RNN LSTM and Deep Learning Libraries

UDRC Summer School

Muhammad Awais
m.a.rana@surrey.ac.uk
Outline

- Recurrent Neural Network
- Application of RNN
- LSTM
- Caffe
- Torch
- Theano
- TensorFlow
Flexibility of Recurrent Neural Networks

Vanilla Neural Networks
Flexibility of Recurrent Neural Networks

e.g. Image Captioning
image -> sequence of words
Flexibility of Recurrent Neural Networks

e.g. Sentiment Classification
sequence of words -> sentiment
Flexibility of Recurrent Neural Networks

- **one to one**
- **one to many**
- **many to one**
- **many to many**

e.g. **Machine Translation**
seq of words -> seq of words
Flexibility of Recurrent Neural Networks

e.g. Video classification on frame level
Recurrent Neural Networks

RNN

x
Recurrent Neural Networks

usually want to predict a vector at some time steps
We can process a sequence of vectors $\mathbf{x}$ by applying a recurrence formula at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

- $h_t$: new state
- $h_{t-1}$: old state
- $x_t$: input vector at some time step
- $f_W$: some function with parameters $W$
Recurrent Neural Networks

We can process a sequence of vectors $\mathbf{x}$ by applying a recurrence formula at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

new state

old state

input vector at some time step

some function with parameters $W$

Notice: the same function and the same set of parameters are used at every time step.
Recurrent Neural Networks

The state consists of a single "hidden" vector $h$:

$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$
Recurrent Neural Networks

Character-level language model example

Vocabulary: [h,e,l,o]

Example training sequence: “hello”
Recurrent Neural Networks

Character-level language model example

Vocabulary: [h, e, l, o]

Example training sequence: “hello”
Recurrent Neural Networks

Character-level language model example

Vocabulary: [h,e,l,o]

Example training sequence: “hello”
Character-level language model example

Vocabulary: [h,e,l,o]

Example training sequence: “hello”
Recurrent Neural Networks

Image Captioning

Explain Images with Multimodal Recurrent Neural Networks, Mao et al.
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei
Show and Tell: A Neural Image Caption Generator, Vinyals et al.
Long-term Recurrent Convolutional Networks for Visual Recognition and Description, Donahue et al.
Learning a Recurrent Visual Representation for Image Caption Generation, Chen and Zitnick
Recurrent Neural Networks

Recurrent Neural Network

Convolutional Neural Network
Recurrent Neural Networks
Recurrent Neural Networks
Recurrent Neural Networks
Recurrent Neural Networks

deep learning architecture

<START>

x0
<START>
Recurrent Neural Networks

before:
\[ h = \tanh(W_{xh} \cdot x + W_{hh} \cdot h) \]

now:
\[ h = \tanh(W_{xh} \cdot x + W_{hh} \cdot h + W_{ih} \cdot v) \]
Recurrent Neural Networks
Recurrent Neural Networks
Recurrent Neural Networks

```
image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
```

```
<START>
straw
hat
```

```
y0
h0
x0
<START>
```

```
y1
h1
```

```
test image
```

```
sample!
```
Recurrent Neural Networks

test image
Recurrent Neural Networks

test image

sample <END> token => finish.
Recurrent Neural Networks

Image Sentence Datasets

Microsoft COCO
[Tsung-Yi Lin et al. 2014]
mscoco.org

currently:
~120K images
~5 sentences each
Recurrent Neural Networks

"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

"boy is doing backflip on wakeboard."


Recurrent Neural Networks
Recurrent Neural Networks

\[ h^l_t = \tanh W^l \left( \begin{pmatrix} h^l_{t-1} \\ h^l_{t-1} \end{pmatrix} \right) \]

\[ h \in \mathbb{R}^n \quad W^l \in [n \times 2n] \]
Recurrent Neural Networks

\[ h_t^l = \tanh(W^l(h_{t-1}^{l-1}, h_t^{l-1})) \]

\[ W^l \in \mathbb{R}^{n \times 2n} \]

**LSTM:**

\[
\begin{pmatrix}
i \\ f \\ o \\ g
\end{pmatrix} =
\begin{pmatrix}
sigm & \text{sigm} & \text{sigm} & \text{tanh}
\end{pmatrix}
W^l(h_{t-1}^{l-1}, h_t^{l-1})
\]

\[ c_t^l = f \odot c_{t-1}^l + i \odot g \]

\[ h_t^l = o \odot \text{tanh}(c_t^l) \]
Long Short Term Memory (LSTM)

\[
\begin{pmatrix}
i \\
f \\
o \\
g
\end{pmatrix} = 
\begin{pmatrix}
sigm \\
sigm \\
sigm \\
tanh
\end{pmatrix} W' \begin{pmatrix} h_{t-1}^l \\ h_{t-1}^l \\ h_{t-1}^l \\ h_{t-1}^l \end{pmatrix}
\]

\[
c_t^l = f \odot c_{t-1}^l + i \odot g
\]

\[
h_t^l = o \odot \tanh(c_t^l)
\]
Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]
Long Short Term Memory (LSTM)

\[
\begin{align*}
(i) &= \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} W^l \begin{pmatrix} h_{t-1}^l \\ h_{t-1}^l \end{pmatrix} \\
(f, o, g) &= \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} \\
(c_t^l) &= f \odot c_{t-1}^l + i \odot g \\
h_t^l &= o \odot \text{tanh}(c_t^l)
\end{align*}
\]
Long Short Term Memory (LSTM)

\[
\begin{align*}
\text{cell state } c &= f \odot c_{t-1} + i \odot g \\
\text{output } h_t &= o \odot \tanh(c_t)
\end{align*}
\]
Long Short Term Memory (LSTM)
Long Short Term Memory (LSTM)
Long Short Term Memory (LSTM)
Summary

- RNNs allow a lot of flexibility in architecture design
- Vanilla RNNs are simple but don’t work very well
- Common to use LSTM or GRU: their additive interactions improve gradient flow
- Backward flow of gradients in RNN can explode or vanish. Exploding is controlled with gradient clipping. Vanishing is controlled with additive interactions (LSTM)
- Better/simpler architectures are a hot topic of current research
- Better understanding (both theoretical and empirical) is needed.
Deep Learning Libraries
Caffe, Torch, Theano, TensorFlow
Caffe overview

From U.C. Berkeley
Written in C++
Has Python and MATLAB bindings
Good for training or finetuning feedforward models
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)
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/
Protocol Buffers

“Typed JSON” from Google

Define “message types” in .proto files

Serialize instances to text files (.prototxt)

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

“Typed JSON” from Google

Define “message types” in .proto files

Serialize instances to text files (.prototxt)

Compile classes for different languages

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

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

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

C++ class
```
Person john;
fstream input(argc[1],
  ios::in | ios::binary);
john.ParseFromIStream(&input);
id = john.id();
name = john.name();
email = john.email();
```
Caffe

Protocol Buffers

```protobuf
message NetParameter {
  optional string name = 1; // consider giving the network a name
  repeated string input = 3;
  // The input blobs to the network.
  repeated BlobShape input_shape = 8;
  // The shape of the input blobs.
  repeated BlobShape input_dim = 4;
  // 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.
  repeated 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 {
  // Specify 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.
  repeated string net = 24;
  // Inline train net param, possibly combined with one or more
  // test nets.
  optional NetParameter net_param = 25;
  repeated string train_net = 1; // Proto filename for the train net
}
```


← 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 2: Define Net

```yaml
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
}
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: SOFTMAXLOSS
}
```
Caffe

Step 2: Define Net

```c
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 and Blobs often have same name!
- Learning rates (weight + bias)
- Regularization (weight + bias)
Caffe

Step 2: Define Net

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

- **Layers and Blobs**
  - often have same name!
- **Learning rates**
  - (weight + bias)
- **Regularization**
  - (weight + bias)
- **Number of output classes**
**Step 2: Define Net**

```python
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
  inner_product_param {
    num_output: 2
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
  include {
    phase: TRAIN
  }
}
layers {
  bottom: "fc1"
  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)**
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

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]

Same name: weights copied

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: 1000
  }
}
```

Pretrained weights:
- "fc7.weight": [values]
- "fc7.bias": [values]
- "fc8.weight": [values]
- "fc8.bias": [values]

Same name: weights copied
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
  }
}

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
  }
}

Pretrained weights:
fc7.weight": [values]
fc7.bias": [values]
fc8.weight": [values]
fc8.bias": [values]
**Step 2: Define Net (finetuning)**

**Original prototxt:**
```proto
layer {
  name: "fc7"
  type: "InnerProduct"
  inner_product_param {
    num_output: 4096
  }
}

[... ReLU, Dropout]

layer {
  name: "fc8"
  type: "InnerProduct"
  inner_product_param {
    num_output: 1000
  }
}
```

**Modified prototxt:**
```proto
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
  }
}
```

- **Same name:** weights copied
- **Pretrained weights:**
  - "fc7.weight": [values]
  - "fc7.bias": [values]
- **Different name:** weights reinitialized
  - "fc8.weight": [values]
  - "fc8.bias": [values]
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

```plaintext
net: "models/bvlc_alexnet/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
steps: 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
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
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

Pros / Cons

(+) Good for feedforward networks
(+) Good for finetuning existing networks
(+) Train models without writing any code!
(+) Python and matlab interfaces are 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

From NYU + IDIAP
Written in C and Lua
Used a lot a Facebook, DeepMind
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

Torch

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 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
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

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'

-- 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)
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

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)
```

```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)
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

Tensor

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

Multi-dimensional matrix

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

Example:

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

Math Functions

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

- **Constructors like** `zeros`, `ones`, `eye`, etc.
- **Extractors like** `diag`, `triu`, etc.
- **Element-wise mathematical operations like** `add`, `sub`, `mul`, `div`;
- **BLAS operations**;
- **Column or row-wise operations like** `sum` and `max`;
- **Matrix-wide operations like** `trace` and `size`;
- **Convolution and cross-correlation operations like** `conv1d`;
- **Basic linear algebra operations like** `eq`, `ne`, `gt`, `lt`, `ge`, `le`, `neg`, `abs`, `sqrt`, `log`, `exp`;
- **Logical operations on** `Tensor`s.

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

https://github.com/torch/torch7/blob/master/doc/maths.md
nn module lets you easily build and train neural nets

```python
1 require 'torch'
2 require 'nn'

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

nn

nn module lets you easily build and train neural nets

Build a two-layer ReLU net

```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: local net = nn.Sequential()
7: net:add(nn.Linear(D, H))
8: net:add(nn.ReLU())
9: net:add(nn.Linear(H, C))
10: -- Collect all weights and gradients in a single Tensor
11: local weights, grad_weights = net:getParameters()
12: -- Loss functions are called "criterions"
13: local crit = nn.CrossEntropyCriterion() -- Softmax loss
14: -- Generate some random input data
15: local x = torch.rand(N, D)
16: local y = torch.Tensor(N):random()
17: -- Forward pass: Compute scores and loss
18: local scores = net:forward(x)
19: local loss = crit:forward(scores, y)
20: -- Backward pass: compute gradients
21: grad_weights:zero()
22: local dscores = crit:backward(scores, y)
23: local dx = net:backward(x, dscores)
24: -- Make a gradient step
25: local learning_rate = 1e-3
26: weights:add(-learning_rate, grad_weights)
```
nn module lets you easily build and train neural nets

Get weights and gradient for entire network

```python
require 'torch'
require 'nn'

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

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

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

-- 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)
```
nn module lets you easily build and train neural nets

Use a softmax loss function

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

Generate random data

```python
require 'torch'
require 'nn'

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

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()

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

```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
36
```
nn module lets you easily build and train neural nets

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

```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 "criteria"
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)
```
nn

nn module lets you easily build and train neural nets

Update: Make a gradient descent step
Running on GPU is easy:

```python
1 require 'torch'
2 require 'cutorch'
3 require 'nn'
4 require 'cunn'

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

Import a few new packages
Running on GPU is easy:

Import a few new packages

Cast network and criterion
Running on GPU is easy:

Import a few new packages

Cast network and criterion

Cast data and labels

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

**optim**

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

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

Import the optim package and use it in your code.
Torch

optim

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

Import optim package

Write a callback function that returns loss and gradients
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
def f(w):
    assert(w == weights)
    # 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(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

-- 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
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

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

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
    end

    return self.output
end

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

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
    end
    return self.gradInput
end
```

[https://github.com/torch/nn/blob/master/Linear.lua](https://github.com/torch/nn/blob/master/Linear.lua)
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
local Linear, parent = torch.class('nn.Linear', 'nn.Module')

function Linear:accGradParameters(input, gradOutput, scale)
    scale = scale or 1
    if input:dim() == 1 then
        self.gradWeight:add(input, gradOutput, scale)
        if self.bias then self.gradBias:add(scale * self.bias) end
    elseif input:dim() == 2 then
        self.gradWeight:addmm(input, scale, gradOutput:transpose(0, 1))
        if self.bias then self.gradBias:addmv(scale, gradOutput:transpose(0, 1), self.addBuffer) end
    end
end
```

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

Modules
Tons of built-in modules and loss functions

- 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

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

Modules
Writing your own modules is easy!

```lua
-- times_two.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
```
Torch

Modules

*Container* modules allow you to combine multiple modules
Container modules allow you to combine multiple modules.
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)
```
Modules

*Container* modules allow you to combine multiple modules

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

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

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

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

*Inputs:* $x, y, z$

*Outputs:* $c$

$a = x + y$

$b = a \odot z$

$c = a + b$
Use `nngraph` to build modules that combine their inputs in complex ways

**Inputs:** x, y, z
**Outputs:** c

\[
\begin{align*}
a &= x + y \\
b &= a \odot z \\
c &= a + b
\end{align*}
\]
Use nngraph to build modules that combine their inputs in complex ways

**Inputs:** x, y, z

**Outputs:** c

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

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

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

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
```
Define symbolic variables; these are inputs to the graph.
Theano

Computational Graphs

```
import theano
import theano.tensor as T

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

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

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

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

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

Compute intermediates and outputs symbolically
Theano

Computational Graphs

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

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

Computational Graphs

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

```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

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)
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
```
Define symbolic variables:
- \( x = \) data
- \( y = \) labels
- \( w_1 = \) first-layer weights
- \( w_2 = \) 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)
w1l = 1e-3 * np.random.randn(D, H)
w2l = 1e-3 * np.random.randn(H, C)
loss, scores = f(xx, yy, w1, w2)
print loss
```
Theano

Simple Neural Net

Forward: Compute scores (symbolically)
Theano

Simple Neural Net

Forward: Compute probs, loss (symbolically)
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)
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

```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
```

Stuff actual numpy arrays into the function
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

```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],
)
```

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

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
The function to perform gradient descent!
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

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)
arelu = tf.nn.relu(a)
scores = tf.matmul(arelu, 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
TensorFlow

TensorFlow: Two-Layer Net

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

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

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

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

TensorFlow: Multi-GPU

Data parallelism:
synchronous or asynchronous
TensorFlow

TensorFlow: Multi-GPU

Data parallelism:
synchronous or asynchronous

Model parallelism:
Split model across GPUs
TensorFlow

TensorFlow: Distributed

Single machine:
Like other frameworks

Many machines:
Not open source (yet) =(
TensorFlow

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
## Comparison between Libraries

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