RNN LSTM and Deep Learning Libraries

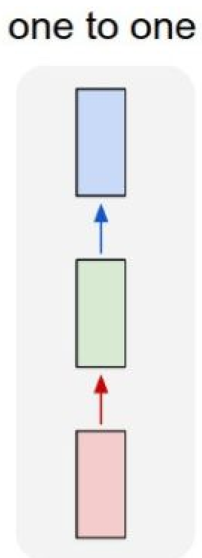
UDRC Summer School

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

Outline

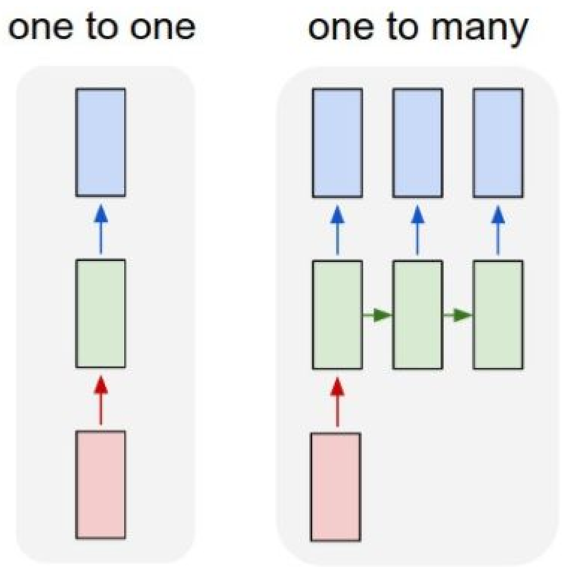
- ➢ Recurrent Neural Network
- ➢ Application of RNN
- ➢ LSTM
- ➢ Caffe
- ➢ Torch
- ➢ Theano
- ➢ TensorFlow

Flexibility of Recurrent Neural Networks



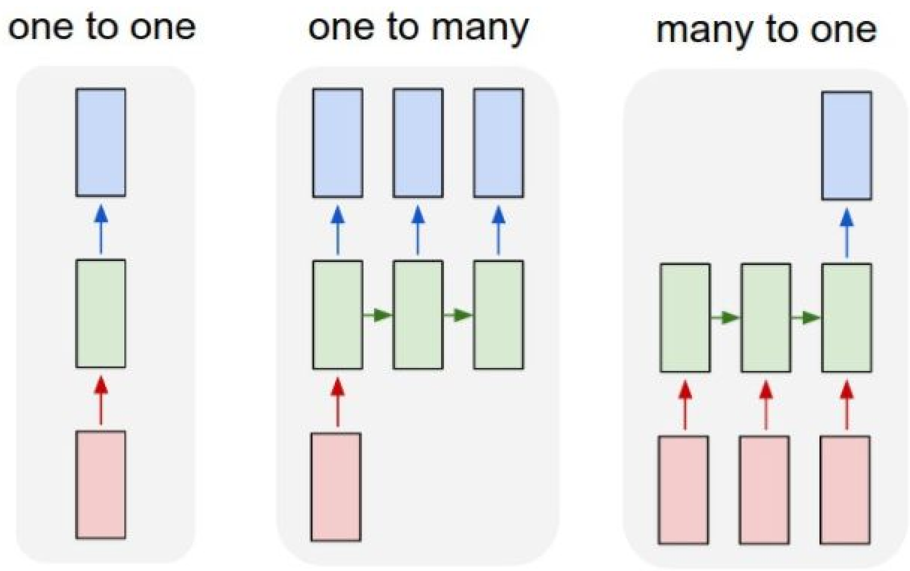
Vanilla Neural Networks

Flexibility of Recurrent Neural Networks



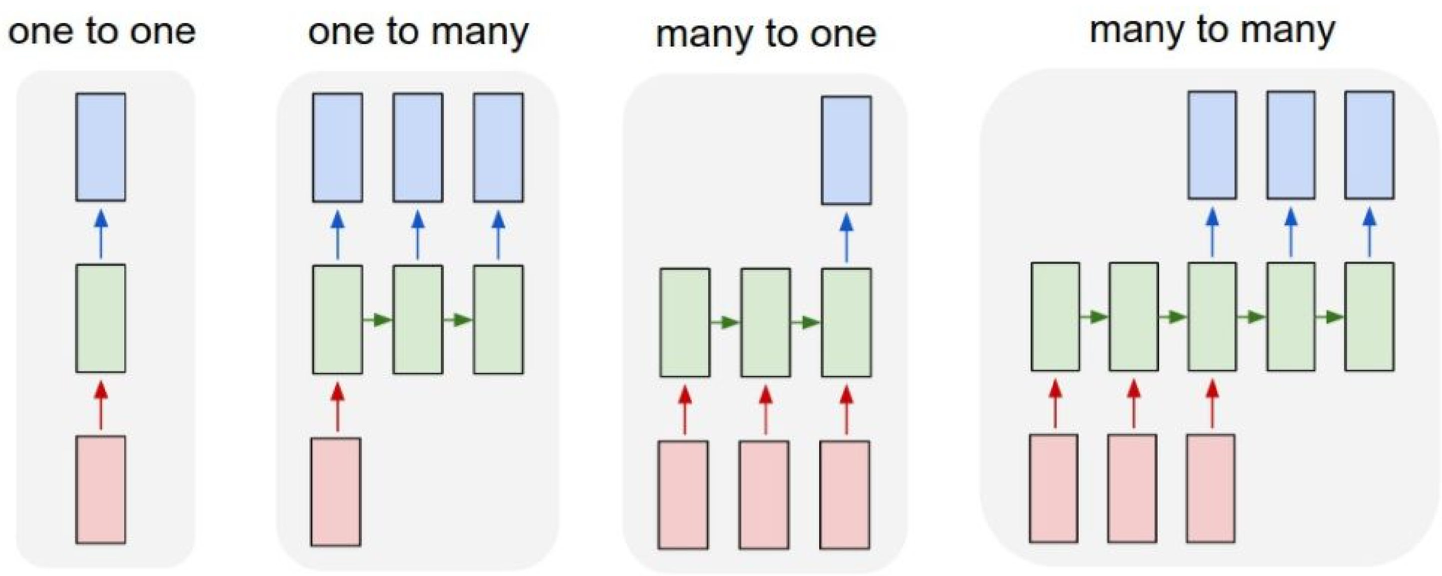
e.g. **Image Captioning**
image -> sequence of words

Flexibility of Recurrent Neural Networks



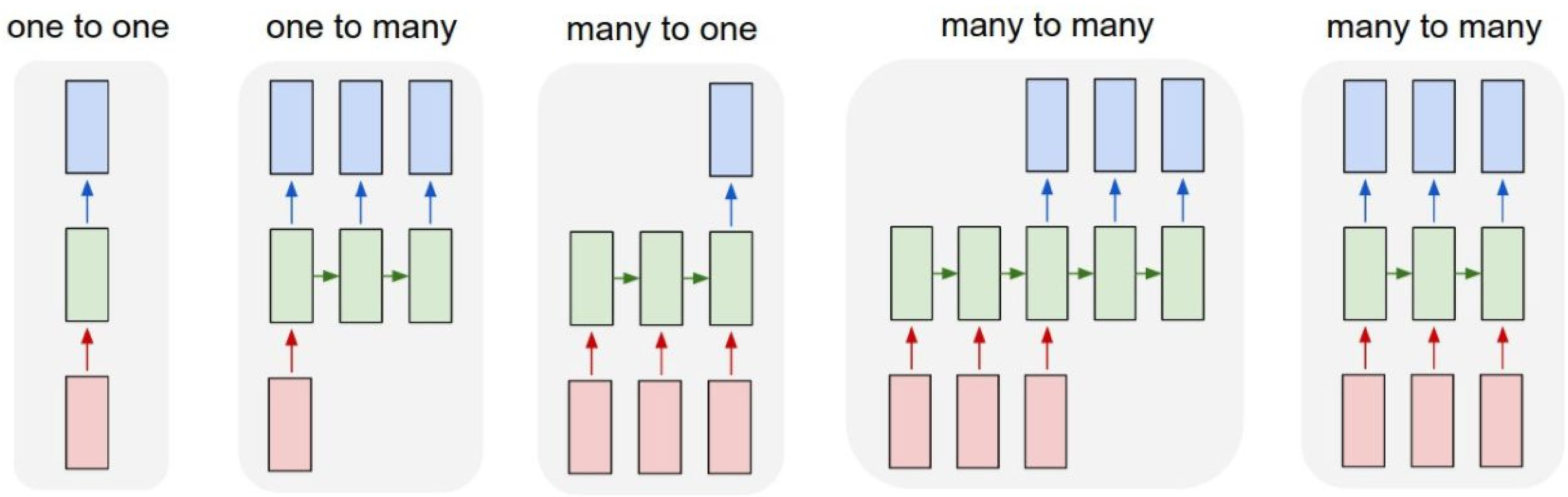
e.g. **Sentiment Classification**
sequence of words -> sentiment

Flexibility of Recurrent Neural Networks



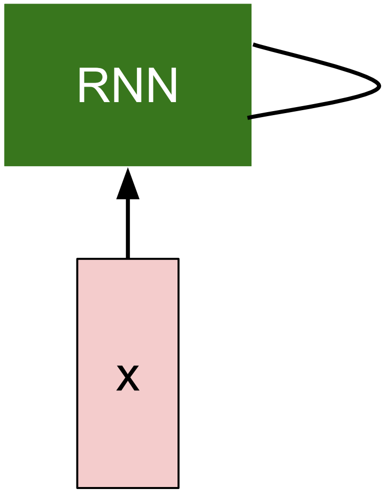
e.g. **Machine Translation**
seq of words -> seq of words

Flexibility of Recurrent Neural Networks

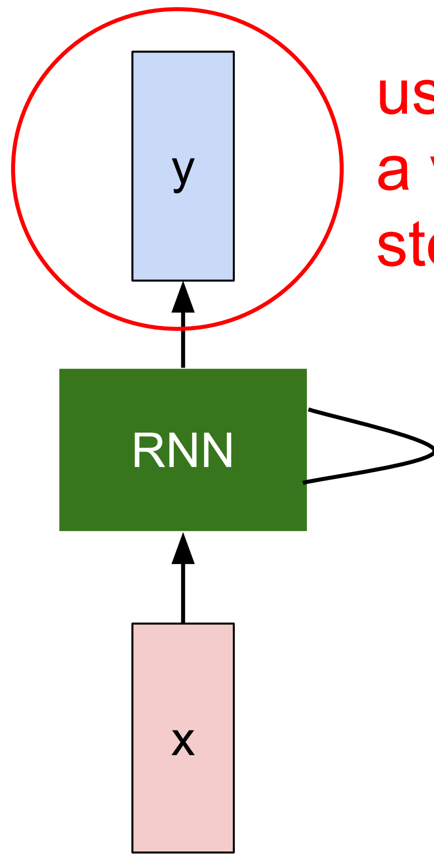


e.g. **Video classification on frame level**

Recurrent Neural Networks



Recurrent Neural Networks



usually want to predict a vector at some time steps

Recurrent Neural Networks

We can process a sequence of vectors **x** by applying a recurrence formula at every time step:

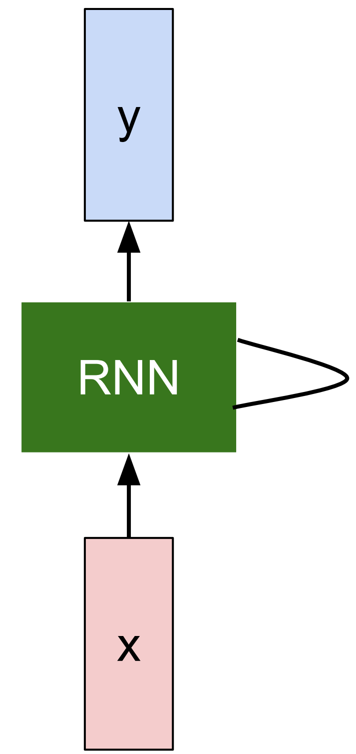
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

old state

input vector at some time step

some function with parameters W



Recurrent Neural Networks

We can process a sequence of vectors **x** by applying a recurrence formula at every time step:

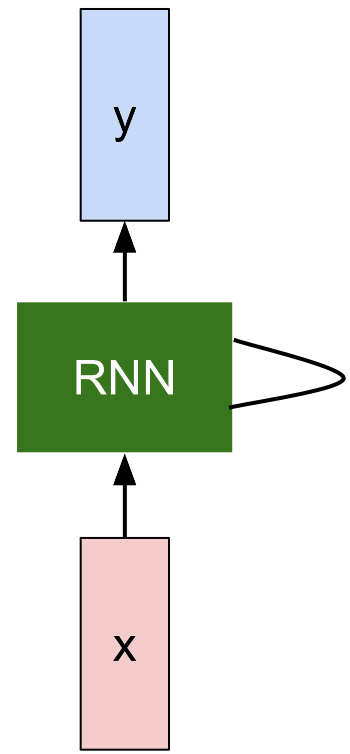
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

old state

input vector at some time step

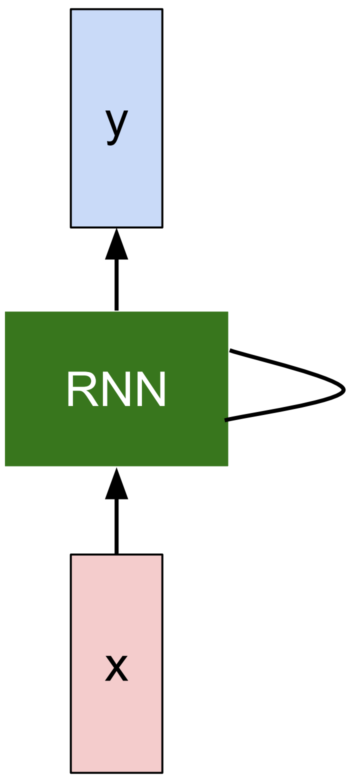
some function with parameters W



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**:



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

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

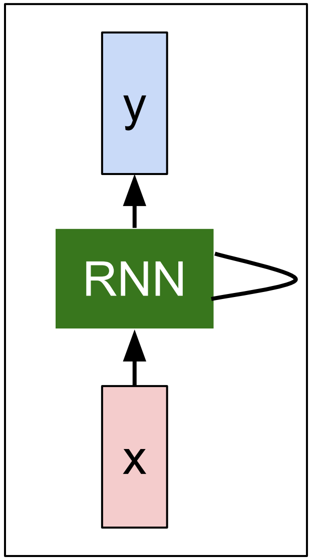
$$y_t = W_{hy}h_t$$

Recurrent Neural Networks

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”

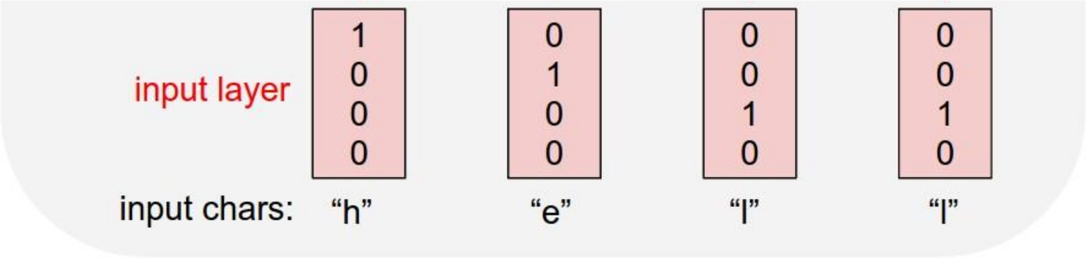


Recurrent Neural Networks

Character-level language model example

Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”



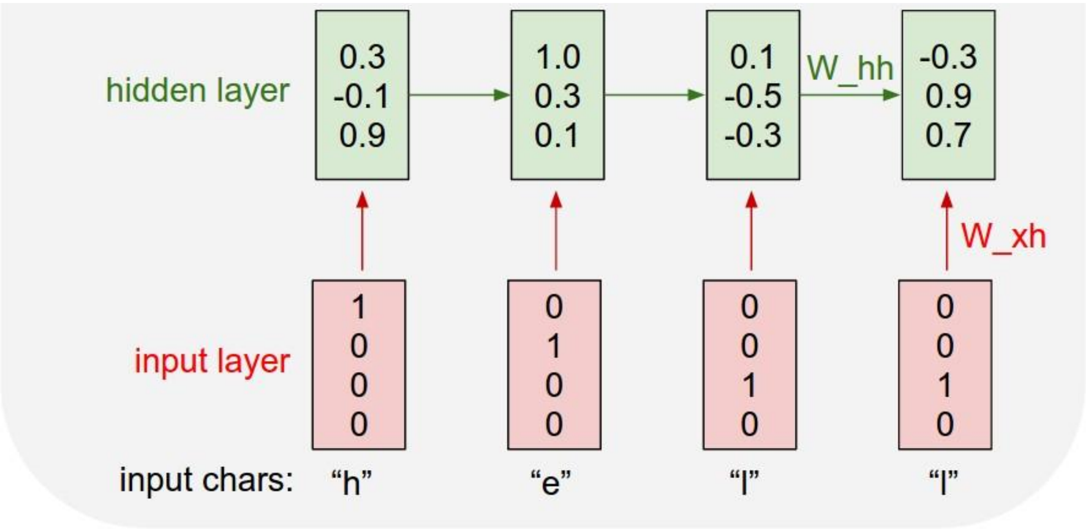
Recurrent Neural Networks

Character-level language model example

Vocabulary: [h,e,l,o]

Example training sequence: **“hello”**

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

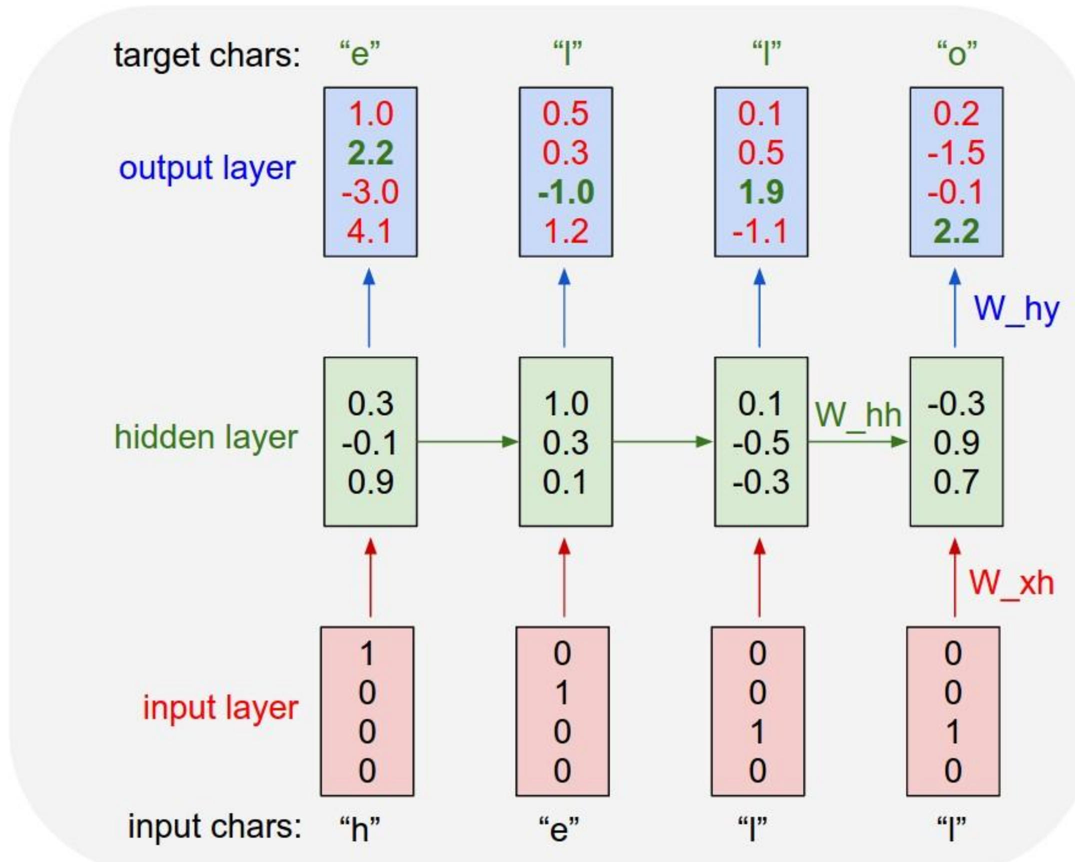


Recurrent Neural Networks

Character-level language model example

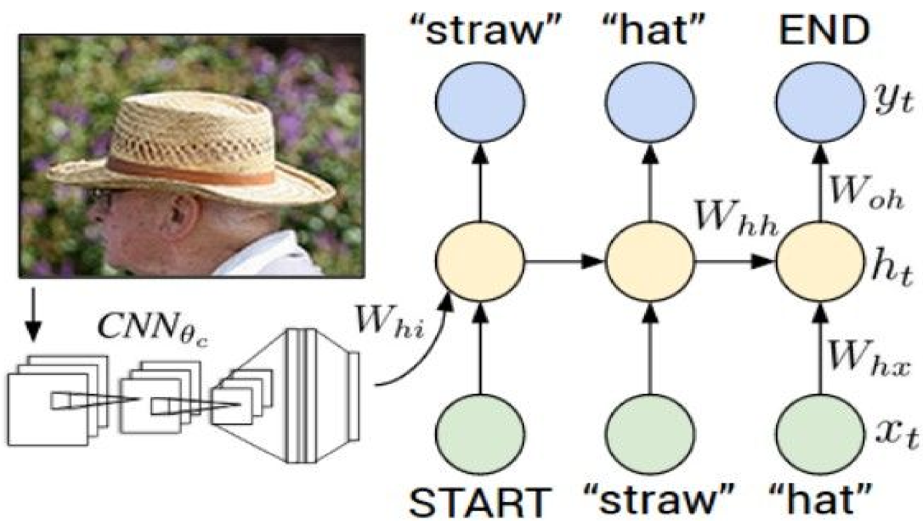
Vocabulary:
[h,e,l,o]

Example training sequence:
“hello”



Recurrent Neural Networks

Image Captioning

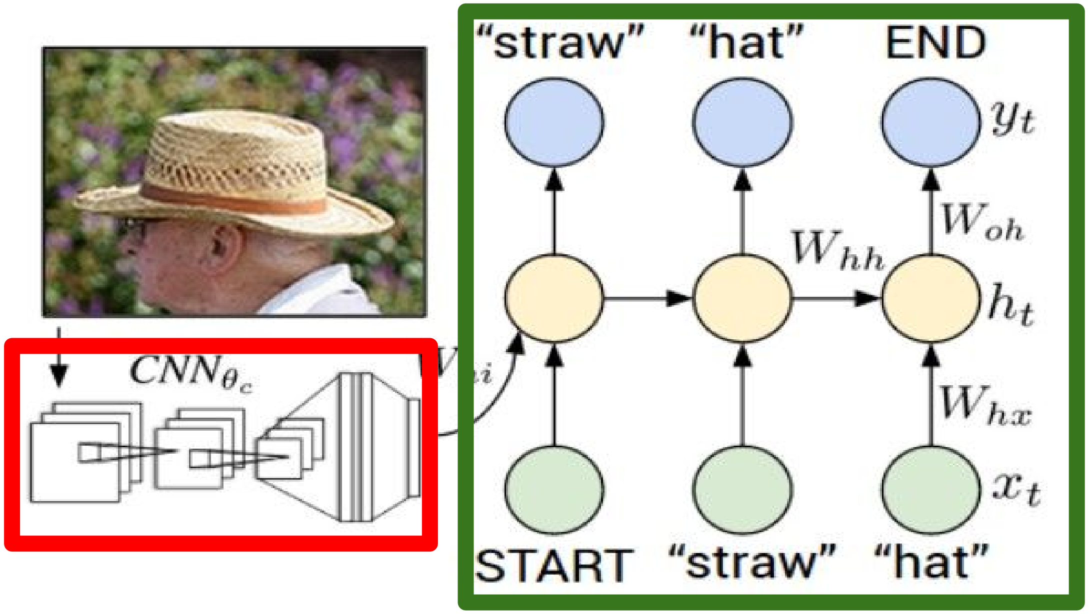


Explain Images with Multimodal Recurrent Neural Networks, Mao et al.
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei
Show and Tell: A Neural Image Caption Generator, Vinyals et al.
Long-term Recurrent Convolutional Networks for Visual Recognition and Description,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

UNIVERSITY OF SURREY

Recurrent Neural Networks

Recurrent Neural Network



Convolutional Neural Network

Recurrent Neural Networks

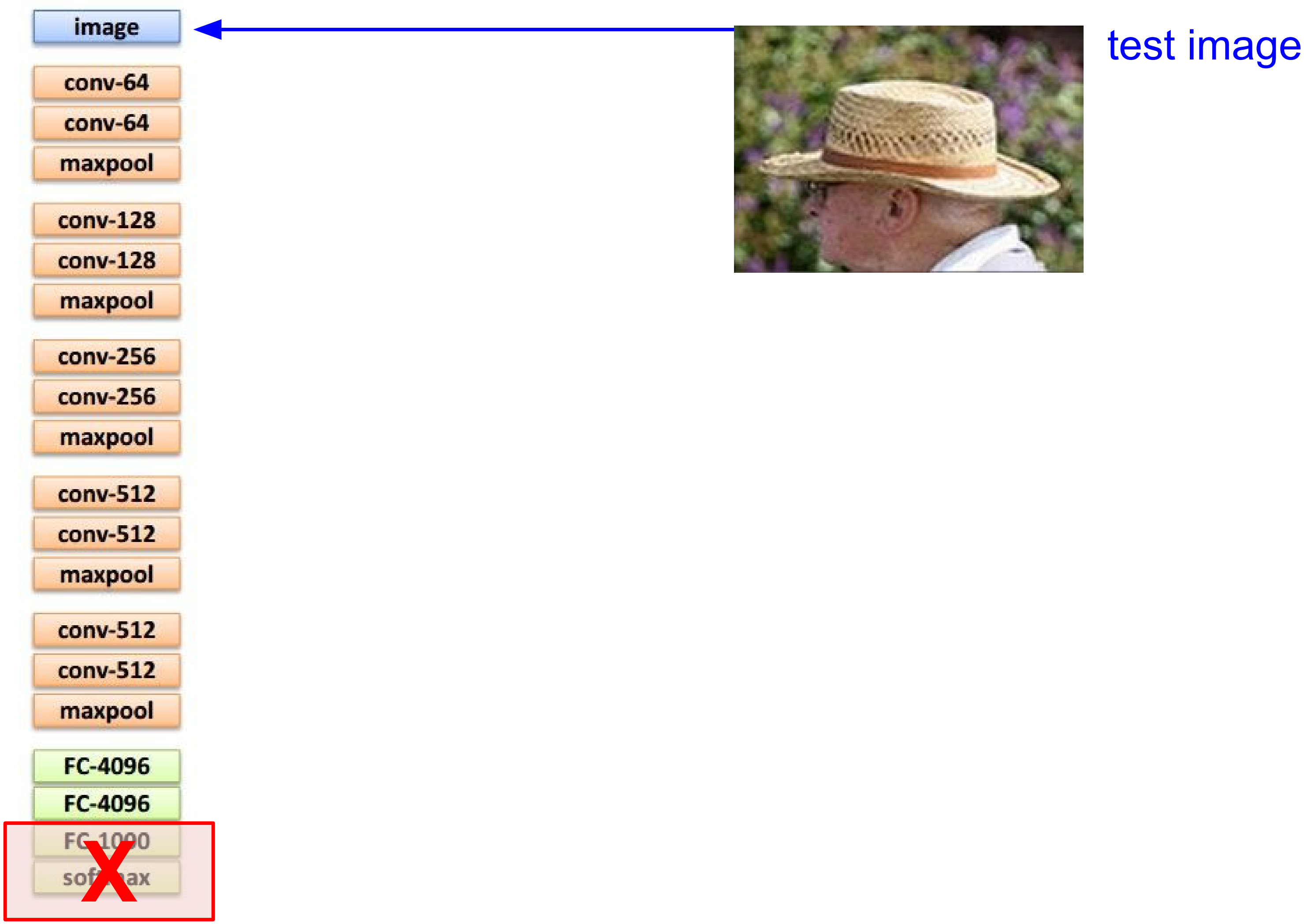
test image

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
FC-1000
softmax

test image

Recurrent Neural Networks



Recurrent Neural Networks

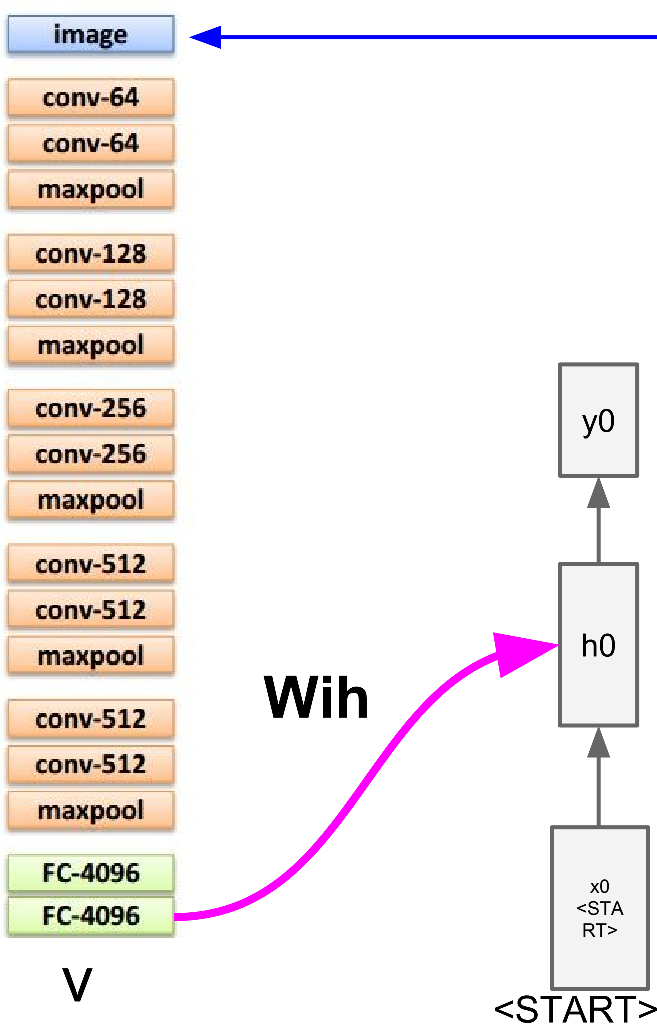
image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096

test image

x0
<STA
RT>

<START>

Recurrent Neural Networks



test image

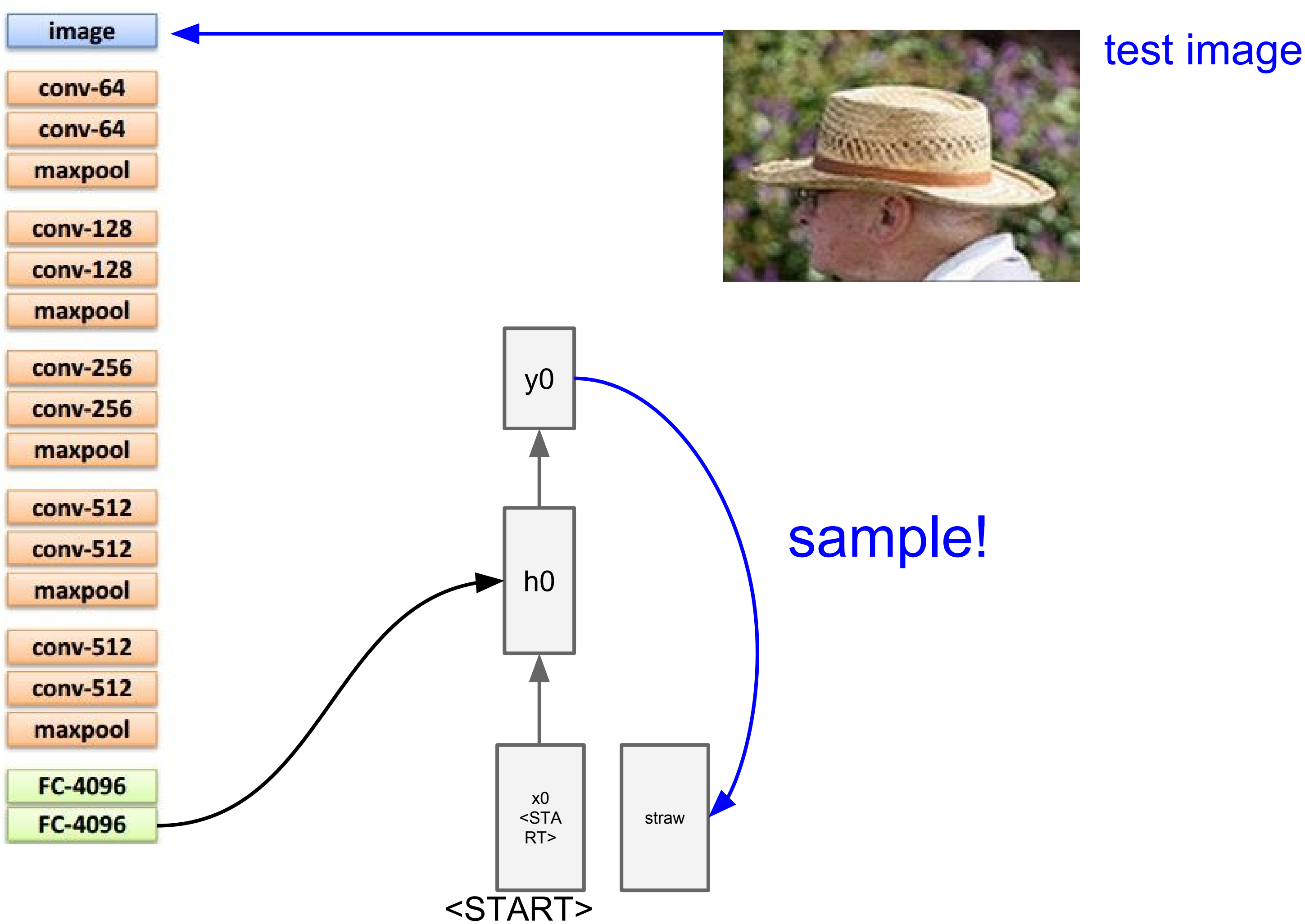
before:

$h = \tanh(W_{xh} * x + W_{hh} * h)$

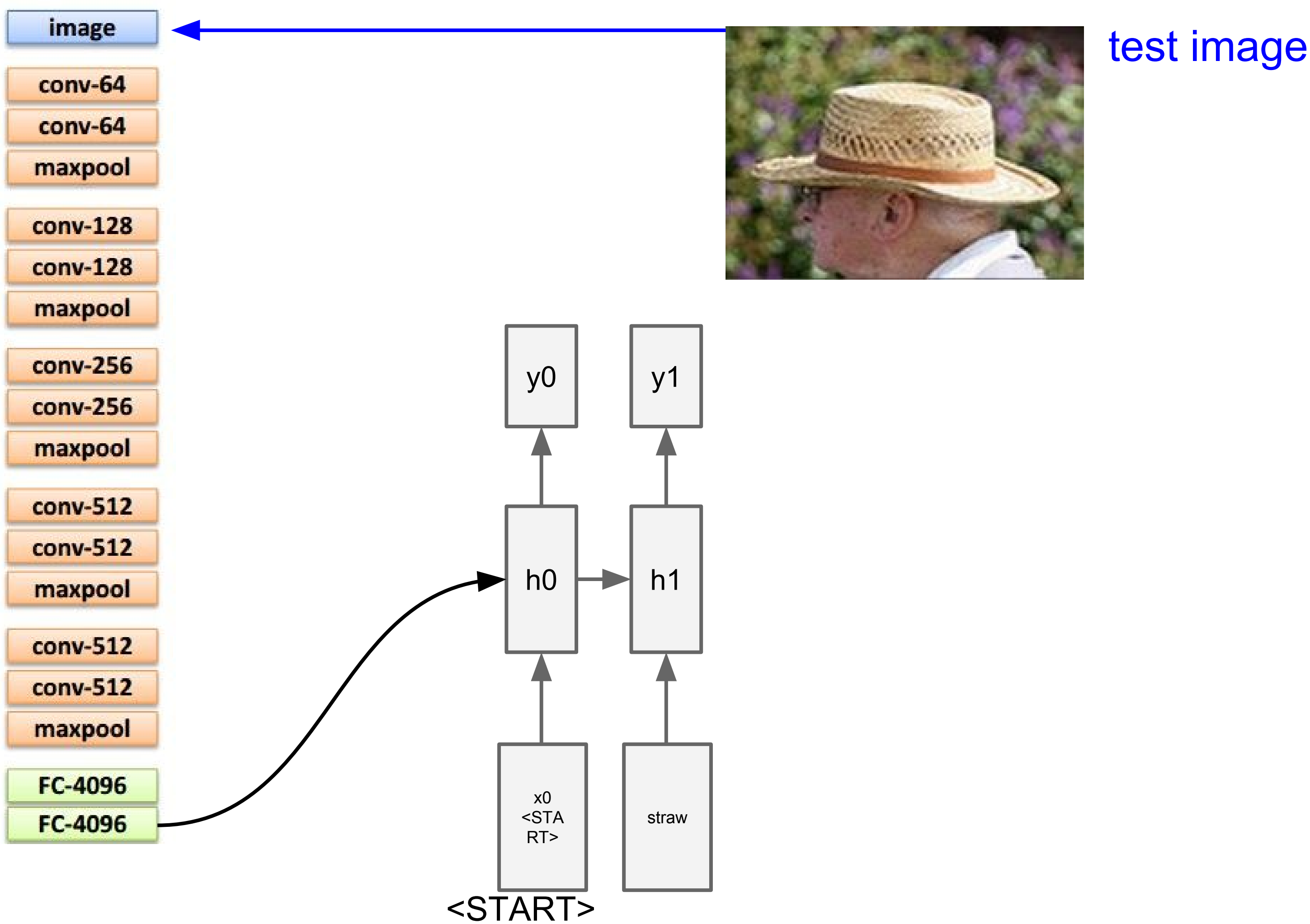
now:

$h = \tanh(W_{xh} * x + W_{hh} * h + \mathbf{W_{ih} * v})$

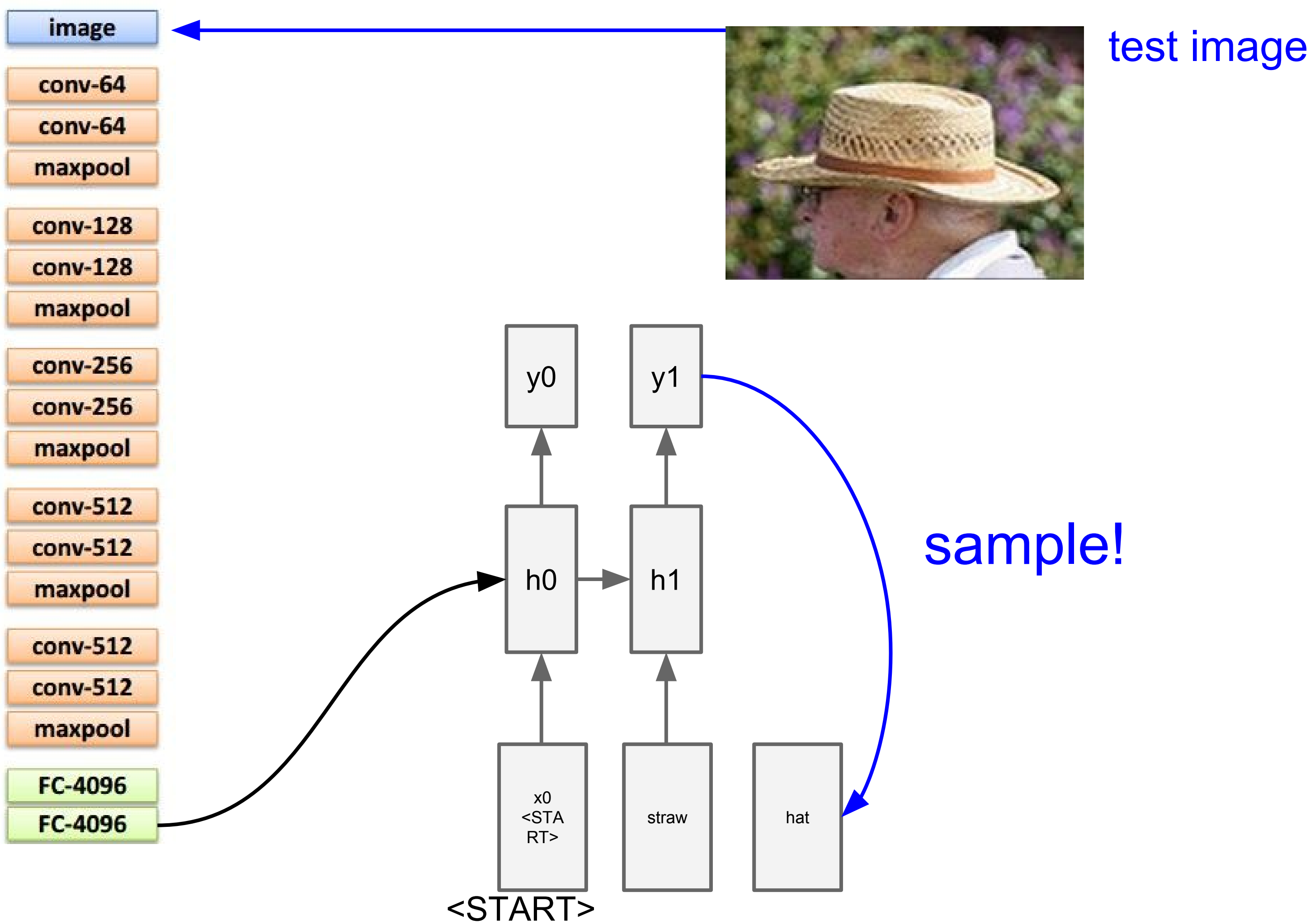
Recurrent Neural Networks



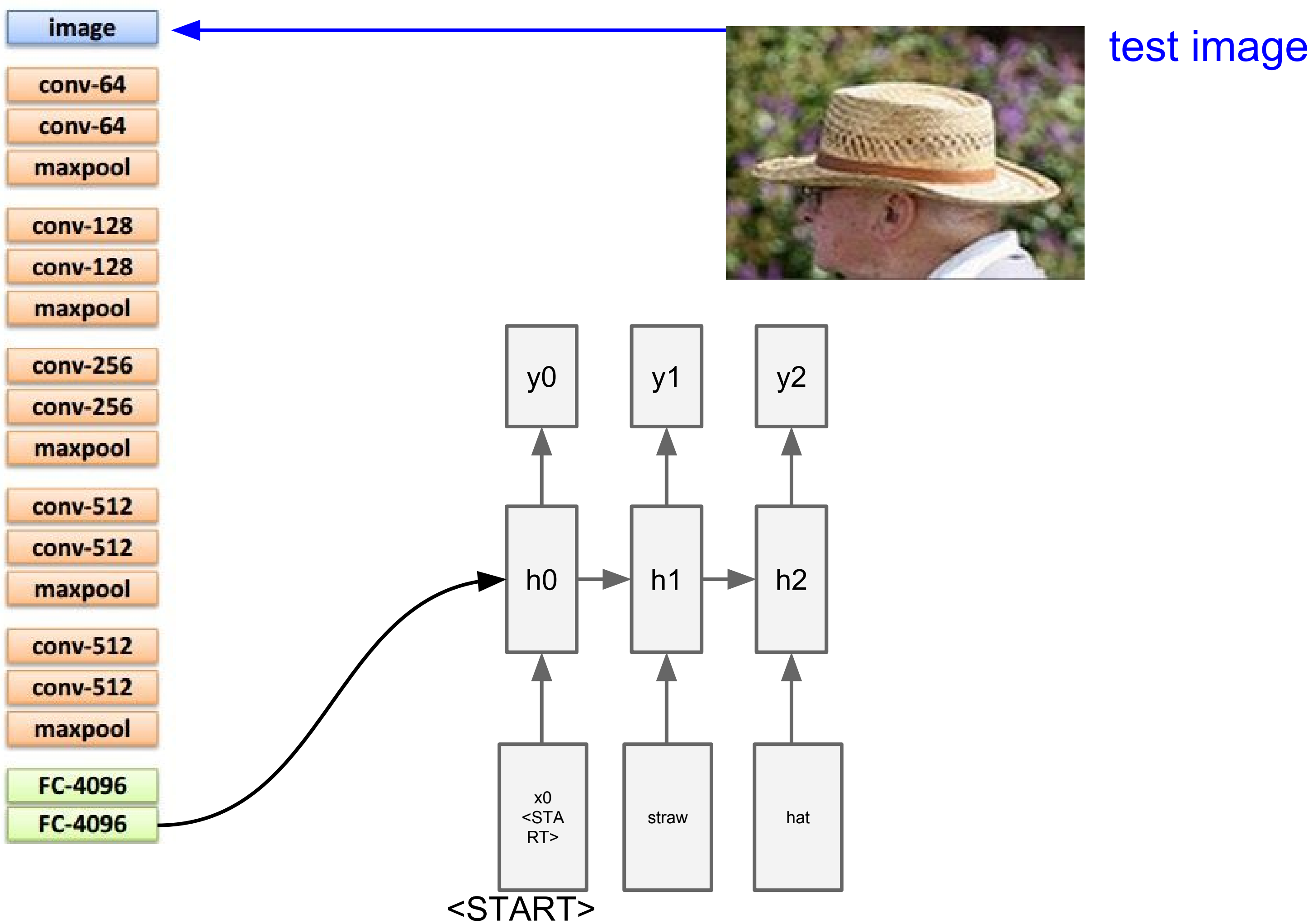
Recurrent Neural Networks



Recurrent Neural Networks

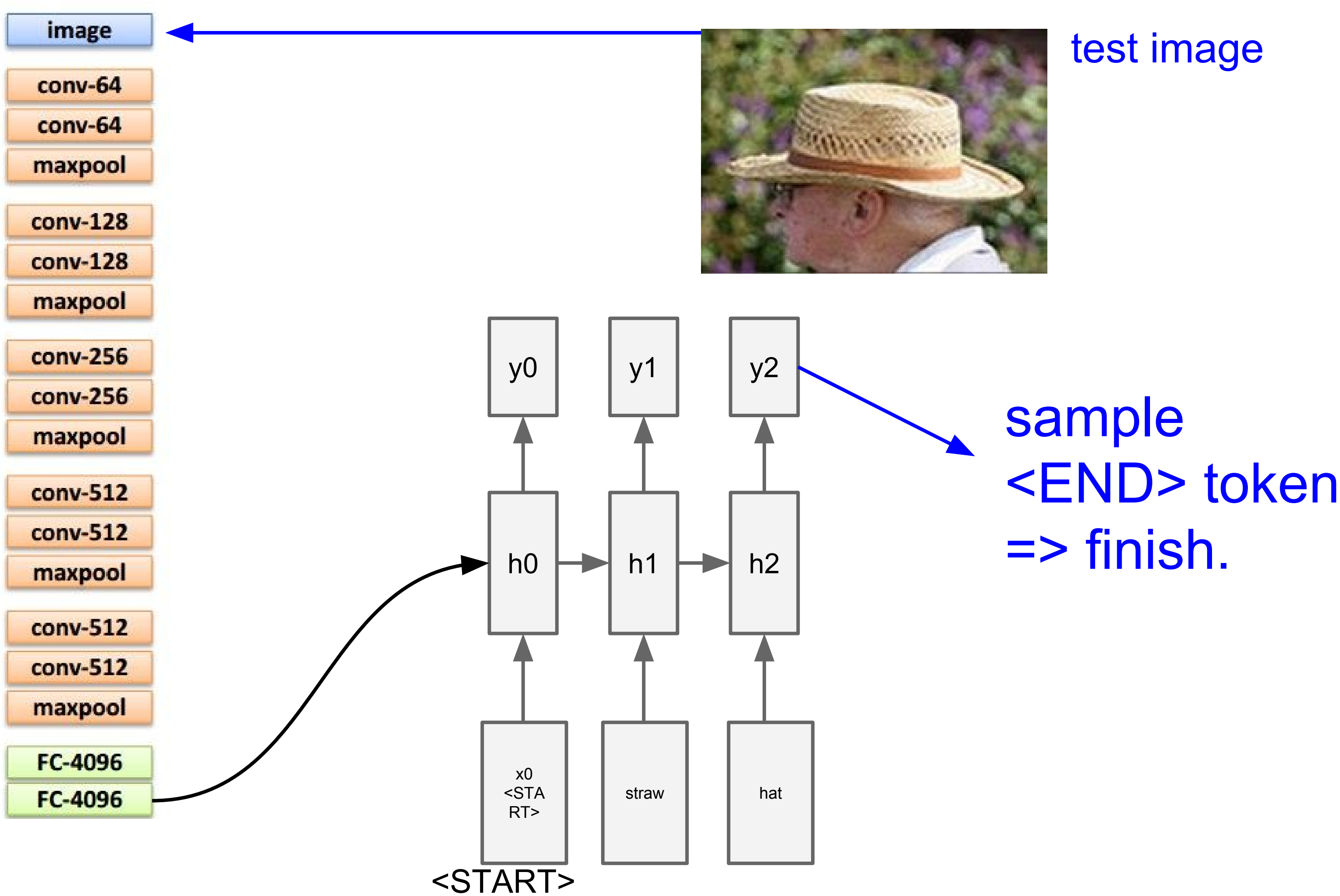


Recurrent Neural Networks



UNIVERSITY OF SURREY

Recurrent Neural Networks



Recurrent Neural Networks

Image Sentence Datasets

a man riding a bike on a dirt path through a forest.
bicyclist raises his fist as he rides on desert dirt trail.
this dirt bike rider is smiling and raising his fist in triumph.
a man riding a bicycle while pumping his fist in the air.
a mountain biker pumps his fist in celebration.

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

"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."

"a young boy is holding a baseball bat."

"a cat is sitting on a couch with a remote control."

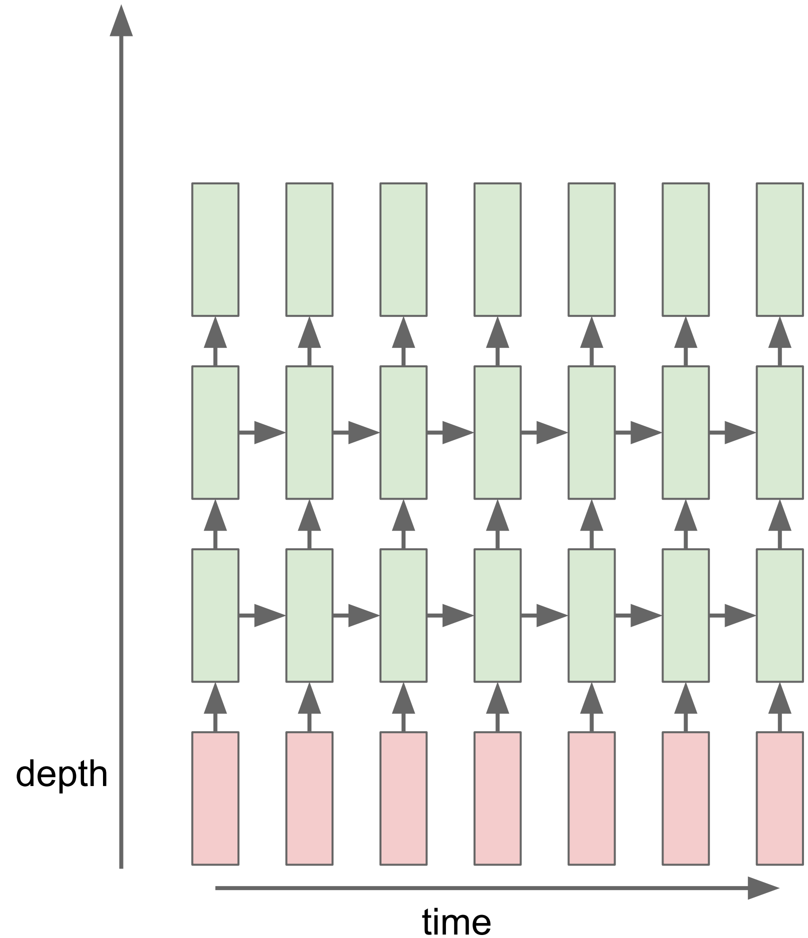
"a woman holding a teddy bear in front of a mirror."

"a horse is standing in the middle of a road."

Recurrent Neural Networks

$$\boxed{h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}}$$

$h \in \mathbb{R}^n$. $\quad W^l \; [n \times 2n]$



Recurrent Neural Networks

$$\boxed{h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}}$$

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

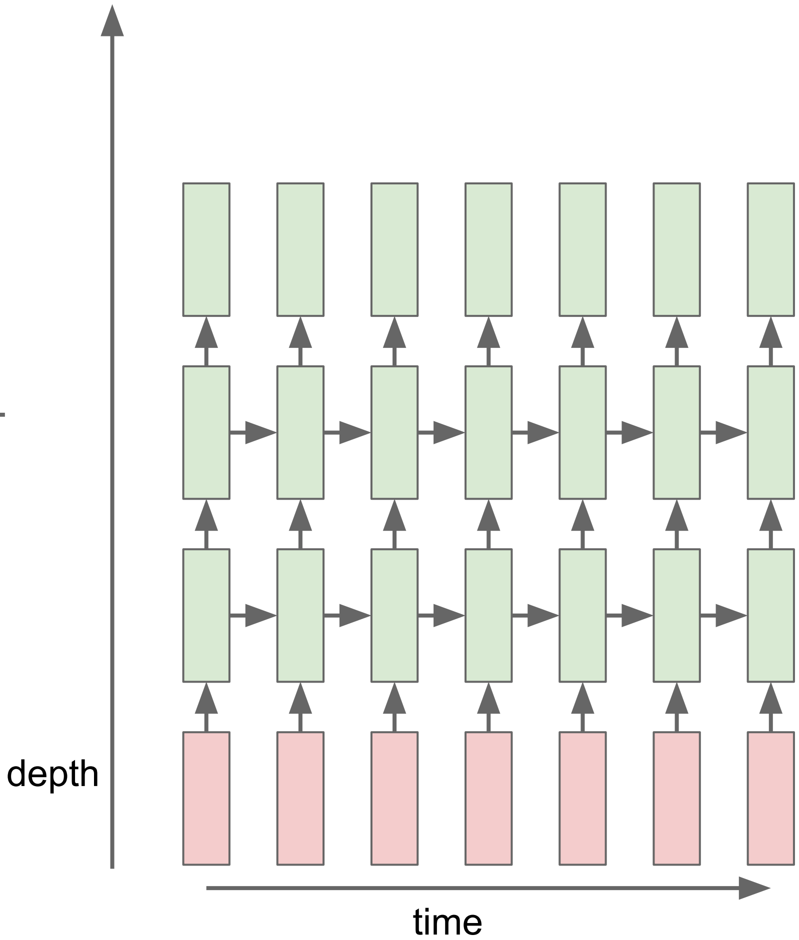
LSTM:

$W^l \ [4n \times 2n]$

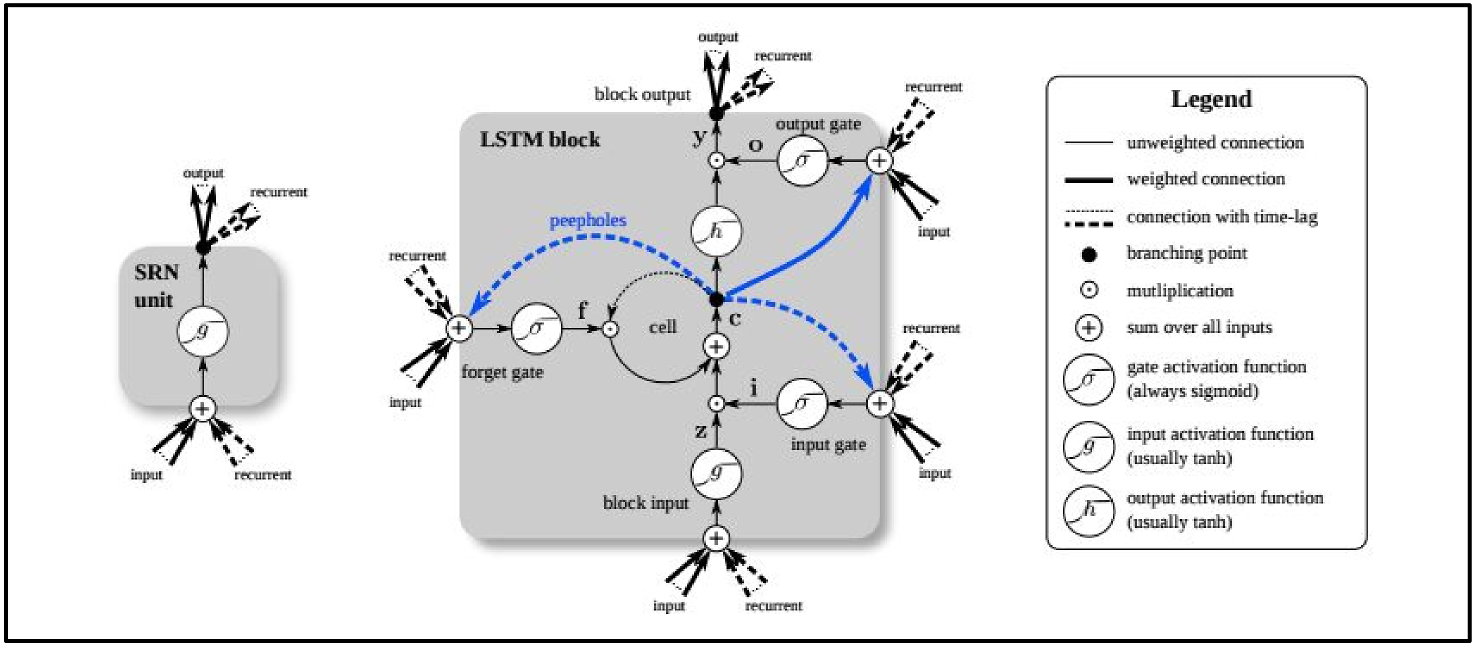
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

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

$$h_t^l = o \odot \tanh(c_t^l)$$



Long Short Term Memory (LSTM)



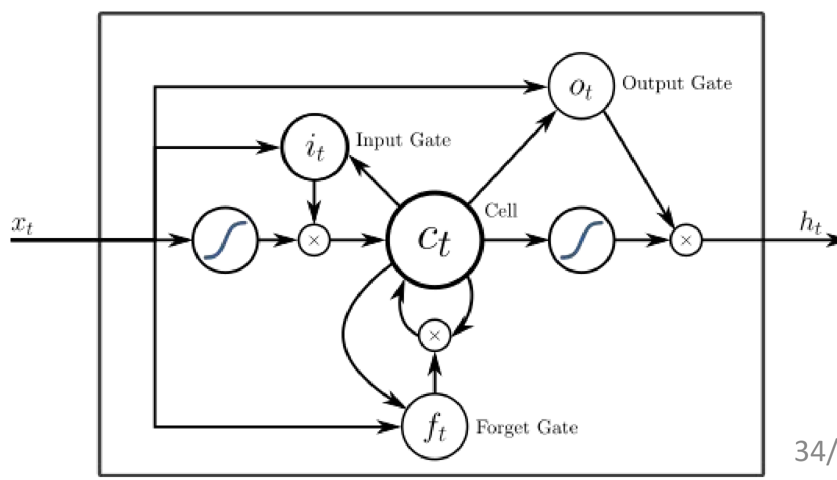
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

x

h

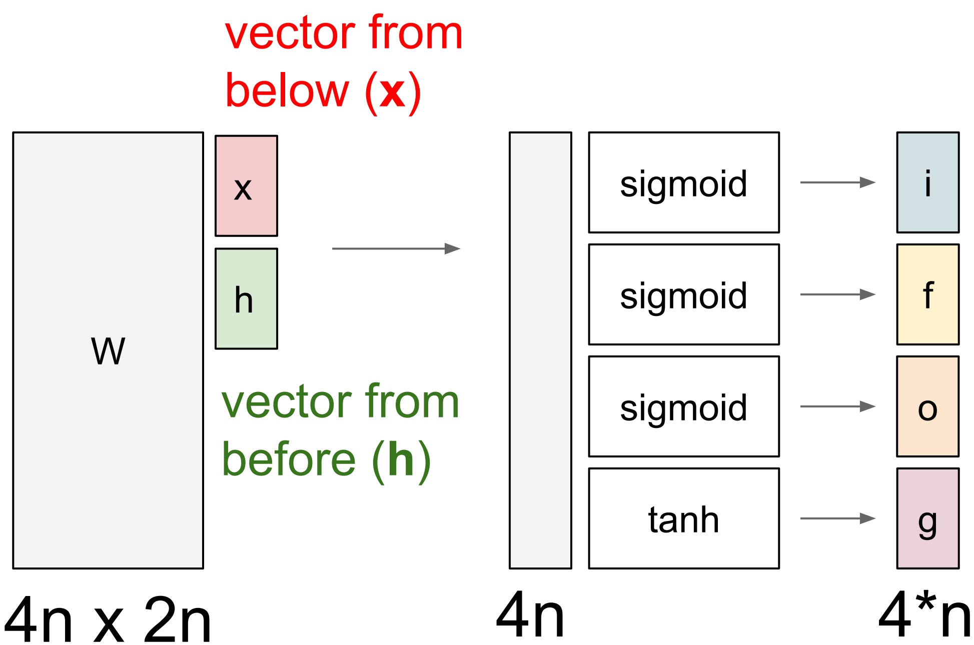
$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$



Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

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

$$h_t^l = o \odot \tanh(c_t^l)$$

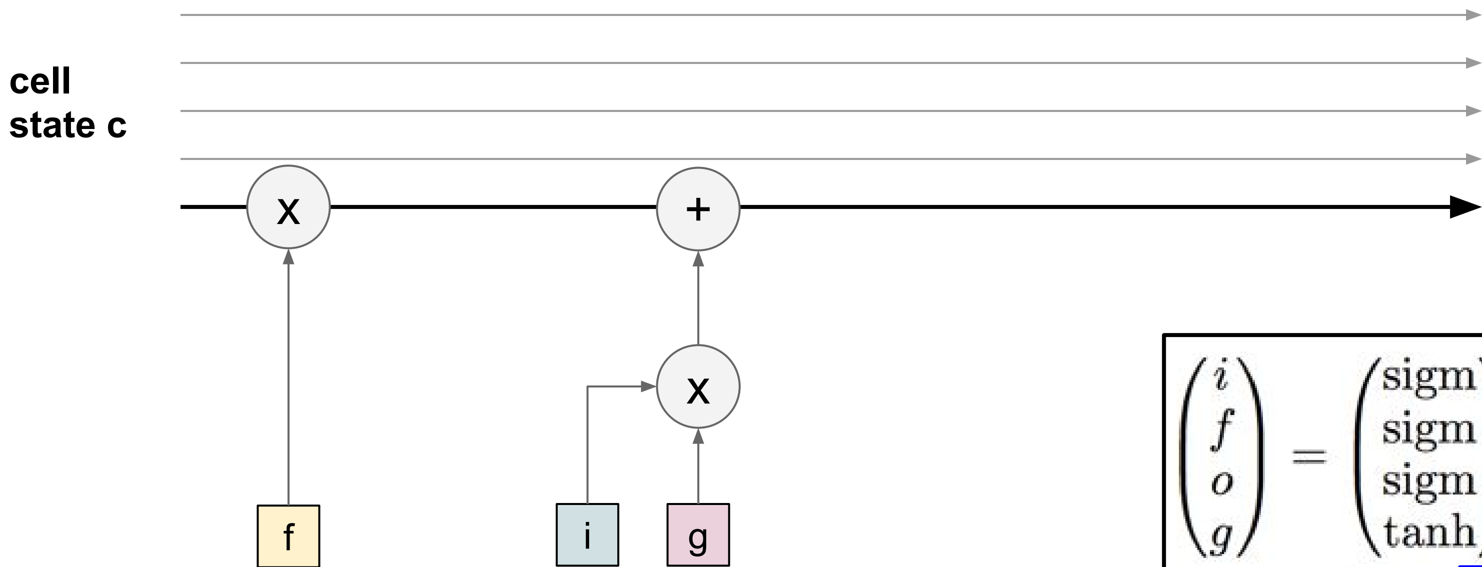
Long Short Term Memory (LSTM)

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

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

$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

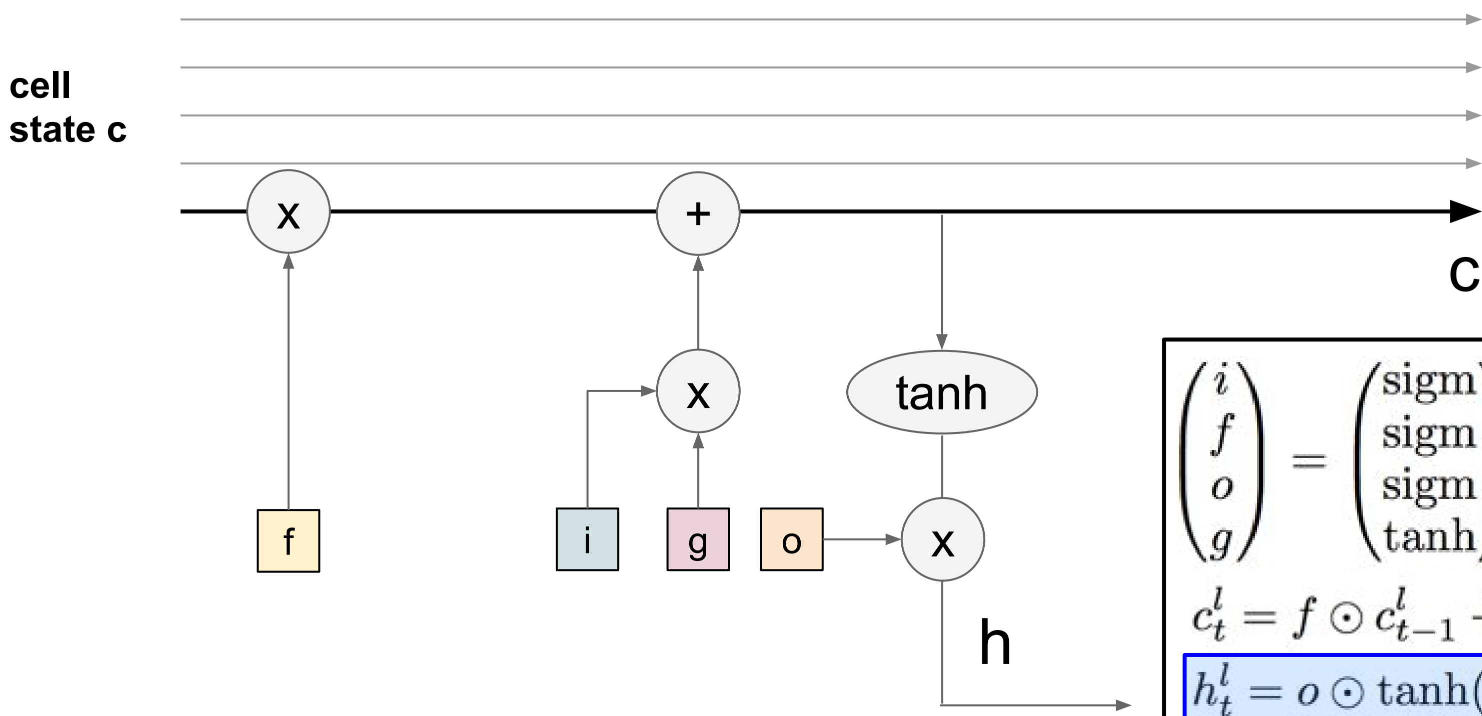


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

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

$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)

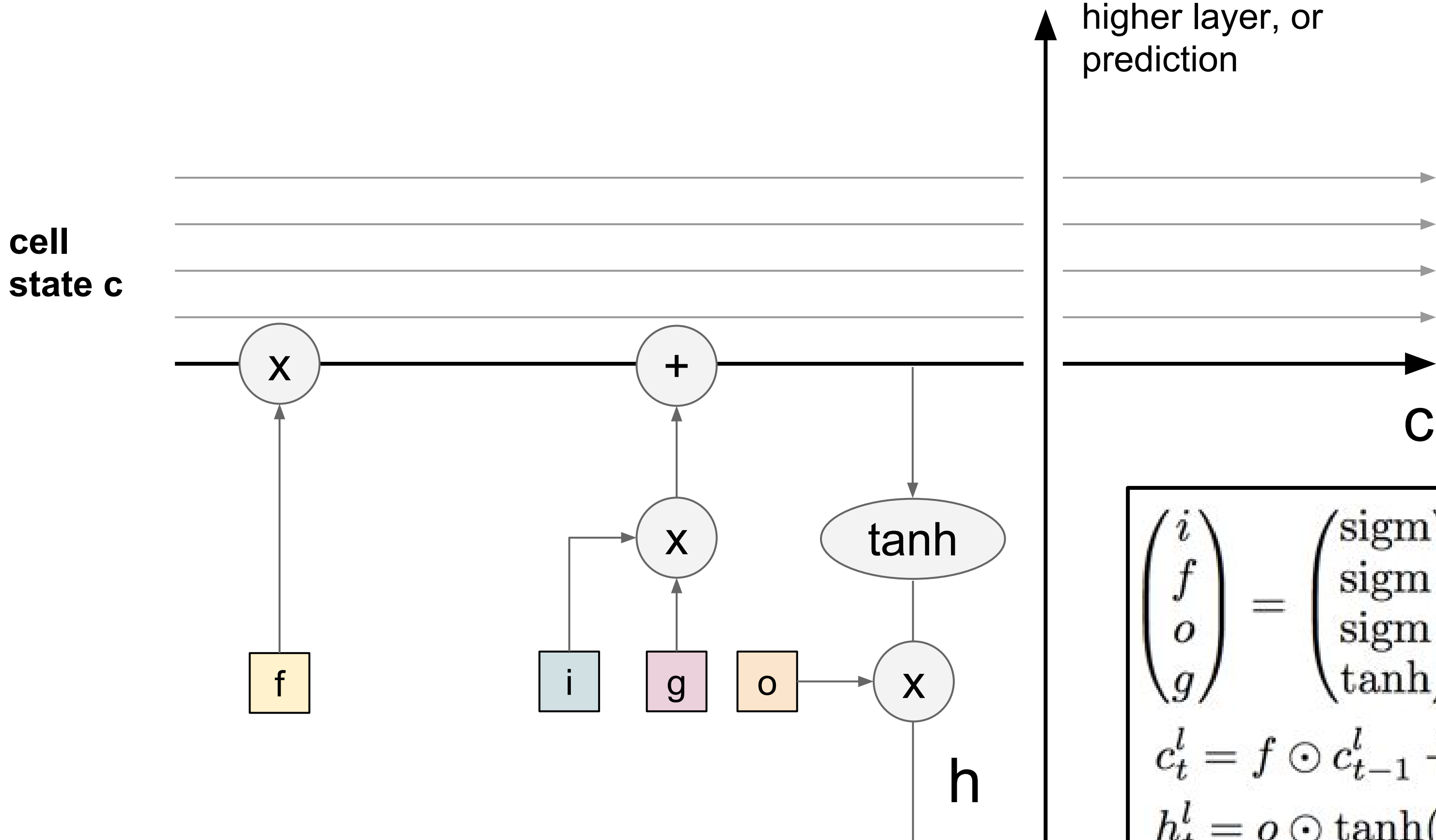


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

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

$$\boxed{h_t^l = o \odot \tanh(c_t^l)}$$

Long Short Term Memory (LSTM)

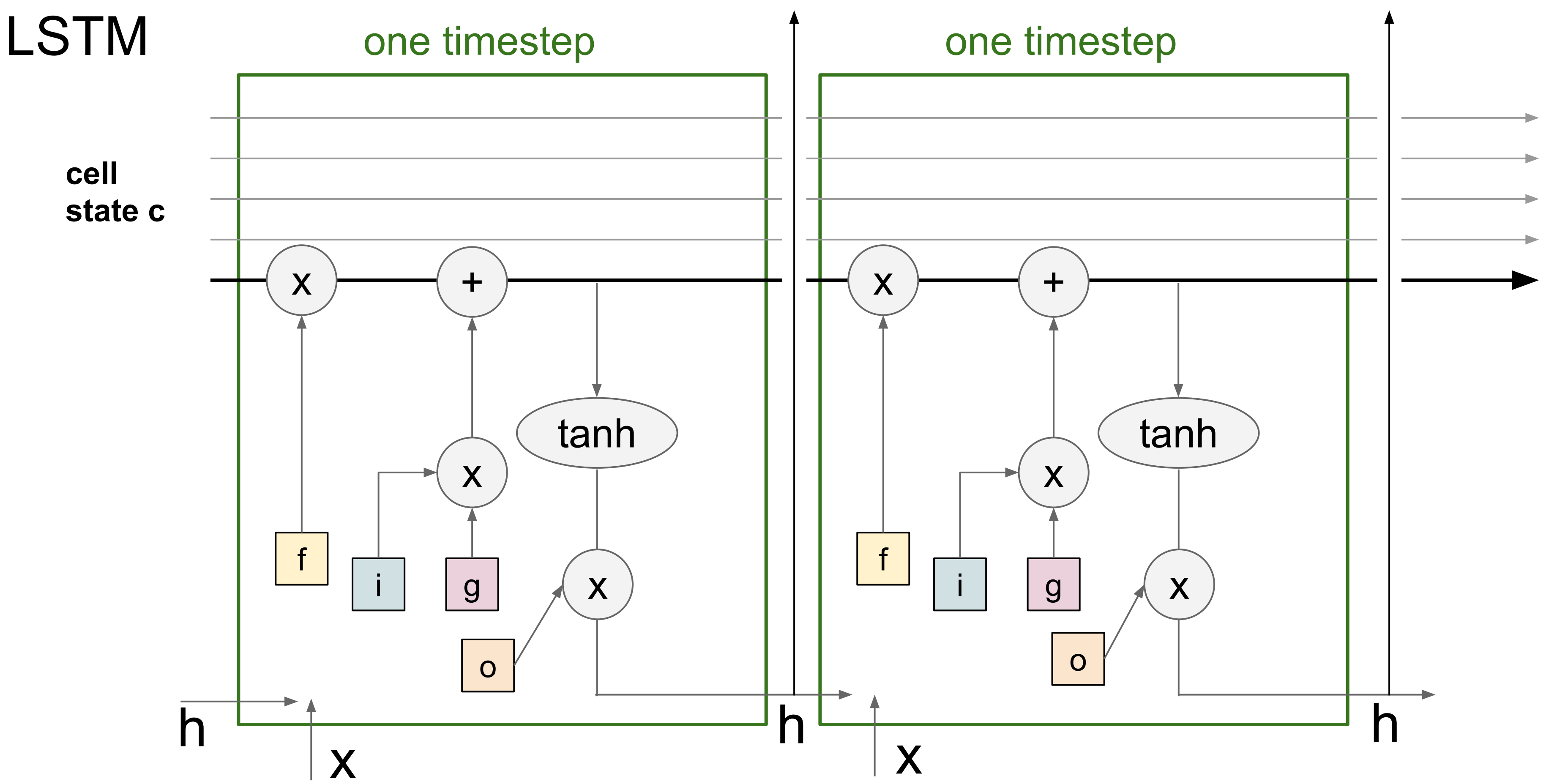


$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

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

$$h_t^l = o \odot \tanh(c_t^l)$$

Long Short Term Memory (LSTM)



UNIVERSITY OF SURREY

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

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

UNIVERSITY OF SURREY

Caffe

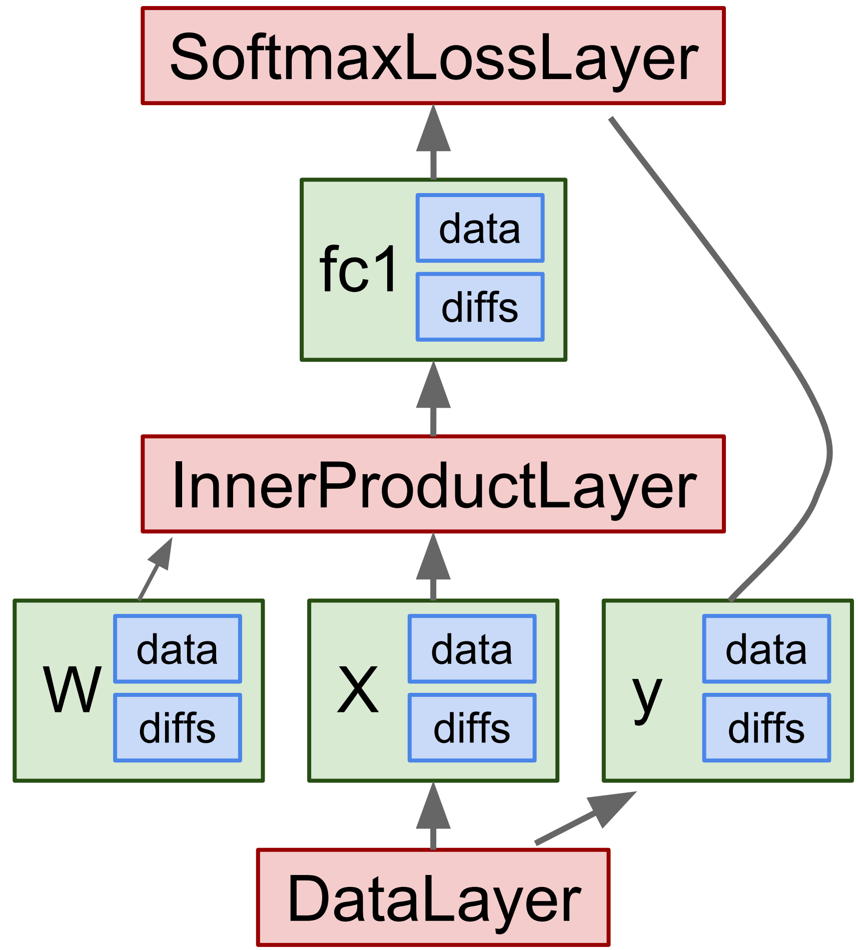
Main classes

Blob: Stores data and derivatives (header source)

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

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

Solver: Uses gradients to update weights (header source)



Caffe

Protocol Buffers

“Typed JSON”
from Google

Define “message types” in
.proto files

.proto file

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

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

Protocol Buffers

"Typed JSON" from Google

Define "message types" in .proto files

Serialize instances to text files (.prototxt)

.proto file

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

.prototxt file

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

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

Protocol Buffers

"Typed JSON" from Google

Define "message types" in .proto files

Serialize instances to text files (.prototxt)

Compile classes for different languages

.proto file

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

Java class

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

.prototxt file

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

C++ class

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

UNIVERSITY OF SURREY

Caffe

Protocol Buffers

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

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

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

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

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

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

  optional string train_net = 1; // Proto filename for the train net.
```

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

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

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

```
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
}
```

UNIVERSITY OF SURREY

Caffe

Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
  top: "data"
  top: "label"
  name: "data"
  type: HDF5_DATA
  hdf5_data_param {
    source: "examples/hdf5_classification/data/train.txt"
    batch_size: 10
  }
  include {
    phase: TRAIN
  }
}
layers {
  bottom: "data"
  top: "fc1"
  name: "fc1"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
```

Layers and Blobs often have same name!

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

UNIVERSITY OF SURREY

Caffe

Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
  top: "data"
  top: "label"
  name: "data"
  type: HDF5_DATA
  hdf5_data_param {
    source: "examples/hdf5_classification/data/train.txt"
    batch_size: 10
  }
  include {
    phase: TRAIN
  }
}
layers {
  bottom: "data"
  top: "fc1"
  name: "fc1"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
```

Layers and Blobs often have same name!

Learning rates (weight + bias)

Regularization (weight + bias)

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

Caffe

Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
  top: "data"
  top: "label"
  name: "data"
  type: HDF5_DATA
  hdf5_data_param {
    source: "examples/hdf5_classification/data/train.txt"
    batch_size: 10
  }
  include {
    phase: TRAIN
  }
}
layers {
  bottom: "data"
  top: "fc1"
  name: "fc1"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
```

Layers and Blobs often have same name!

Learning rates (weight + bias)

Regularization (weight + bias)

Number of output classes

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

UNIVERSITY OF SURREY

Caffe

Step 2: Define Net

```
name: "LogisticRegressionNet"
layers {
  top: "data"
  top: "label"
  name: "data"
  type: HDF5_DATA
  hdf5_data_param {
    source: "examples/hdf5_classification/data/train.txt"
    batch_size: 10
  }
  include {
    phase: TRAIN
  }
}
layers {
  bottom: "data"
  top: "fc1"
  name: "fc1"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
```

Layers and Blobs often have same name!

Set these to 0 to freeze a layer

Learning rates (weight + bias)

Regularization (weight + bias)

Number of output classes

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

UNIVERSITY OF SURREY

Caffe

Step 2: Define Net

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

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

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

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

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

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

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

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

Caffe

Step 2: Define Net (finetuning)

Original prototxt:

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

Same name:
weights copied

Pretrained weights:

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

Modified prototxt:

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

Caffe

Step 2: Define Net (finetuning)

Original prototxt:

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

Same name: weights copied

Pretrained weights:

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

Modified prototxt:

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

Caffe

Step 2: Define Net (finetuning)

Original prototxt:

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

Same name: weights copied

Pretrained weights:

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

Different name: weights reinitialized

Modified prototxt:

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

Caffe

Step 3: Define Solver

Write a prototxt file defining a SolverParameter

If finetuning, copy existing solver.prototxt file

- Change net to be your net
- Change snapshot_prefix to your output
- Reduce base learning rate (divide by 100)
- Maybe change max_iter and snapshot

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

Caffe

Step 4: Train!

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

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

Caffe

Step 4: Train!

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

`-gpu -1` for CPU mode

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

Caffe

Step 4: Train!

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

`-gpu all` for multi-GPU data parallelism

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

Caffe

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

Torch

Lua

High level scripting language, easy to interface with C

Similar to Javascript:

- One data structure:
  - table == JS object
- Prototypical inheritance
  - metatable == JS prototype
- First-class functions

Some gotchas:

- 1-indexed =(
- Variables global by default =(
- Small standard library

Learn Lua in 15 Minutes

more or less

For a more in-depth Lua tutorial, watch this video or check out a transcript of the video.

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

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

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

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

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

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

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

Torch

Tensors

Torch tensors are just like numpy arrays

Torch

Tensors

Torch tensors are just like numpy arrays

```
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
```

UNIVERSITY OF SURREY

Torch

Tensors

Torch tensors are just like numpy arrays

```python
import numpy as np

# Simple feedforward network (no biases) in numpy

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

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

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

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

print scores
```

```lua
require 'torch'

-- Simple feedforward network (no biases) in torch

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

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

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

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

print(scores)
```

Torch

Tensors

Like numpy, can easily change data type:

```python
import numpy as np


# Simple feedforward network (no biases) in numpy

dtype = np.float32  # Use 32-bit floats

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

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

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

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

print scores
```

```lua
require 'torch'


-- Simple feedforward network (no biases) in torch

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

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

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

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

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

print(scores)
```

Torch

Tensors

Unlike numpy, GPU is just a datatype away:

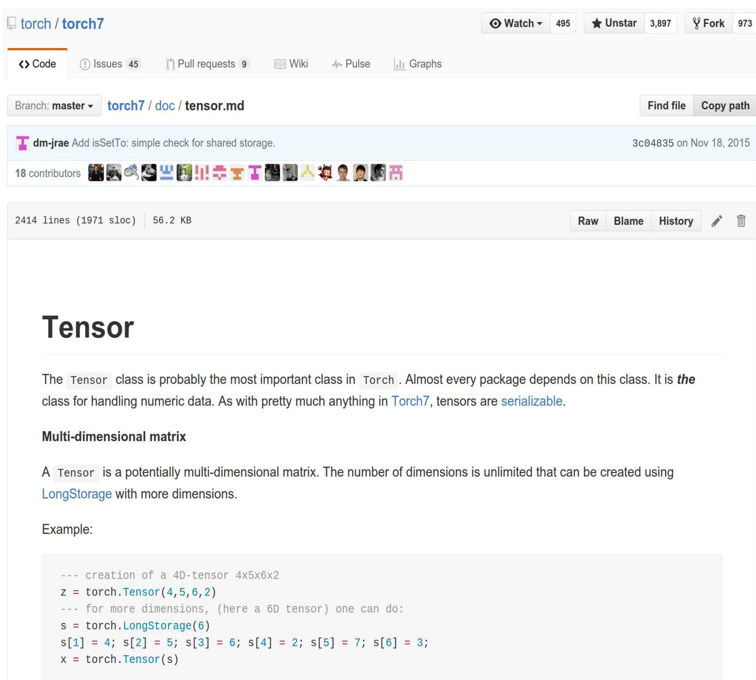
```python
import numpy as np


# Simple feedforward network (no biases) in numpy

dtype = np.float32  # Use 32-bit floats

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

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

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

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

print scores
```

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

-- Simple feedforward network (no biases) in torch

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

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

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

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

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

print(scores)
```

Torch

Tensors

Documentation on GitHub:


torch / torch7 Watch 495 Unstar 3,897 Fork 973

Code Issues 45 Pull requests 9 Wiki Pulse Graphs

Branch: master torch7 / doc / tensor.md Find file Copy path

dm-jrae Add isSetTo: simple check for shared storage. 3c04835 on Nov 18, 2015

18 contributors

2414 lines (1971 sloc) 56.2 KB Raw Blame History

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.

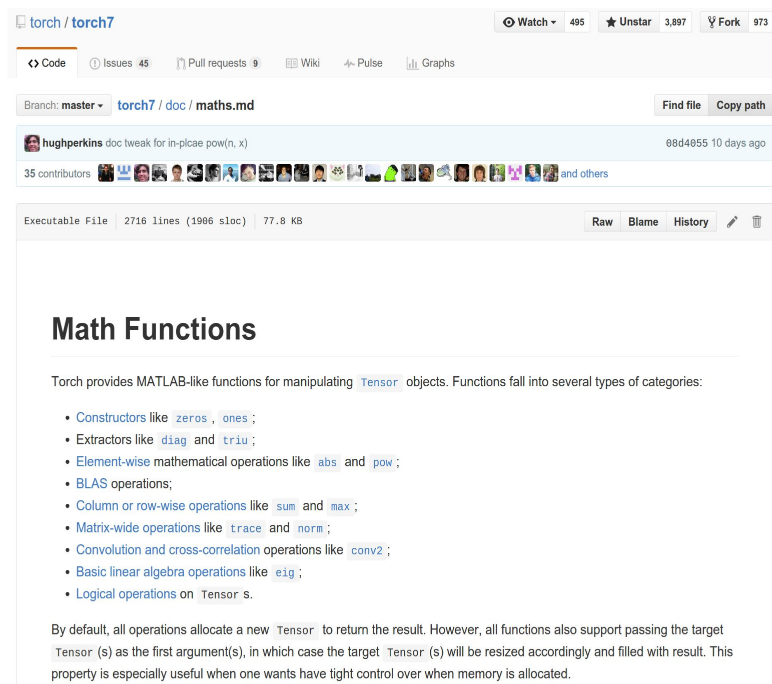
Multi-dimensional matrix

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

Example:

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

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


torch / torch7 Watch 495 Unstar 3,897 Fork 973

Code Issues 45 Pull requests 9 Wiki Pulse Graphs

Branch: master torch7 / doc / maths.md Find file Copy path

hughperkins doc tweak for in-plcae pow(n, x) 08d4055 10 days ago

35 contributors and others

Executable File 2716 lines (1906 sloc) 77.8 KB Raw Blame History

Math Functions

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

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

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

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

Torch

nn

nn module lets you easily build and train neural nets

```
require 'torch'
require 'nn'


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

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

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

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

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

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

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

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

Torch

nn

nn module lets you easily build and train neural nets

Build a two-layer ReLU net

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


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

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

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

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

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

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

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

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

Torch

nn

nn module lets you easily build and train neural nets

Get weights and gradient for entire network

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


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

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

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

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

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

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

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

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

Torch

nn

nn module lets you easily build and train neural nets

Use a softmax loss function

```
require 'torch'
require 'nn'


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

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

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

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

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

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

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

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

```

Torch

nn

nn module lets you easily build and train neural nets

Generate random data

```
require 'torch'
require 'nn'


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

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

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

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

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

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

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

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

```

Torch

nn

nn module lets you easily build and train neural nets

Forward pass: compute scores and loss

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


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

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

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

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

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

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

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

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

UNIVERSITY OF SURREY

Torch

nn

nn module lets you easily build and train neural nets

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

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


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

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

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

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

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

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

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

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

Torch

nn

nn module lets you easily build and train neural nets

Update: Make a gradient descent step

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


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

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

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

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

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

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

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

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

Torch

cunn

Running on GPU is easy:

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

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

local dtype = 'torch.CudaTensor'

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

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

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

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

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

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

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

Torch

cunn

Running on GPU is easy:

Import a few new packages

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

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

local dtype = 'torch.CudaTensor'

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

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

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

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

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

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

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

Torch

cunn

Running on GPU is easy:

Import a few new packages

Cast network and criterion

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

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

local dtype = 'torch.CudaTensor'

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

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

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

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

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

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

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

Torch

cunn

Running on GPU is easy:

Import a few new packages

Cast network and criterion

Cast data and labels

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

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

local dtype = 'torch.CudaTensor'

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

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

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

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

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

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

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

Torch

optim

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

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

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

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

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

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

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

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

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

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

  return loss, grad_weights
end

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

Torch

optim

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

Import optim package

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

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

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

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

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

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

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

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

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

  return loss, grad_weights
end

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

Torch

optim

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

Import optim package

Write a callback function that returns loss and gradients

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

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

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

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

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

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

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

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

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

  return loss, grad_weights
end

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

Torch

optim

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

Import optim package

Write a callback function that returns loss and gradients

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

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

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

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

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

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

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

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

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

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

  return loss, grad_weights
end

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

Torch

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

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

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

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

Torch

Modules

Caffe has Nets and Layers; Torch just has Modules

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

updateOutput: Forward pass; compute output

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

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

   return self.output
end
```

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

Torch

Modules

Caffe has Nets and Layers; Torch just has Modules

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

updateGradInput: Backward; compute gradient of input

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

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

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

      return self.gradInput
   end
end
```

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

Torch

Modules

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')
```

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

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

UNIVERSITY OF SURREY

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

- ClassSimplexCriterion.lua
- Concat.lua
- ConcatTable.lua
- Container.lua
- Contiguous.lua
- Copy.lua
- Cosine.lua
- CosineDistance.lua
- CosineEmbeddingCriterion.lua
- Criterion.lua
- CriterionTable.lua
- CrossEntropyCriterion.lua
- DepthConcat.lua
- DistKLDivCriterion.lua
- DotProduct.lua
- Dropout.lua
- ELU.lua

https://github.com/torch/nn

Torch

Modules

Writing your own modules is easy!

TimesTwo.lua

```lua
require 'nn'

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


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


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


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

times_two_example.lua

```lua
require 'nn'

require 'TimesTwo'

local times_two = nn.TimesTwo()

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

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

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

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

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

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

Modules

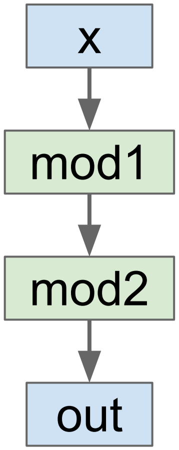
Container modules allow you to combine multiple modules

Torch

Modules

Container modules allow you to combine multiple modules

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

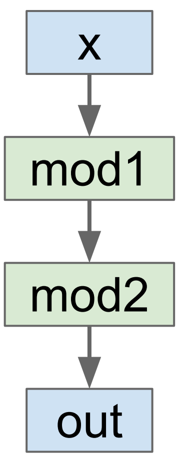


Torch

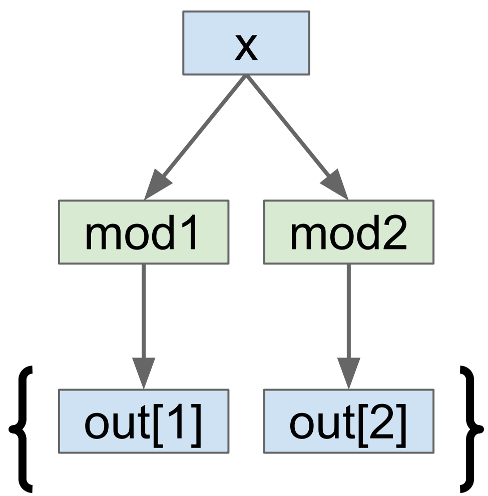
Modules

Container modules allow you to combine multiple modules

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




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

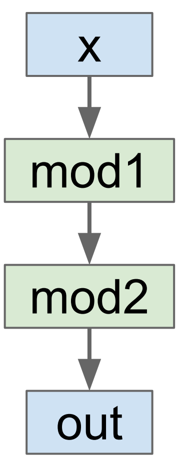


Torch

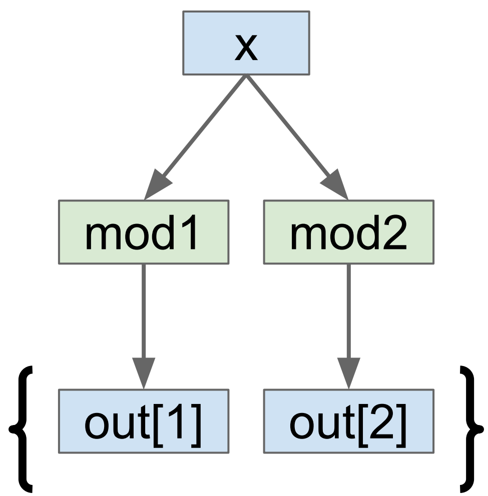
Modules

Container modules allow you to combine multiple modules

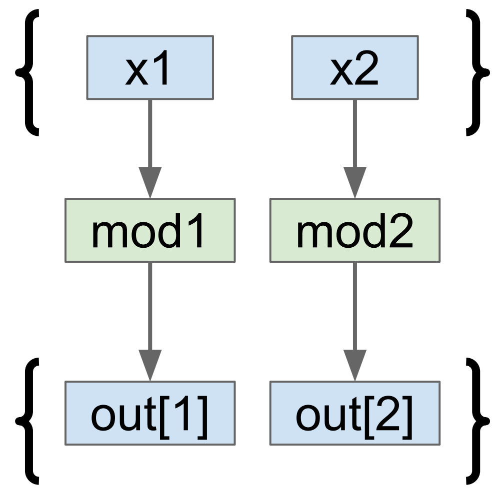
```lua
local seq = nn.Sequential()
seq:add(mod1)
seq:add(mod2)
local out = seq:forward(x)
```




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




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



Torch

nngraph

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

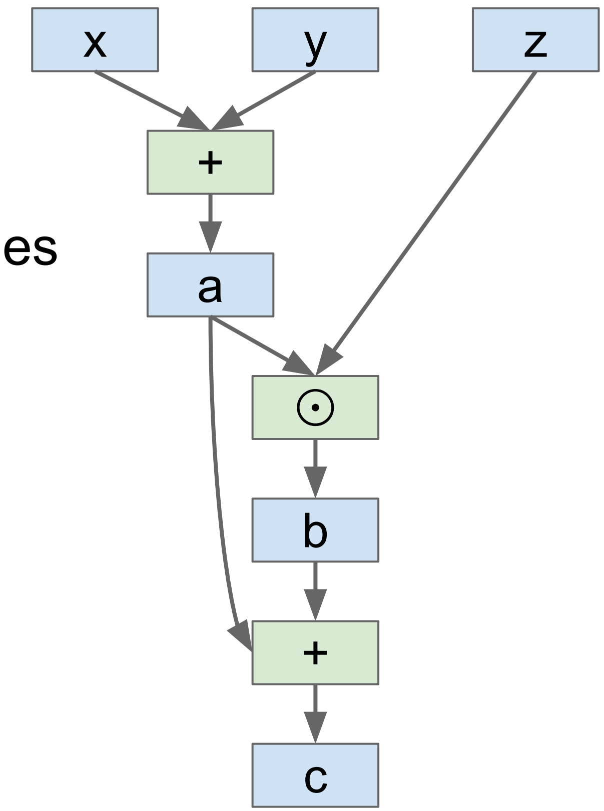
Inputs: x, y, z
Outputs: c
$a = x + y$
$b = a \odot z$
$c = a + b$

Torch

nngraph

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

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

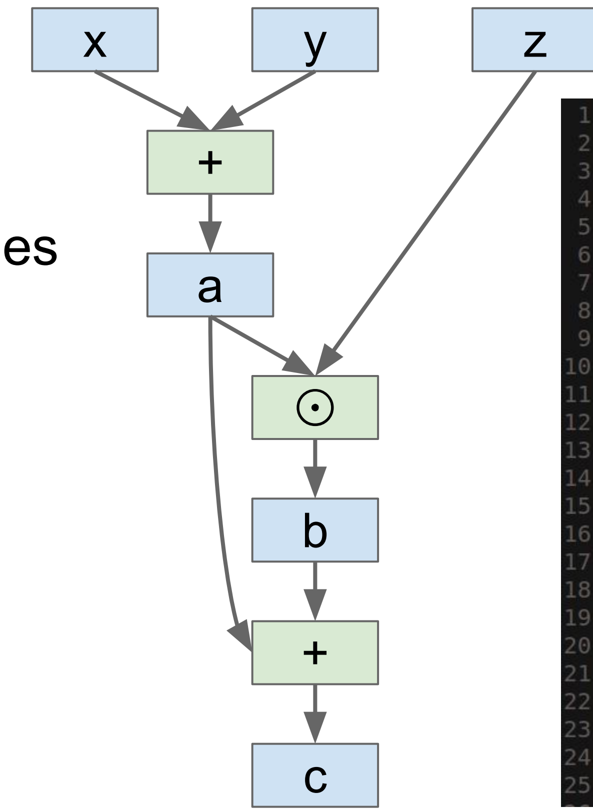


Torch

nngraph

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

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




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

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

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

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

local mod = build_module()

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

local c = mod:forward({x, y, z})
```

Torch

Pretrained Models

loadcaffe: Load pretrained Caffe models: AlexNet, VGG, some others
https://github.com/szagoruyko/loadcaffe

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

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

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

Torch

Package Management

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

(Similar to pip install from Python)

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

Torch: Other useful packages

torch.cudnn: Bindings for NVIDIA cuDNN kernels
https://github.com/soumith/cudnn.torch

torch-hdf5: Read and write HDF5 files from Torch
https://github.com/deepmind/torch-hdf5

lua-cjson: Read and write JSON files from Lua
https://luarocks.org/modules/luarocks/lua-cjson

cltorch, clnn: OpenCL backend for Torch, and port of nn
https://github.com/hughperkins/cltorch, https://github.com/hughperkins/clnn

torch-autograd: Automatic differentiation; sort of like more powerful nngraph, similar to Theano or TensorFlow
https://github.com/twitter/torch-autograd

fbcunn: Facebook: FFT conv, multi-GPU (DataParallel, ModelParallel)
https://github.com/facebook/fbcunn

Torch

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

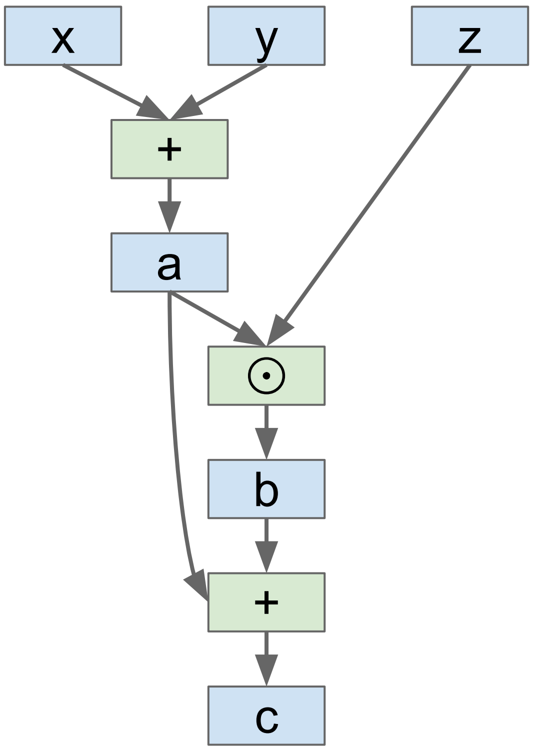
From Yoshua Bengio’s group at University of Montreal

Embracing computation graphs, symbolic computation

High-level wrappers: Keras, Lasagne

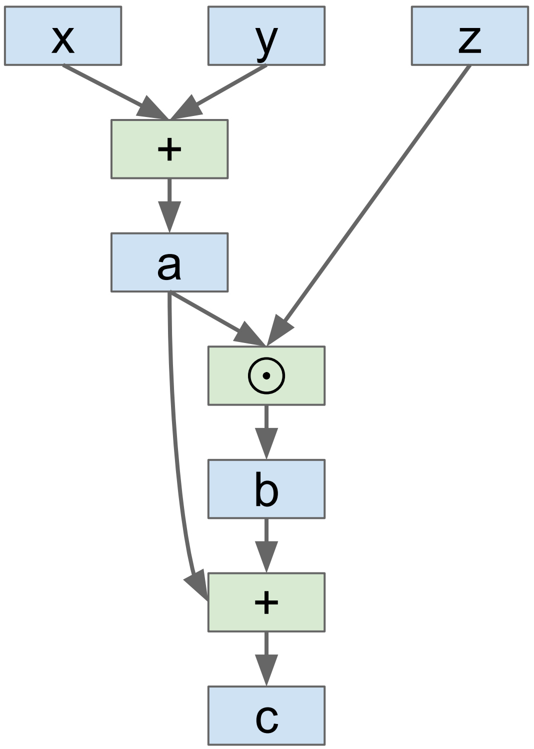
Theano

Computational Graphs



Theano

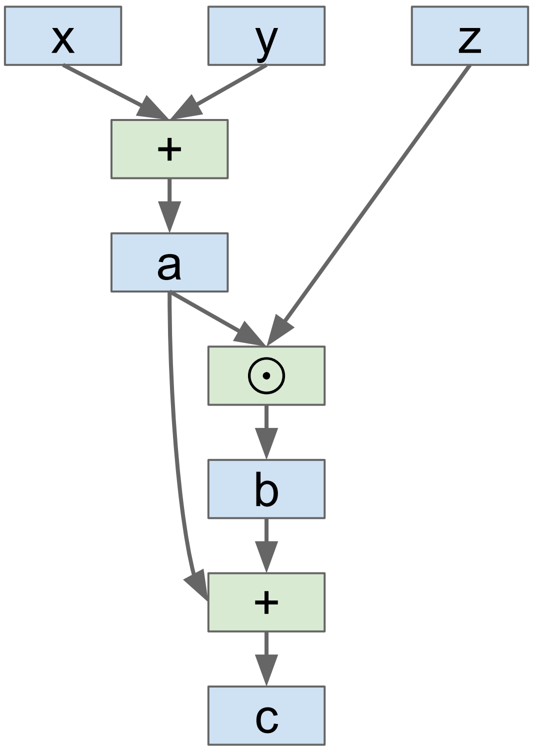
Computational Graphs




```
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

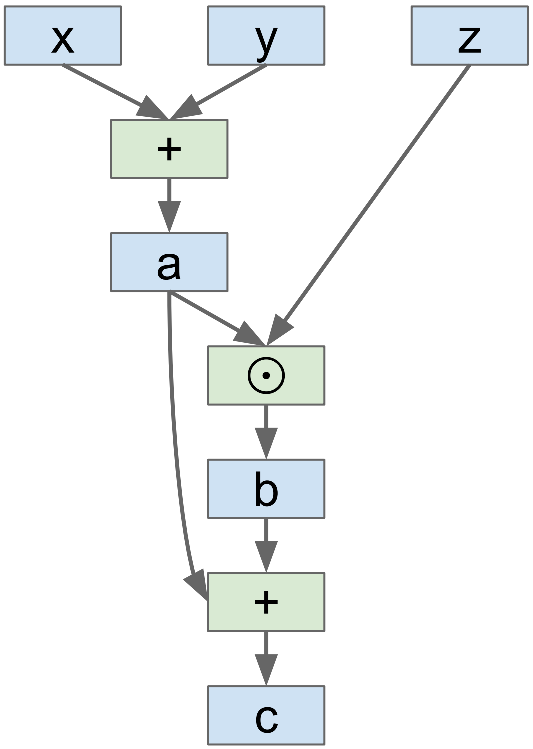



```python
import theano
import theano.tensor as T

# 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

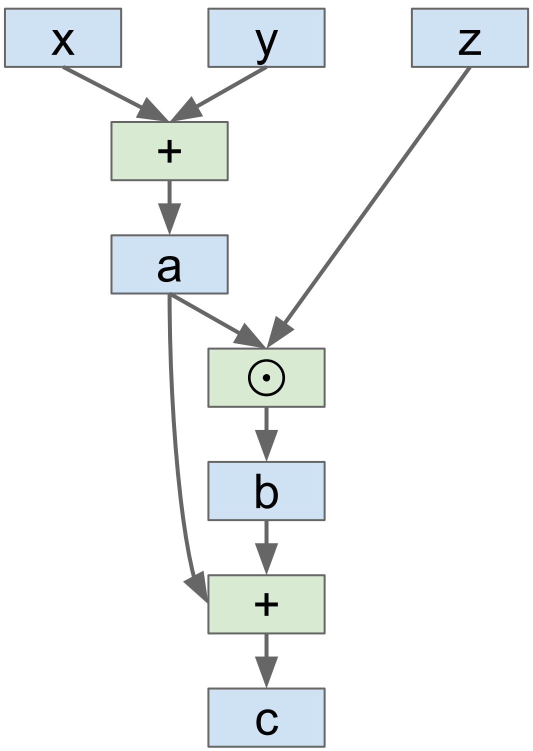



```python
import theano
import theano.tensor as T

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

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

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

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

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

Compute intermediates and outputs symbolically

Theano

Computational Graphs



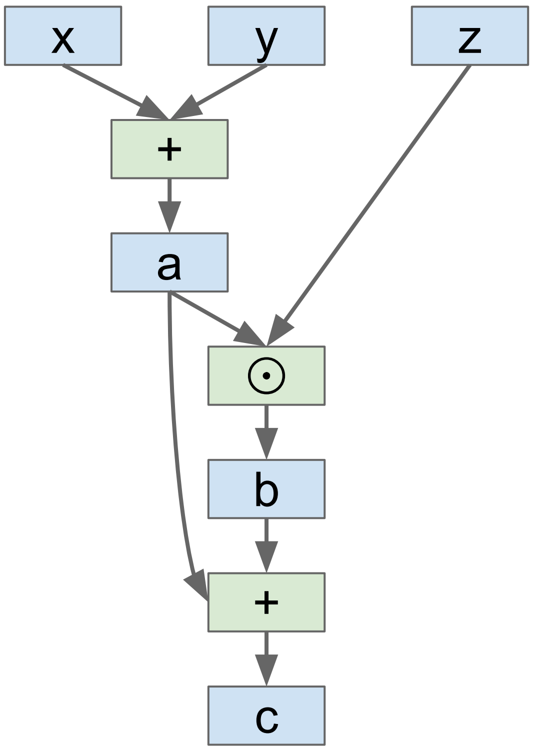

```
import theano
import theano.tensor as T

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

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

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

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

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

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

University of Surrey

Theano

Computational Graphs

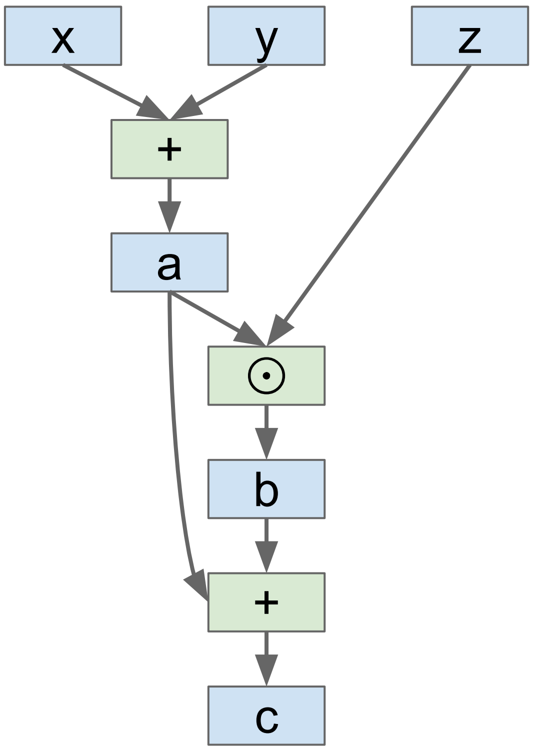



```
import theano
import theano.tensor as T

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

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

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

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

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

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

Theano

Computational Graphs




```
import theano
import theano.tensor as T

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

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

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

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

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

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

Theano

Simple Neural Net

```
import theano
import theano.tensor as T

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

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

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

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

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

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

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

Theano

Simple Neural Net

Define symbolic variables:

- x = data
- y = labels
- w1 = first-layer weights
- w2 = second-layer weights

```
import theano
import theano.tensor as T

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

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

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

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

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

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

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

Theano

Simple Neural Net

Forward: Compute scores (symbolically)

```
import theano
import theano.tensor as T

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

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

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

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

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

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

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

UNIVERSITY OF SURREY

Theano

Simple Neural Net

Forward: Compute probs, loss (symbolically)

```python
import theano
import theano.tensor as T

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

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

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

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

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

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

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

UNIVERSITY OF SURREY

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)
ww1 = 1e-3 * np.random.randn(D, H)
ww2 = 1e-3 * np.random.randn(H, C)

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

Theano

Simple Neural Net

Stuff actual numpy arrays into the function

```
import theano
import theano.tensor as T

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

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

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

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

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

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

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

Theano

Computing Gradients

```python
import theano
import theano.tensor as T

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

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

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

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

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

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

Theano

Computing Gradients

```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

```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

```
import theano
import theano.tensor as T

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

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

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

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

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

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

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

learning_rate = 1e-1
for t in xrange(50):
  loss, scores, 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

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

```
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

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

```
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

Create Variables to hold weights; similar to Theano shared variables

Initialize variables with numpy arrays

```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
```

UNIVERSITY OF SURREY

TensorFlow

TensorFlow: Two-Layer Net

Forward: Compute scores, probs, loss (symbolically)

```
import tensorflow as tf
import numpy as np

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

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

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

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

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

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

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

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

TensorFlow

TensorFlow: Two-Layer Net

Running train_step will use SGD to minimize loss

```
import tensorflow as tf
import numpy as np

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

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

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

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

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

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

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

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

TensorFlow: Two-Layer Net

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

```
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

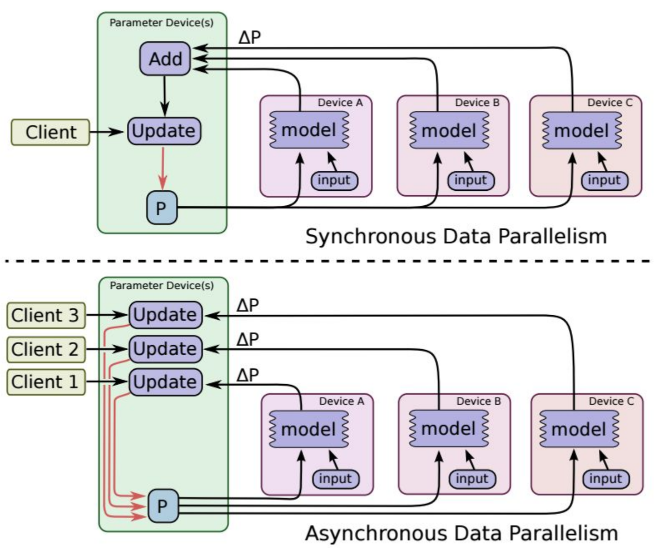
TensorFlow: Two-Layer Net

```
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
```

Actually train the model

TensorFlow: Multi-GPU

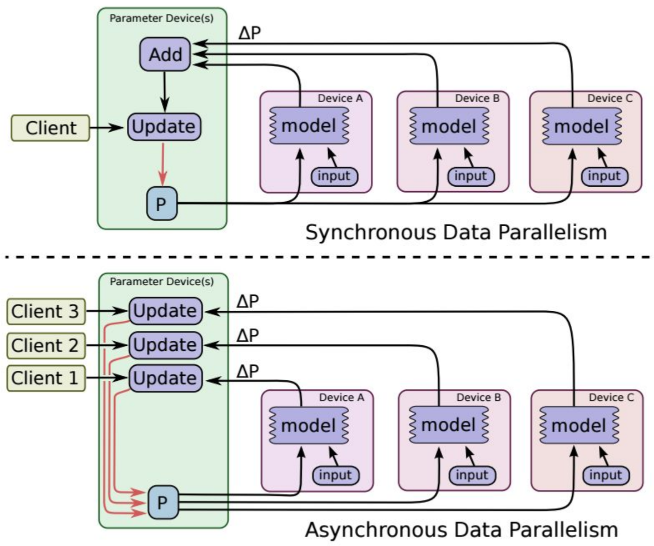
Data parallelism:
synchronous or asynchronous



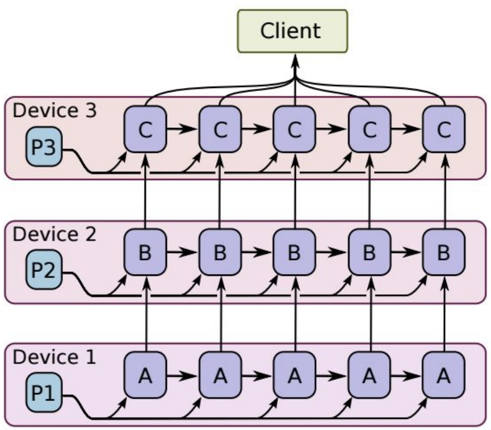
TensorFlow

TensorFlow: Multi-GPU

Data parallelism:
synchronous or asynchronous



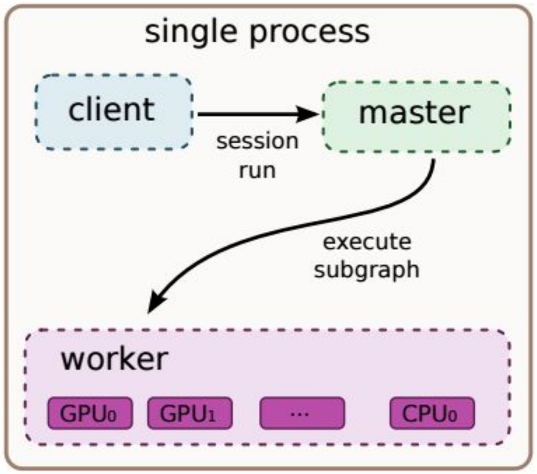
Model parallelism:
Split model across GPUs

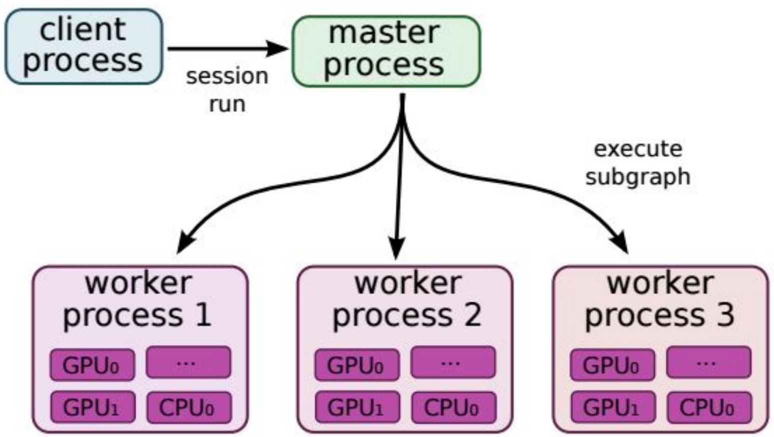


TensorFlow: Distributed

Single machine:
Like other frameworks

Many machines:
Not open source (yet) =(





TensorFlow: 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

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

Any Question???
Thanks