TTIC 31230, Fundamentals of Deep Learning

David McAllester, Winter 2018

Language Modeling

Machine Translation

Attention

Overview of PyTorch

Review: Controlling Gradients

Relu Activation Functions

Xavier Initialization

Batch Normalization

Gradient Clipping

Skip Connections (a.k.a Highway Paths)

Spatial Batch Normalization

Given a spatial tensor (CNN tensor) $x[b,i,j,c_x]$ we normalize as follows.

$$\tilde{x}[b,i,j,c_x] = \frac{x[b,i,j,c_x] - \hat{\mu}[c_x]}{\hat{\sigma}[c_x]}$$

$$\hat{\mu}[c_x] = \frac{1}{BIJ} \sum_{b,i,j} x[b,i,j,c_x]$$

$$\hat{\sigma}[c_x] = \sqrt{\frac{1}{BIJ-1} \sum_{b,i,j} (x[b,i,j,c_x] - \hat{\mu}[c_x])^2}$$

Adding an Affine Transformation

$$\breve{x}[b,i,j,c_x] = \gamma[c_x]\tilde{x}[b,i,j,c_x] + \beta[c_x]$$

Here $\gamma[c_x]$ and $\beta[c_x]$ are parameters of the batch normalization operation.

This allows the batch normlization to learn an arbitrary affine transformation (offset and scaling).

It can even undo the normaliztion.

Batch Normalization

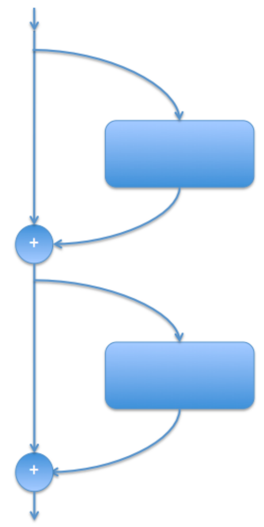
Batch Normalization is is empirically successful in CNNs.

Not so successful in RNNs.

It is typically used just prior to a nonlinear activation function.

It is intuitively justified in terms of “internal covariate shift”: as the inputs to a layer change the zero mean unit variance property underlying Xavier initialization are maintained.

Highway Architectures (Skip Connections)



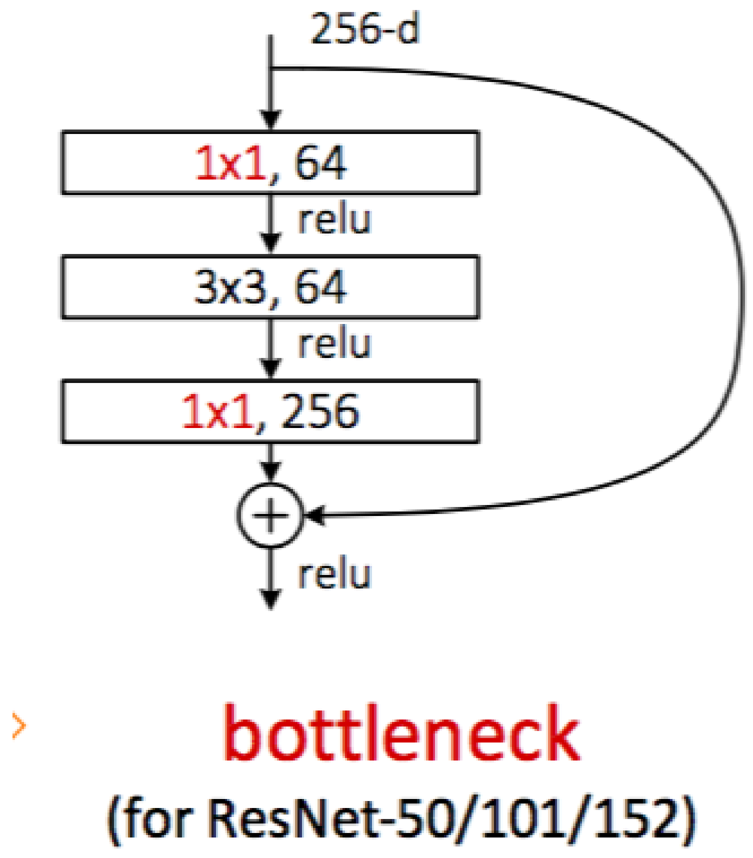
Residual: $y^{\ell+1} = y^{\ell} + d^{\ell}$

LSTM: $h^{t+1} = f^t \odot h^t + i^t \odot d^t$

GRU: $h^{t+1} = f^t \odot h^t + (1 - f^t) \odot d^t$

$\odot$ is Hadamard product — NumPy element-wise product.

Expressive Power

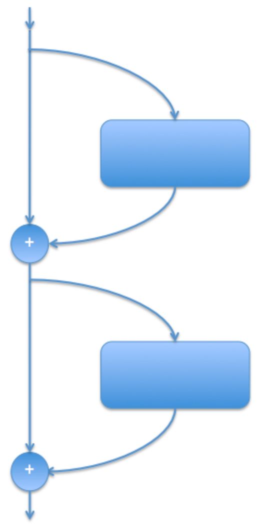


[Kaiming He]

This architecture can express fairly arbitrary state updates.

Each layer has data-flow parameters.

Gating gives Data-Dependent Data Flow



Residual: $y^{\ell+1} = y^{\ell} + d^{\ell}$

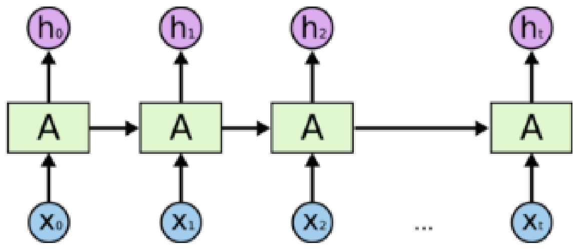
LSTM: $h^{t+1} = f^{t} \odot h^{t} + i^{t} \odot d^{t}$

GRU: $h^{t+1} = f^{t} \odot h^{t} + (1 - f^{t}) \odot d^{t}$

Resnet has data-flow parameters at each layer.

Gated RNNs use the same parameters at each time step but use gating for data-flow control.

Vanilla RNNs

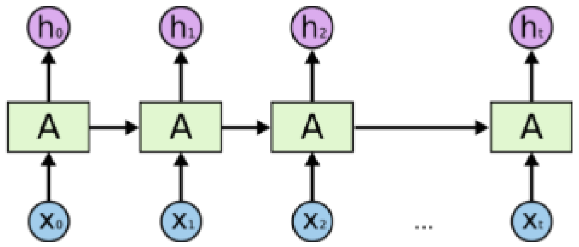


[Christopher Olah]

$$h^{t+1} = \tanh(W^h[h^t, x^t] + \beta)$$

where $[x, y]$ denotes concatenation.

Update Gate RNN (UGRNN)



[Christopher Olah]

$$h^{t+1} = f^t \odot h^t + (1 - f^t) \odot d^t$$

$$f^t = \sigma(W^f[x^t, h^t] + b^f)$$

$$d^t = \tanh(W^d[x^t, h^t] + b^d)$$

Applications of RNNs

Language Modeling

Machine Translation

Speech Recognition

Reading Comprenhesion

⋮

Language Modeling

We are given a sequence x^1, ..., x^T and we want to compute a model probability.

$$Q_\Phi(x_1, \ldots, x^T)$$

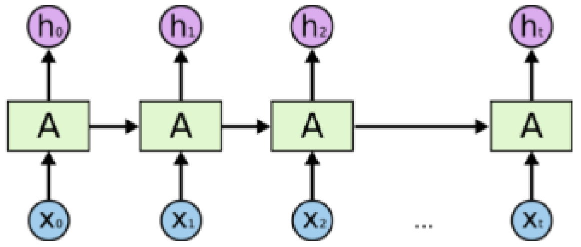
Here x^t can be either characters or words.

We represent the probability as a product of conditionals.

$$Q_\Phi(x_1, \ldots, x^T) = \prod_t Q_\Phi(x^t \mid x^1, \ldots, x^{t-1})$$

A model of this form is called **Autoregressive** — a model (regression) is used to predict the next element from the previous elements.

RNN Language Modeling



[Christopher Olah]

$$Q_\Phi(x^{t+1} \mid x^1, \ldots, x^t) = \mathrm{Softmax}(W^o\, h^t)$$

For word language models this softmax is the computation bottleneck in training.

Standard Measures of Performance

Bits per Character: For character language models performance is measured in bits per character. Typical numbers are slightly over one bit per character.

Perplexity: It would be natural to measure word language models in bits per word. However, it is traditional to measure then in perplexity which is defined to be 2^b where b is bits per world. Perplexities of about 60 are typical.

According to Quora there are 4.79 letters per word. 1 bit per character (including space characters) gives a perplexity of $2^{5.79}$ or 55.3.

Sampling From an Autoregressive Model

To sample a sentence

$$x^1, \ldots, x^T, \texttt{<eos>}$$

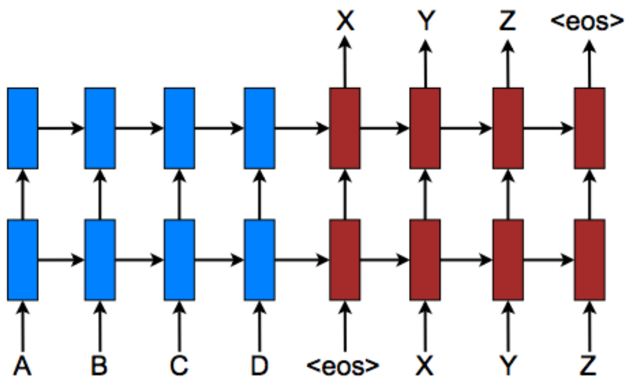
we sample x^t from

$$Q_\Phi(x^t | x^1, \ldots, x^{t-1})$$

until we get `<eos>`.

Sampling from an autoregressive language model plays a key role in machine translation.

Machine Translation



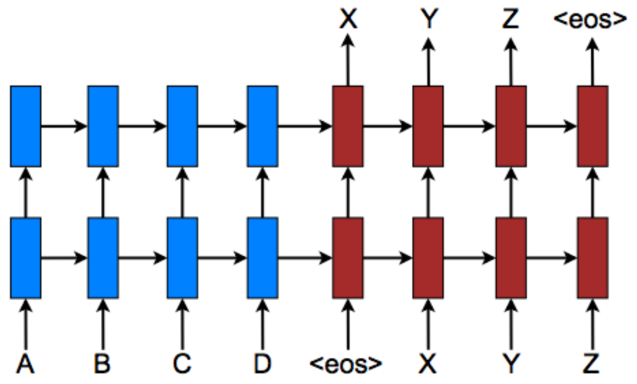
[Figure from Luong et al.]

$$DCBA \Rightarrow XYX$$

Translation is a **sequence to sequence** (seq2seq) task.

Sequence to Sequence Learning with Neural Networks, Sutskever, Vinyals and Le, NIPS 2014, arXiv Sept 10, 2014.

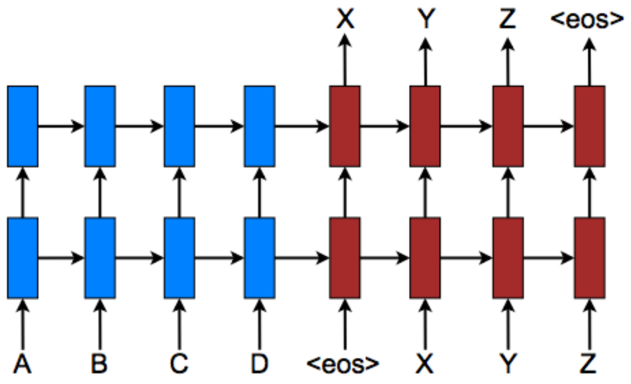
Machine Translation



[Figure from Luong et al.]

The input sentence is represented by a "thought vector" produced by an RNN.

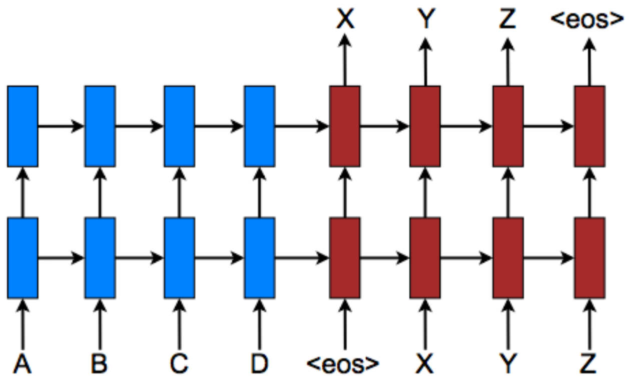
Machine Translation



[Figure from Luong et al.]

The thought vector is the initial state vector used in “sampling” a translation.

Machine Translation

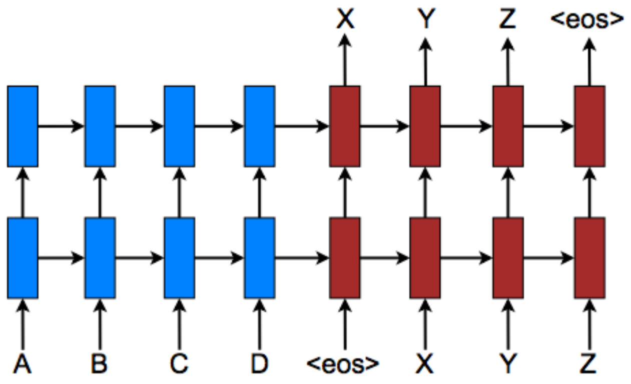


[Figure from Luong et al.]

For the training pair $DCBA \Rightarrow XYX$ the training loss is

$$-\log\ Q_\Phi(XYZ \mid DCBA)$$

Machine Translation



[Figure from Luong et al.]

In Sutskever et al. (2014) the LSTMs are layered 4 deep.

Training Details (Sutskever et al. (2014))

We used deep LSTMs with 4 layers, with 1000 cells at each layer and 1000 dimensional word embeddings, with an input vocabulary of 160,000 and an output vocabulary of 80,000.

We used a naive softmax over 80,000 words at each output.

We used batches of 128 sequences.

We used stochastic gradient descent without momentum, with a fixed learning rate of 0.7.

After 5 epochs, we begun halving the learning rate every half epoch. We trained our models for a total of 7.5 epochs.

Training Details

[We clipped gradients] by scaling [the gradient] when its norm exceeded 5.

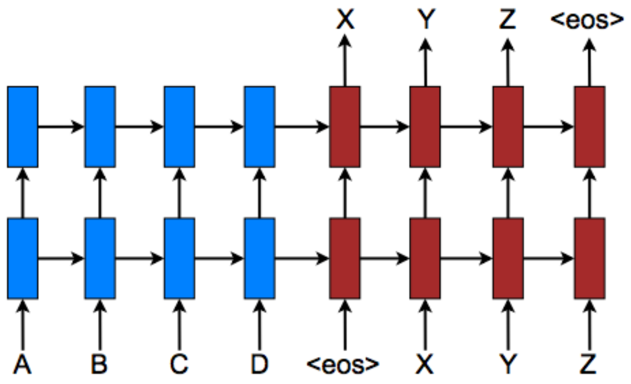
Different sentences have different lengths. ... To address this problem, we made sure that all sentences within a minibatch were roughly of the same length. [This gave] a 2x speedup.

Training Details

We parallelized our [C++] model using an 8-GPU machine.

Each layer of the LSTM was executed on a different GPU and communicated its activations to the next GPU (or layer) as soon as they were computed.

Our models have 4 layers of LSTMs, each of which resides on a separate GPU.



Training Details

The remaining 4 GPUs were used to parallelize the softmax, so each GPU was responsible for multiplying by a 1000 20,000 matrix. [20,000 is 1/4 of the output vocabulary]

Training took about ten days with this implementation [on a training set of 348M French words and 304M English words].

Greedy Decoding vs. Beam Search

We would like

$$\mathbf{y}^* = \operatorname*{argmax}_{\mathbf{y}} Q_\Phi(\mathbf{y}|\mathbf{x})$$

A **greedy algorithm** may do well

$$y^{t+1} = \operatorname*{argmax}_{\hat{y}} \; Q_\Phi(\hat{y} \mid \mathbf{x},\; y^1, \ldots, y^t)$$

But these are not the same.

Example

$$\mathbf{y}^* = \operatorname*{argmax}_{\mathbf{y}} Q_\Phi(\mathbf{y}|\mathbf{x})$$

$$y^{t+1} = \operatorname*{argmax}_{\hat{y}}\ Q_\Phi(\hat{y} \mid \mathbf{x},\ y^1, \ldots, y^t)$$

“Those apples are good” vs. “Apples are good”

$$Q_\Phi(\text{Apples are Good }\texttt{<eos>}) > Q_\Phi(\text{Those apples are good }\texttt{<eos>})$$

$$Q_\Phi(\text{Those}|\varepsilon) > Q_\Phi(\text{Apples}|\varepsilon)$$

Beam Search

At each time step we maintain a list of k word-vector pairs:

$$Y^t = ((y^{t,1}, h^{t,1}), \ldots, (y^{t,k}, h^{t,k})$$

$$Y^{t+1} = \underset{(\hat{y},\hat{h})\in C^t}{\text{kbest}} \; Q_\Phi(y \mid \mathbf{x},\; Y^1, \ldots, Y^t)$$

$$C^t = \left\{ (y^{t+1,i,j}, h^{t+1,i}) : \begin{array}{rcl} y^{t+1,i,j} & \in & \text{kbest}_{\hat{y}}\; Q_\Phi(\hat{y} | (y^{t,i}, h^{t,i})) \\ h^{t+1,i,j} & = & \text{RNNCELL}(h^{t,i}, e(y^{t,i,j})) \end{array} \right\}$$

Attention-Based Translation

Translation is improved by aligning input words with output words.

This is done with “Attention”.

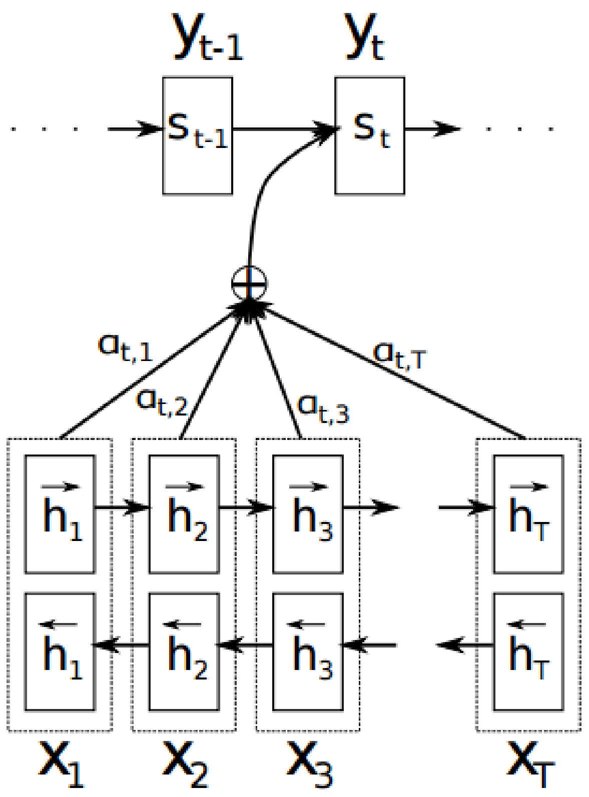
Neural Machine Translation by Jointly Learning to Align and Translate Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio, ICLR 2015 (arXiv Sept. 1, 2014)

Attention

The input sentence is no longer represented by just one thought vector.

Instead the entire sequence of hidden vectors produced by the RNN is used during generation of the translation.

Attention



[Bahdanau, Cho, Bengio (2014)]

A first step: BiRNNs

$$\vec{h}^{t+1} = \mathrm{RNNCELL}_{\Phi}(\vec{h}^{t}, x^{t})$$

$$\overleftarrow{h}^{t-1} = \mathrm{RNNCELL}_{\Psi}(\overleftarrow{h}^{t}, x^{t})$$

$$\overleftrightarrow{h}^{t} = \left[\vec{h}^{t}, \overleftarrow{h}^{t}\right] \qquad ([x, y] \text{ denotes vector concatenation})$$

Basic Sequence to Sequence Model

$$\begin{aligned} s^0 &= \vec{h}^T \\ y^0 &= \text{<eos>} \end{aligned}$$

$$\begin{aligned} Q^\Phi(:\ |\mathbf{x},\ y^1,\ldots,y^i) &= \operatorname*{softmax}_{\hat{y}}\ W^y\ s^{i+1} \\ s^{i+1} &= \mathrm{RNNCELL}_\Phi(s^i,\ e(y^i))) \end{aligned}$$

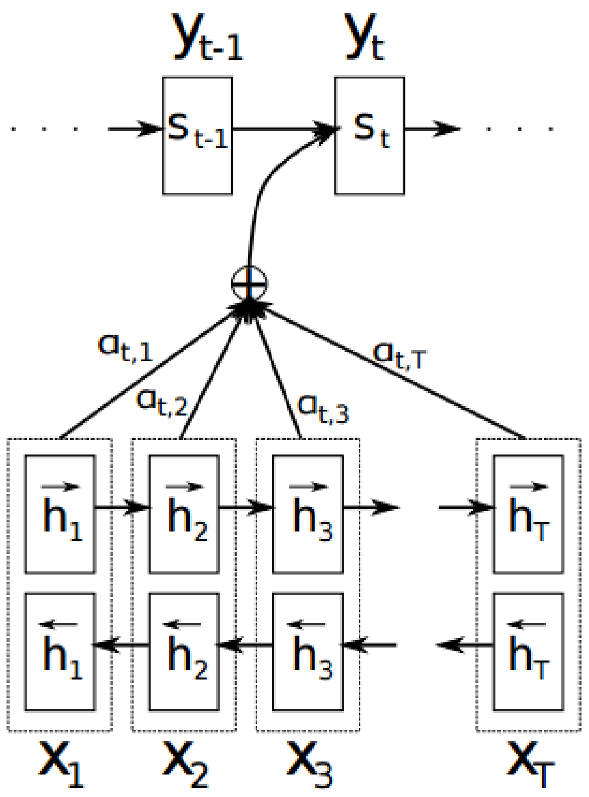
Adding Attention

$$
\begin{aligned}
c^0 &= \overset{\leftrightarrow}{h}^T \\
s^0 &= \vec{h}^T \\
y^0 &= \text{<eos>}
\end{aligned}
$$

$$
\begin{aligned}
Q_\Phi(:\ |\mathbf{x},\ y^1,\ldots,y^i) &= \underset{\hat{y}}{\text{softmax}}\ W^y\ s^{i+1} \\
s^{i+1} &= \text{RNNCELL}_\Phi(s^i,\ [e(y^i),\ c^i])
\end{aligned}
$$

$$
\begin{aligned}
c^i &= \sum_t \alpha^{i,t}\ \overset{\leftrightarrow}{h}^t \\
\alpha^{i,t} &= \underset{t}{\text{softmax}}\ \tanh(W^a\ [s^i,\ \overset{\leftrightarrow}{h}^t])
\end{aligned}
$$

Attention



[Bahdanau, Cho, Bengio (2014)]

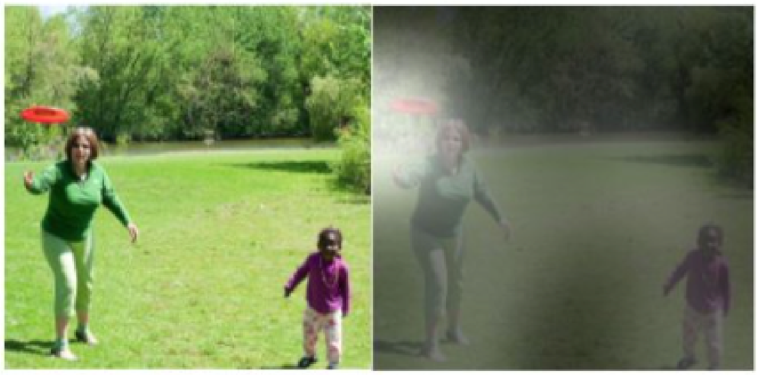
Bilinear Attention is More Popular Today

$$
\begin{aligned}
c^0 &= \overleftrightarrow{h}^T \\
s^0 &= \vec{h}^T \\
y^0 &= \text{<eos>}
\end{aligned}
$$

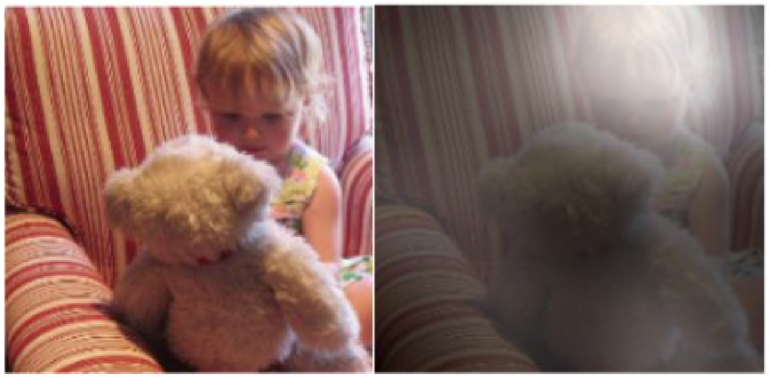
$$
\begin{aligned}
Q_\Phi(: \ |\mathbf{x},\ y^1, \ldots, y^i) &= \operatorname*{softmax}_{\hat{y}} \ W^y \ s^{i+1} \\
s^{i+1} &= \text{RNNCELL}_\Phi(s^i,\ [e(y^i),\ c^i])
\end{aligned}
$$

$$
\begin{aligned}
c^i &= \sum^t \alpha^{i,t} \ \overleftrightarrow{h}^t \\
\alpha^{i,t} &= \operatorname*{softmax}_{t} \ (s^i)^\top W^a \ \overleftrightarrow{h}^t
\end{aligned}
$$

Attention in Image Captioning



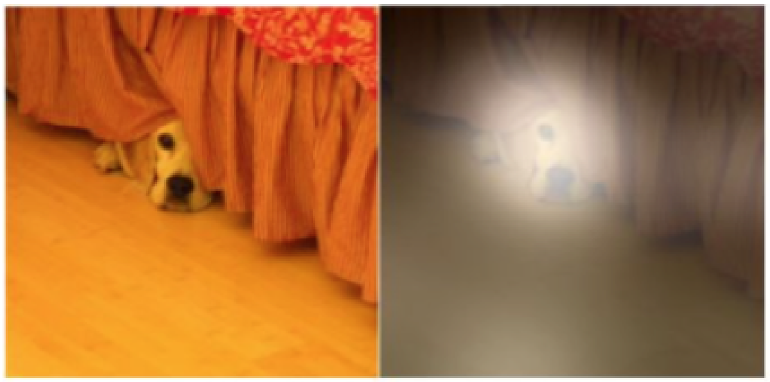
A woman is throwing a frisbee in a park.



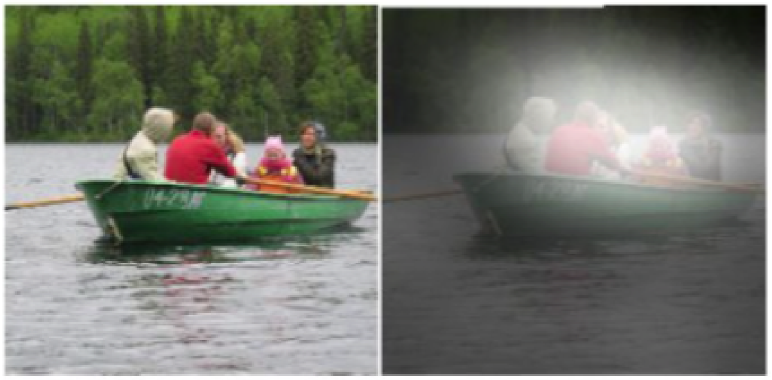
A little girl sitting on a bed with a teddy bear.

Xu et al. ICML 2015

Attention in Image Captioning



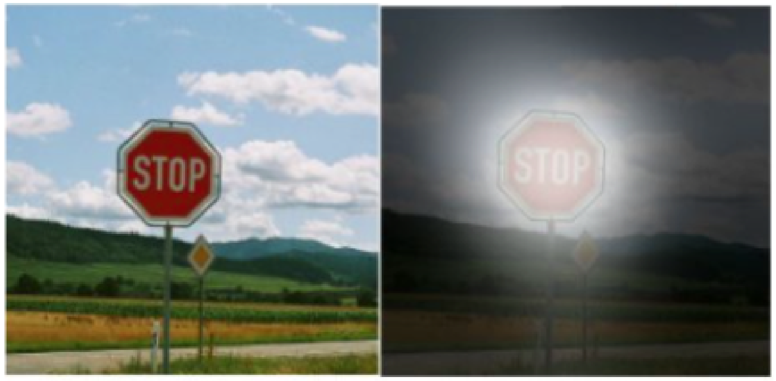
A dog is standing on a hardwood floor.



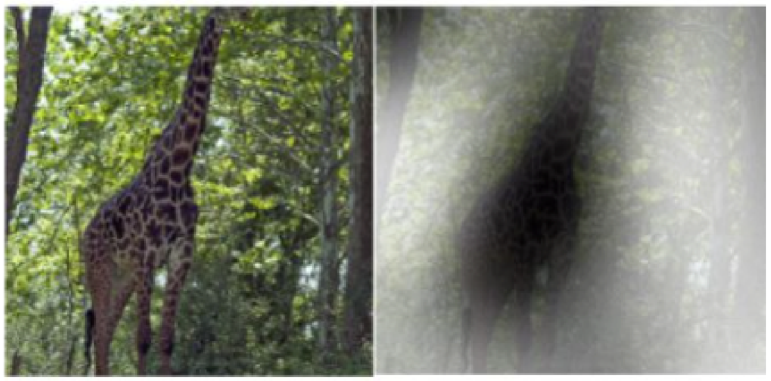
A group of people sitting on a boat in the water.

Xu et al. ICML 2015

Attention in Image Captioning



A stop sign is on a road with a mountain in the background.



A giraffe standing in a forest with trees in the background.

Xu et al. ICML 2015

Introduction to PyTorch

PyTorch Tensors

PyTorch uses **tensors** rather than Numpy's **ndarrays**.

They are essentially the same.

We can map back and forth between these data types.

```
x = np.ones(5)
y = torch.from_numpy(x)

y = torch.ones(5)
x = y.numpy()
```

PyTorch Variables

In PyTorch a node of a computation graph is called a “Variable”. This makes sense:

$$\begin{aligned} y &= f(x) \\ z &= g(y,x) \\ u &= h(z) \\ \ell &= u \end{aligned}$$

$$\begin{aligned} z.\mathrm{grad} &= y.\mathrm{grad} = x.\mathrm{grad} = 0 \\ u.\mathrm{grad} &= 1 \\ z.\mathrm{grad} &\mathrel{+}= u.\mathrm{grad} * \partial h/\partial z \\ x.\mathrm{grad} &\mathrel{+}= z.\mathrm{grad} * \partial g/\partial x \\ x.\mathrm{grad} &\mathrel{+}= y.\mathrm{grad} * \partial f/\partial x \end{aligned}$$

PyTorch Variables

In EDF `x.value` is the value of an EDF node.

`x.value` is a NumPy **ndarray**.

In PyTorch `x.data` is the value of a PyTorch variable.

`x.data` is a PyTorch **tensor**.

Tensor Operations on PyTorch Variables Support Backprop

```
import torch
from torch.autograd import Variable

x = Variable(torch.ones(2,2), requires_grad = True)
y = x + 2
z = 3*y*y
loss = z.mean()

loss.backward()
print x.grad

    4.5 4.5
    4.5 4.5
```

Static vs. Dynamic Computation Graphs

EDF uses static computational graphs.

- The same graph is used many times by setting inputs and doing a forward computation.
- Before the backward computation all gradient values need to be initialized to zero.

PyTorch uses dynamic computation graphs.

- A computation graph is only used once.
- There is only one backward pass ever computed on a graph.
- Gradient values are initialized to zero at graph creation time.

Tensor Operations on PyTorch Variables Support Backprop

```
x = Variable(torch.randn(3), requires_grad = True)
y = 2*x
while y.data.norm() < 1000:
  y = 2*y

loss = y.mean()
loss.backward()
print(x.grad)
```

PyTorch Functions

PyTorch functions (function in torch.nn.functional) are functions that map variables to variables.

```
import torch
from torch.autograd import Variable
import torch.nn.functional as F

x = Variable(torch.randn(5,5), requires_grad = true)
y = F.relu(F.sigmoid(x))
loss = y.mean()
loss.backward()
```

Modules: Functions with Internal Parameters

In EDF parameter packages are passed as arguments.

In PyTorch parameters are not passed as arguments.

Instead parameters are internal to functions.

A function with internal parameters is called a **module**.

The parameters in a module are initialized when the module is initialized.

Parameters are then updates by backprop as modules are used in (various dynamic) computation graphs.

Example of a Module

```
import torch.nn.functional as F
import torrch.nn as nn

class relu_norm_conv(nn.Module):

    def __init__(self, in_channels, out_channels):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels,out_channels)
        self.bn1 = nn.BatchNorm2d(out_channels)

    def forward(self, x):
        return F.relu(self.bn1(self.conv1(x)))
```

Tensor Operations on Variables in Forward Methods

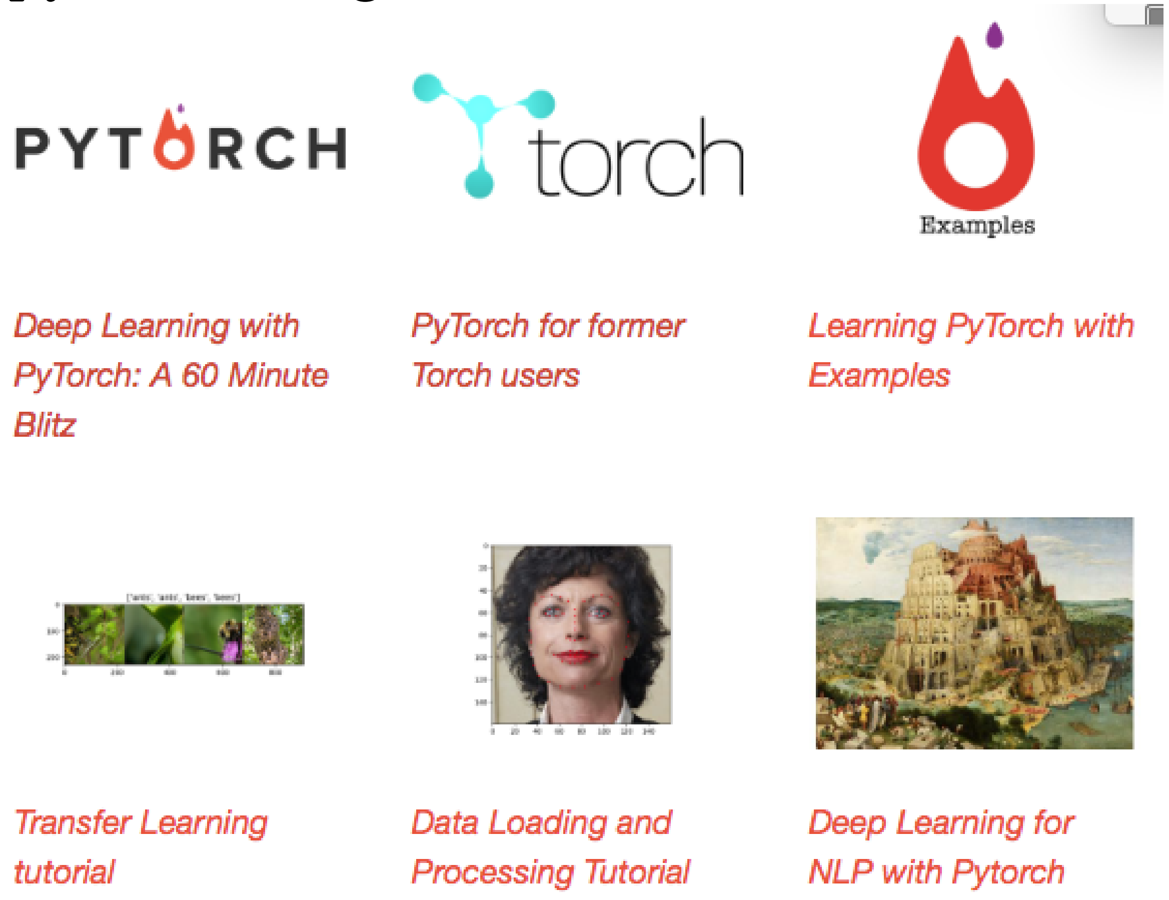
```
class BasicBlock(nn.Module):

    def __init__(...)
        ...

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = F.relu(self.bn2(self.conv2(out)))
        out = out + self.skip(x)
        return out
```

PyTorch Tutorials, Documentation and Repositories

http://pytorch.org/tutorials/



Deep Learning with PyTorch: A 60 Minute Blitz

PyTorch for former Torch users

Learning PyTorch with Examples

Transfer Learning tutorial

Data Loading and Processing Tutorial

Deep Learning for NLP with Pytorch

Phrase Based Statistical Machine Translation (SMT)

Step I: Learn a phrase table — a set of triples (p, q, s) where

- p is a (short) sequence of source words.
- q is a (short) sequence of target words.
- s is a score.

(“au”, “to the”, .5) (“au banque”, “the the bank”, .01)

For a phrase P we will write P.source for the source phrase, P.target for the target phrase, and P.score for the score.

Derivations

Consider an input sentence x of length T.

We will write $x[s:t]$ for the substring $x[s], \ldots, x[t-1]$.

A derivation d from x is a sequence $(P_1, s_1, t_1,), \ldots, (P_K, s_K, t_K)$ where P_k.source $= x[s_k : t_k]$.

The substrings $x[s_k : t_k]$ should be disjoint and "cover" x.

For $d = [(P_1, s_1, t_1,), \ldots, (P_L, s_K, t_K)]$ we define

$$y(d) \equiv P_1.\text{target} \;\cdots P_K.\text{target}$$

We let $D(x)$ be the set of derivations from x.

Scoring

For $d \in D(x)$ we define a score $s(d)$

$$s(d) = \alpha \ln P_{\text{LM}}(y(d)) + \beta \sum_k P_k.\text{score} + \gamma \text{ distortion}(d)$$

where $P_{\text{LM}}(y)$ is the probability assigned to string y under a language model for the target language

and distortion(d) is a measure of consistency of word ordering between source and target strings as defined by the indeces (s_1, t_1), ..., (s_K, t_K).

Translation

$$y(x) = y(d^*(x))$$

$$d^*(x) = \operatorname*{argmax}_{d \in D(x)} s(d)$$

END