes for different languages

https://developers.google.com/protocol-buffers/
Caffe: Protocol Buffers

<- 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

```cpp
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
  weight_decay: 1
}
layers {
  bottom: "fc1"
top: "loss"
name: "loss"
type: SOFTMAX_LOSS
  weight_decay: 0
}
inner_product_param {
  num_output: 2
  weight_filler {
    type: "gaussian"
    std: 0.01
  }
  bias_filler {
    type: "constant"
    value: 0
  }
}
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
    }
  }
}
```
Caffe Step 2: Define Net

Layers and Blobs often have same name!

```
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
  }
}
```

Learning rates (weight + bias)

Regularization (weight + bias)

Number of output classes
Caffe Step 2: Define Net

```cpp
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

Caffe Step 2: Define Net (finetuning)

**Original prototxt:**
```protobuf
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:**
```protobuf
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
  }
}

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:

```protobuf
define the original prototxt...
```

Modified prototxt:

```protobuf
define the modified prototxt...
```

Same name: weights copied

- Pretrained weights:
  - fc7.weight: [values]
  - fc7.bias: [values]

- Different name: weights reinitialized
  - fc8.weight: [values]
  - fc8.bias: [values]
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

```python
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!

```bash
./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 \n  -model path/to/trainval.prototxt \n  -solver path/to/solver.prototxt \n  -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 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

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)
```

```python
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
local a:max(0)  -- In-place ReLU
local scores = torch.mm(a, w2)  -- Second layer

print(scores)
```
Like numpy, can easily change data type:
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
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)
```

```python
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
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: \textit{nn}

\textit{nn} module lets you easily build and train neural nets

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

nn module lets you easily build and train neural nets

Build a two-layer ReLU net
Torch: nn

nn module lets you easily build and train neural nets

Get weights and gradient for entire network
Torch: nn

nn module lets you easily build and train neural nets

Use a softmax loss function
Torch: nn

nn module lets you easily build and train neural nets

Generate random data

---

```python
1 import torch
2
3 N, D, H, C = 100, 1000, 100, 10
4
5 net = nn.Sequential()
6 net.add(nn.Linear(D, H))
7 net.add(nn.ReLU())
8 net.add(nn.Linear(H, C))
9
10 weights, grad_weights = net.getParameters()
11
12 crit = nn.CrossEntropyCriterion() -- Softmax loss
13
14 x = torch.randn(N, D)
15 y = torch.randn(N, C)
16
17 scores = net(x)
18 loss = crit(scores, y)
19
20 grad_weights.zero()
21 d_scores = crit.backward(scores, y)
22 dx = net.backward(x, d_scores)
23
24 learning_rate = 1e-3
25 weights.add(-learning_rate, grad_weights)
```
Torch: nn

nn module lets you easily build and train neural nets

**Forward pass**: compute scores and loss
Torch: nn

nn module lets you easily build and train neural nets

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

nn module lets you easily build and train neural nets

**Update:** Make a gradient descent step

```python
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 "criteria"
local crit = nn.CrossEntropyCriterion() -- Softmax loss

-- Generate some random input data
local x = torch.randn(N, D)
local y = torch.randn(N, 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)
```
Running on GPU is easy:
Torch: cunn

Running on GPU is easy:

Import a few new packages

```c
require 'torch'
require 'cutorch'
require 'nn'
require 'cunn'
```
Torch: cunn

Running on GPU is easy:

Import a few new packages

Cast network and criterion
Torch: cunn

Running on GPU is easy:

Import a few new packages

Cast network and criterion

Cast data and labels
Torch: optim

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

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

Import optim package
Torch: optim

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

Import optim package

Write a callback function that returns loss and gradients

```
1 require 'torch'
2 require 'nn'
3 require 'optim'

-- 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))
net:add(nn.Linear(C, N))

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

-- Loss functions are called " criterions "
nlocal 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 d_scores = crit:backward(scores, y)
    local dx = net:backward(x, d_scores)
    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

```python
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

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

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

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

Caffe has Nets and Layers; Torch just has Modules

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

**accGradParameters**: Backward; compute gradient of weights

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

Tons of built-in modules and loss functions

<table>
<thead>
<tr>
<th>Module</th>
<th>Module</th>
<th>Module</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs.lua</td>
<td>AbsCriterion.lua</td>
<td>Add.lua</td>
<td>AddConstant.lua</td>
</tr>
<tr>
<td>Add.lua</td>
<td>AddCriterion.lua</td>
<td>BCCECriterion.lua</td>
<td>BatchNormalization.lua</td>
</tr>
<tr>
<td>Bilinear.lua</td>
<td>CAddTable.lua</td>
<td>CDivTable.lua</td>
<td>CMakeLists.txt</td>
</tr>
<tr>
<td>CMul.lua</td>
<td>CMulTable.lua</td>
<td>ClassSimplexCriterion.lua</td>
<td>Concat.lua</td>
</tr>
<tr>
<td>ConcatTable.lua</td>
<td>Container.lua</td>
<td>Contiguous.lua</td>
<td>Copy.lua</td>
</tr>
<tr>
<td>Cosine.lua</td>
<td>CosineDistance.lua</td>
<td>CosineEmbeddingCriterion.lua</td>
<td>Criterion.lua</td>
</tr>
<tr>
<td>Criterion.lua</td>
<td>CriterionTable.lua</td>
<td>CrossEntropyCriterion.lua</td>
<td>DropOut.lua</td>
</tr>
<tr>
<td>DepthConcat.lua</td>
<td>DistKLDivCriterion.lua</td>
<td>DotProduct.lua</td>
<td>ELU.lua</td>
</tr>
<tr>
<td>ELU.lua</td>
<td>ELU.lua</td>
<td>ELU.lua</td>
<td>ELU.lua</td>
</tr>
<tr>
<td>L2Criterion.lua</td>
<td>L2Criterion.lua</td>
<td>L2Criterion.lua</td>
<td>L2Criterion.lua</td>
</tr>
<tr>
<td>MarginCriterion.lua</td>
<td>MarginCriterion.lua</td>
<td>MarginRankingCriterion.lua</td>
<td>MarginCriterion.lua</td>
</tr>
<tr>
<td>Max.lua</td>
<td>Mean.lua</td>
<td>Min.lua</td>
<td>Min.lua</td>
</tr>
<tr>
<td>Min.lua</td>
<td>Mul.lua</td>
<td>MultiCriterion.lua</td>
<td>MultiCriterion.lua</td>
</tr>
<tr>
<td>MulConstant.lua</td>
<td>MultiCriterion.lua</td>
<td>MultiLabelMarginCriterion.lua</td>
<td>MultiLabelMarginCriterion.lua</td>
</tr>
<tr>
<td>MultiLabelSoftMarginCriterion.lua</td>
<td>MultiLabelMarginCriterion.lua</td>
<td>MultiMarginCriterion.lua</td>
<td>MultiMarginCriterion.lua</td>
</tr>
<tr>
<td>Narrow.lua</td>
<td>Narrow.lua</td>
<td>Narrow.lua</td>
<td>Narrow.lua</td>
</tr>
<tr>
<td>Pooling.lua</td>
<td>Pooling.lua</td>
<td>Pooling.lua</td>
<td>Pooling.lua</td>
</tr>
<tr>
<td>SpatialAdaptiveMaxPooling.lua</td>
<td>SpatialAveragePooling.lua</td>
<td>SpatialBatchNormalization.lua</td>
<td>SpatialConvolution.lua</td>
</tr>
<tr>
<td>SpatialAveragePooling.lua</td>
<td>SpatialBatchNormalization.lua</td>
<td>SpatialContrastiveNormalization.lua</td>
<td>SpatialConvolution.lua</td>
</tr>
<tr>
<td>SpatialConvolution.lua</td>
<td>SpatialConvolutionLocal.lua</td>
<td>SpatialConvolutionMM.lua</td>
<td>SpatialConvolutionMap.lua</td>
</tr>
<tr>
<td>SpatialConvolutionMap.lua</td>
<td>SpatialCrossMapLRN.lua</td>
<td>SpatialDivisiveNormalization.lua</td>
<td>SpatialDropout.lua</td>
</tr>
<tr>
<td>SpatialDropout.lua</td>
<td>SpatialFractionalMaxPooling.lua</td>
<td>SpatialFullConvolution.lua</td>
<td>SpatialFullConvolutionMap.lua</td>
</tr>
<tr>
<td>SpatialFullConvolution.lua</td>
<td>SpatialFullConvolutionMap.lua</td>
<td>SpatialFullConvolutionMap.lua</td>
<td>SpatialMaxPooling.lua</td>
</tr>
<tr>
<td>SpatialMaxPooling.lua</td>
<td>SpatialMaxUnpooling.lua</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpatialMaxUnpooling.lua</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

https://github.com/torch/nn
Torch: Modules

Tons of built-in modules and loss functions

New ones all the time:

https://github.com/torch/nn
Torch: Modules

Writing your own modules is easy!

```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
```

```lua
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)
```
Container modules allow you to combine multiple modules.

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

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

*Container* modules allow you to combine multiple modules.

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

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

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

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

Inputs: x, y, z
Outputs: c
a = x + y
b = a ⊙ 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
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: Pretrained Models

**loadcaffe**: Load pretrained Caffe models: AlexNet, VGG, some others
[https://github.com/szagoruyko/loadcaffe](https://github.com/szagoruyko/loadcaffe)

**GoogLeNet v1**: [https://github.com/soumith/inception.torch](https://github.com/soumith/inception.torch)

**GoogLeNet v3**: [https://github.com/Moodstocks/inception-v3.torch](https://github.com/Moodstocks/inception-v3.torch)

**ResNet**: [https://github.com/facebook/fb.resnet.torch](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

**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](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](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](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](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

x + a ⊙ b + c → z

\[ x + y + z = a \]

\[ x + y + z = b \]

\[ x + y + z = c \]
Theano: Computational Graphs

Theano code:

```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)
xy = np.random.randn(4, 5)
zz = np.random.randn(4, 5)
print f(xx, xy, zz)

# Repeat the same computation
# explicitly using numpy ops
aa = xx + yy
bb = aa + zz
cc = aa + bb
print cc
```
Theano: Computational Graphs

Define symbolic variables; these are inputs to the graph

```
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

Compute intermediates and outputs symbolically

\[
\begin{align*}
\text{x} & \quad \text{y} \quad \text{z} \\
+ & \quad a \\
a & \quad b \\
\odot & \quad c \\
+ & \quad + \\
& 
\end{align*}
\]

```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

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

```
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

\[ \begin{align*}
  & x \quad y \quad z \\
  & + \\
  & a \\
  & \odot \\
  & b \\
  & + \\
  & c
\end{align*} \]

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

```
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

\[ x + y + z \]

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:

\( x = \text{data} \)
\( y = \text{labels} \)
\( w_1 = \text{first-layer weights} \)
\( w_2 = \text{second-layer weights} \)
Theano: Simple Neural Net

**Forward**: Compute scores (symbolically)

```python
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)
w1 = 1e-3 * np.random.randn(D, H)
w2 = 1e-3 * np.random.randn(H, C)
loss, scores = f(xx, yy, w1, w2)
print loss
```
Theano: Simple Neural Net

**Forward**: Compute probs, loss (symbolically)
Theano: Simple Neural Net

Compile a function that computes loss, scores

```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)
w1 = 1e-3 * np.random.randn(D, H)
w2 = 1e-3 * np.random.randn(H, C)

loss, scores = f(xx, yy, w1, w2)
print loss
```
Theano: Simple Neural Net

Stuff actual numpy arrays into the function

```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)
w1 = 1e-3 * np.random.randn(D, H)
w2 = 1e-3 * np.random.randn(H, C)

loss, scores = f(xx, yy, w1, w2)
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

```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 computes gradients for us symbolically!
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],
)
```

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],
)

# 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 range(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],
)

# 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 range(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...
Same as before: Define dimensions, define symbolic variables for x, y

```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)
    )
)
```
Define weights as **shared variables** that persist in the graph between calls; initialize with numpy arrays.
Theano: Shared Variables

```python
N, D, H, C = 64, 1000, 100, 10
x = T.matrix('x')
y = T.vector('y', dtype='int64')
wl = theano.shared(1e-3 * np.random.randn(D, H), name='wl'
w2 = theano.shared(1e-3 * np.random.randn(H, C), name='w2')

a = x.dot(wl)
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, [wl, w2])

learning_rate = 1e-1
train = theano.function(
    inputs=[x, y],
    outputs=loss,
    updates=(
        (wl, wl - learning_rate * dw1),
        (w2, w2 - learning_rate * dw2)
    )
)
```

Same as before: Compute scores, loss, gradients symbolically
Theano: Shared Variables

Compiled function inputs are $x$ and $y$; weights live in the graph.
Theano: Shared Variables

```plaintext
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

To train the net, just call function repeatedly!

```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)
)
```

\[ xx = \text{np.random.randn}(N, D) \]
\[ yy = \text{np.random.randint}(C, \text{size}=N) \]

\[ \text{for } t \text{ in } \text{xrange}(100): \]
\[ \text{loss} = \text{train}(xx, yy) \]
\[ \text{print } \text{loss} \]
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
Lasagne: High Level Wrapper

Set up symbolic Theano variables for data, labels

```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=0.9)

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')
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=0.8)
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=0.8)
train_fn = theano.function([x, y], loss, updates=updates)
xx = np.random.randn(N, D)
xy = 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

```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()
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)
xy = np.random.randint(C, size=N).astype(np.int64)
for t in xrange(100):
    loss_val = train_fn(xx, xy)
    print loss_val
```
Lasagne: High Level Wrapper

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

```python
import numpy as np
import theano
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)
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:

```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

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

We will optimize the model using SGD with Nesterov momentum.

```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

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

Generate some random data and train the model.

```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

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 … )
Theano: Pretrained Models

**Lasagne Model Zoo** has pretrained common architectures: https://github.com/Lasagne/Recipes/tree/master/modelzoo

**AlexNet with weights:** [link](https://github.com/uoguelph-mlrg/theano_alexnet)

**sklearn-theano:** Run OverFeat and GoogLeNet forward, but no fine-tuning? [link](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 [link](https://github.com/kitofans/caffe-theano-conversion)
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
1 import tensorflow as tf
2 import numpy as np
3
4 N, D, H, C = 64, 1000, 100, 10
5 x = tf.placeholder(tf.float32, shape=[None, D])
6 y = tf.placeholder(tf.float32, shape=[None, C])
7
8 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
9 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
10
11 a = tf.matmul(x, w1)
12 a_relu = tf.nn.relu(a)
13 scores = tf.matmul(a_relu, w2)
14 probs = tf.nn.softmax(scores)
15 loss = -tf.reduce_sum(y * tf.log(probs))
16
17 learning_rate = 1e-2
18 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
19
20 xx = np.random.randn(N, D).astype(np.float32)
21 yy = np.zeros((N, C)).astype(np.float32)
22 yy[np.arange(N), np.random.randint(C, size=N)] = 1
23
24 with tf.Session() as sess:
25   sess.run(tf.initialize_all_variables())
26
27 for t in range(100):
28   _, loss_value = sess.run([train_step, loss],
29                     feed_dict={x: xx, y: yy})
30
31 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
TensorFlow: Two-Layer Net

**Forward**: Compute scores, probs, loss (symbolically)

```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 range(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

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

```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(np.random.randn(D, H).astype(np.float32))
w2 = tf.Variable(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)
for t in xrange(100):
    loss_value = sess.run([train_step, loss],
                          feed_dict={x: xx, y: yy})
    print loss_value
```

Actually train the model
TensorFlow: Tensorboard

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

x = tf.placeholder(tf.float32, shape=[None, 1])
y = tf.placeholder(tf.float32, shape=[None, 1])
w1 = tf.Variable(1e-3 * np.random.randn(0, 2).astype(np.float32))
w2 = tf.Variable(1e-3 * np.random.randn(2, 1).astype(np.float32))
a = tf.matmul(x, w1)
a_reu = tf.nn.relu(a)
scores = tf.matmul(a_reu, w2)
probs = tf.nn.softmax(scores)
loss = -tf.reduce_sum * tf.log(probs)
loss_summary = tf.scalar_summary('loss', loss)
w1_hist = tf.histogram_summary('w1', w1)
w2_hist = tf.histogram_summary('w2', w2)

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

    for t in range(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.

```python
for t in range(100):
    summary_str, _, loss_value = sess.run([merged, train_step, loss],
                                           feed_dict={x: xx, y: yy})
    writer.add_summary(summary_str, t)
```
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)
relu = tf.nn.relu(a)
scores = tf.matmul(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
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

<table>
<thead>
<tr>
<th></th>
<th>Caffe</th>
<th>Torch</th>
<th>Theano</th>
<th>TensorFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
<td>C++, Python</td>
<td>Lua</td>
<td>Python</td>
<td>Python</td>
</tr>
<tr>
<td><strong>Pretrained</strong></td>
<td>Yes ++</td>
<td>Yes ++</td>
<td>Yes (Lasagne)</td>
<td>Inception</td>
</tr>
<tr>
<td><strong>Multi-GPU: Data parallel</strong></td>
<td>Yes</td>
<td>Yes (cunn.DataParallelTable)</td>
<td>Yes (platoon)</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Multi-GPU: Model parallel</strong></td>
<td>No</td>
<td>Yes (fbcunn.ModelParallel)</td>
<td>Experimental</td>
<td>Yes (best)</td>
</tr>
<tr>
<td><strong>Readable source code</strong></td>
<td>Yes (C++)</td>
<td>Yes (Lua)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Good at RNN</strong></td>
<td>No</td>
<td>Mediocre</td>
<td>Yes</td>
<td>Yes (best)</td>
</tr>
</tbody>
</table>
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

```cpp
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](https://github.com/BVLC/caffe/blob/master/include/caffe/blob.hpp)
Caffe: Blobs

N-dimensional array for storing activations and weights

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

N-dimensional array for storing activations and weights

Template over datatype

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

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:
  \textbf{data}: values
  \textbf{diffs}: gradients

Stores CPU / GPU versions of each tensor

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

A small unit of computation

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

```cpp
// TEMPLATE

// For reference:
// https://github.com/BVLC/caffe/blob/master/include/caffe/layer.hpp
```

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

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

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

Tons of different layer types:

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

[GitHub Repository](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

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

```cpp
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.
Caffe: Solver

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

Handles snapshotting, restoring from snapshots

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

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