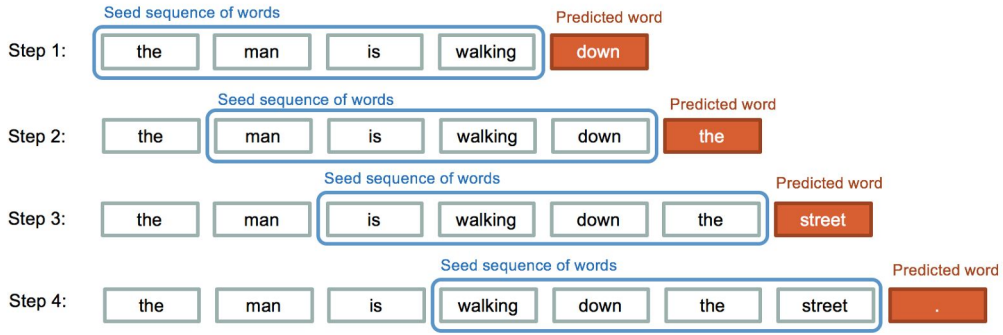
Recurrent Neural Network

Outline

- Why RNNs?
- What are RNNs are used for?
- Vanilla RNNs
  - Difference in structure
  - Unrolling
  - Loss function discussion
- Long short term memory networks
- Gated recurrent units
- Applications
- Bidirectional RNNs
- Stacked RNNs
- Best practices

Intro to recurrent neural networks (RNN)

- We’re going to take a look at a new architectural structure
- The recurrent unit uses past information to predict sequence/ future outputs
- RNNs are used mainly with text
- They are well suited to the task of dealing with grammar and syntax structures
- They also allow variable size inputs/outputs and alignment
- When coupled with beam search they can produce several possible phrase outputs



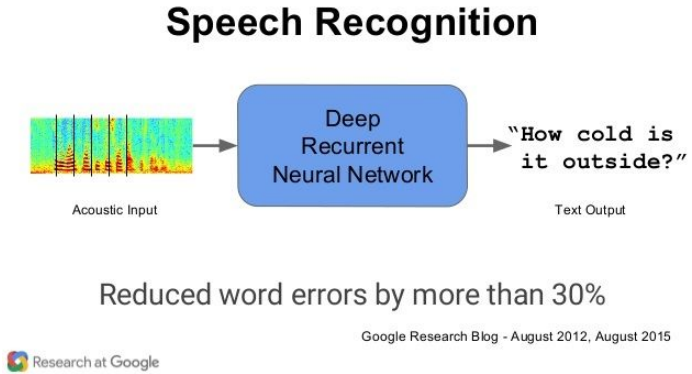
Phrase generation with LSTMs through sequential prediction.
Image From: *Text Generation using Bidirectional LSTM and Doc2Vec models*, David Campion (2018)

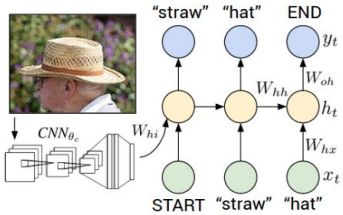
Where are RNNs used?

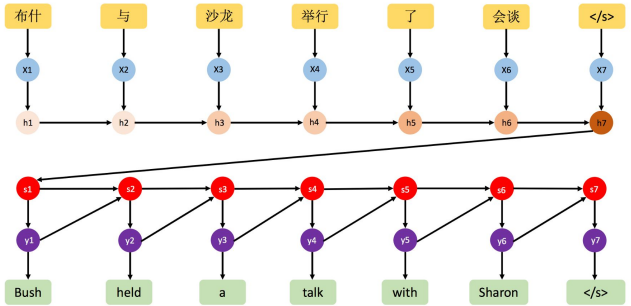
- It is used for sequences analysis such as:
  - Text
  - Translation
  - Image captioning
  - Time series
  - Vocal Analysis

Used as blocks for:

- Translation - neural language models
- Chatbots
- Question/answer

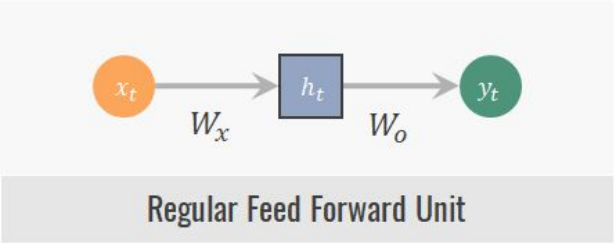




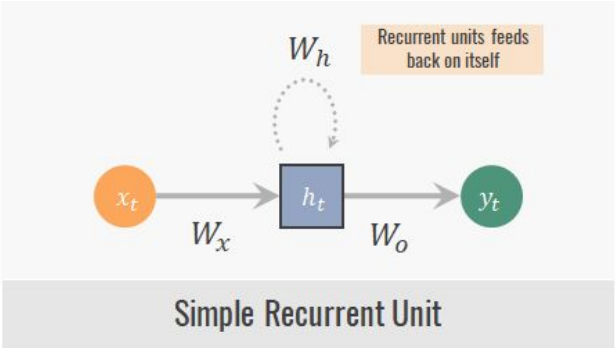


(Sutskever et al., 2014)

Simple Recurrent Unit - Elman Unit



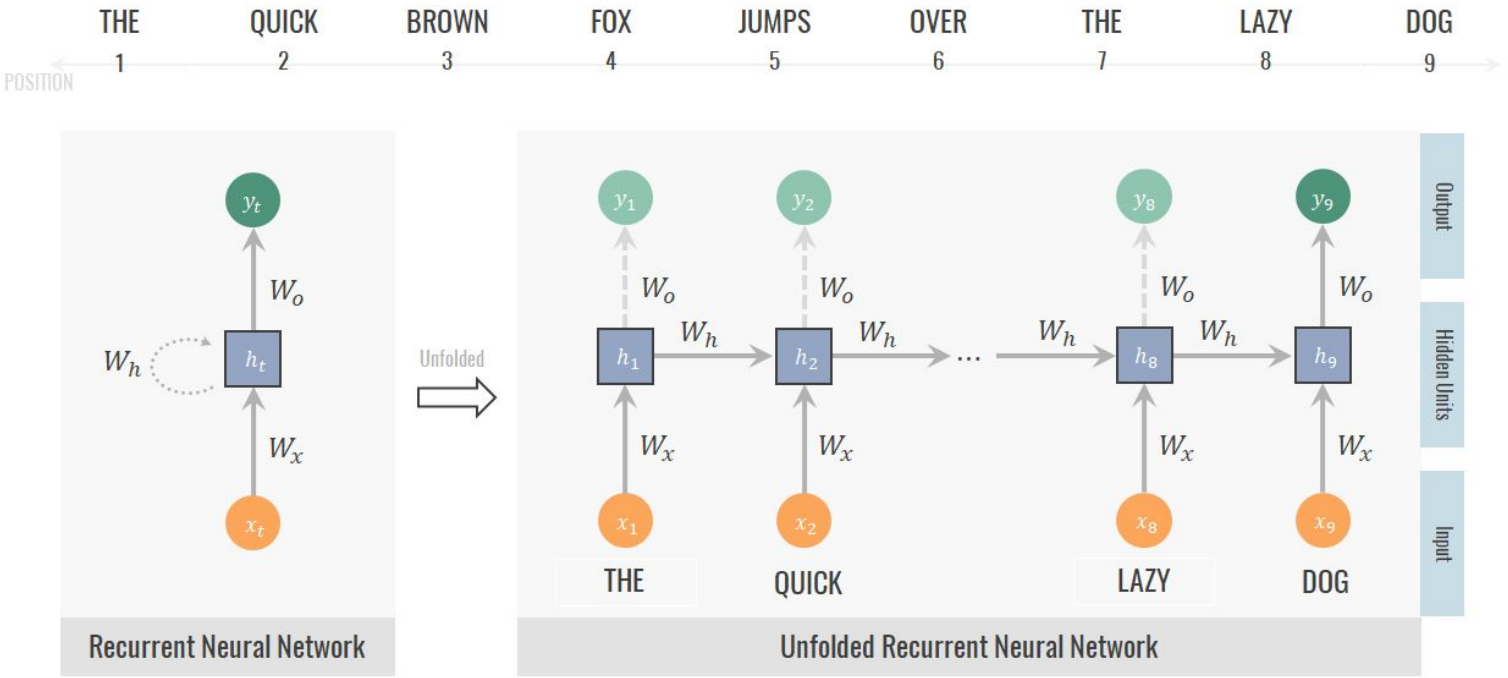
Standard unit



Elman unit

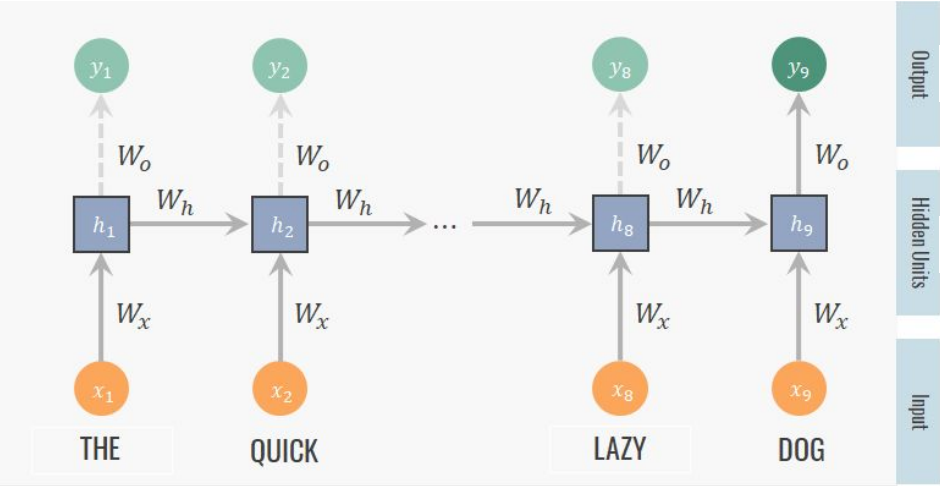
In a standard 1-layer hidden network there are two weight matrices W_x and W_o.
In a recurrent net an additional weight matrix W_h is introduced to retain past information

Recurrent units are actually stacked



Unfolding a recurrent unit highlights the relation among W_x, W_o and W_h matrices

How is information propagated in RNNs?



$$y_t = softmax(W_o^T h_t + b_o)$$

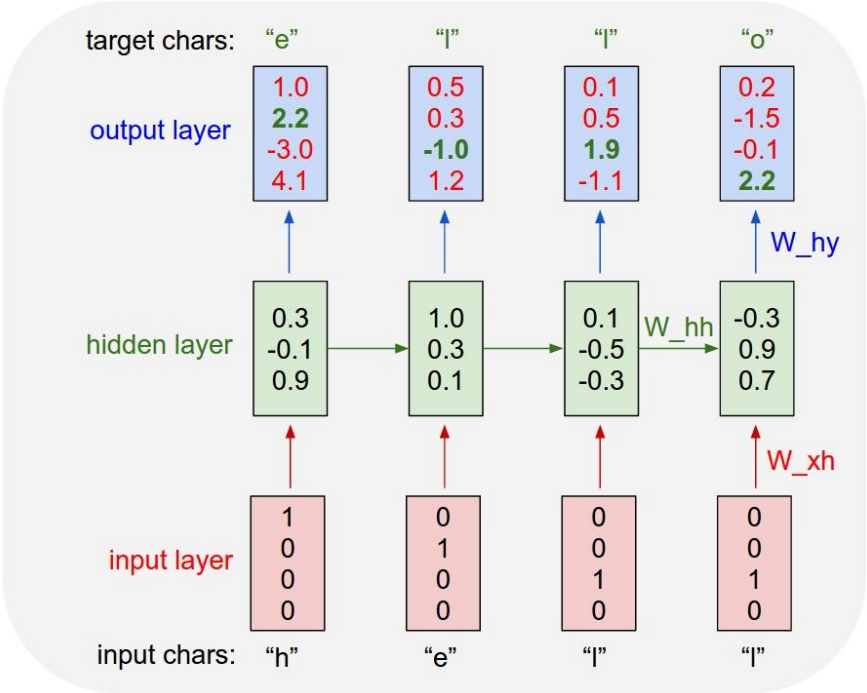
$$h_t = f(a_t) \Leftrightarrow f(W_h h_{t-1} + W_x^T x_t + b_h)$$

$$a_t = W_h h_{t-1} + W_x^T x_t + b_h$$

- h_0 is usually initialized to 0
- In previous situations (i.e. classification) our output was one class, RNNs allow us to output a value at every step

- Weights are shared in a recurrent net
- RNNs are trained using back propagation through time (a type of gradient descent)

Example of sequential prediction



An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons)

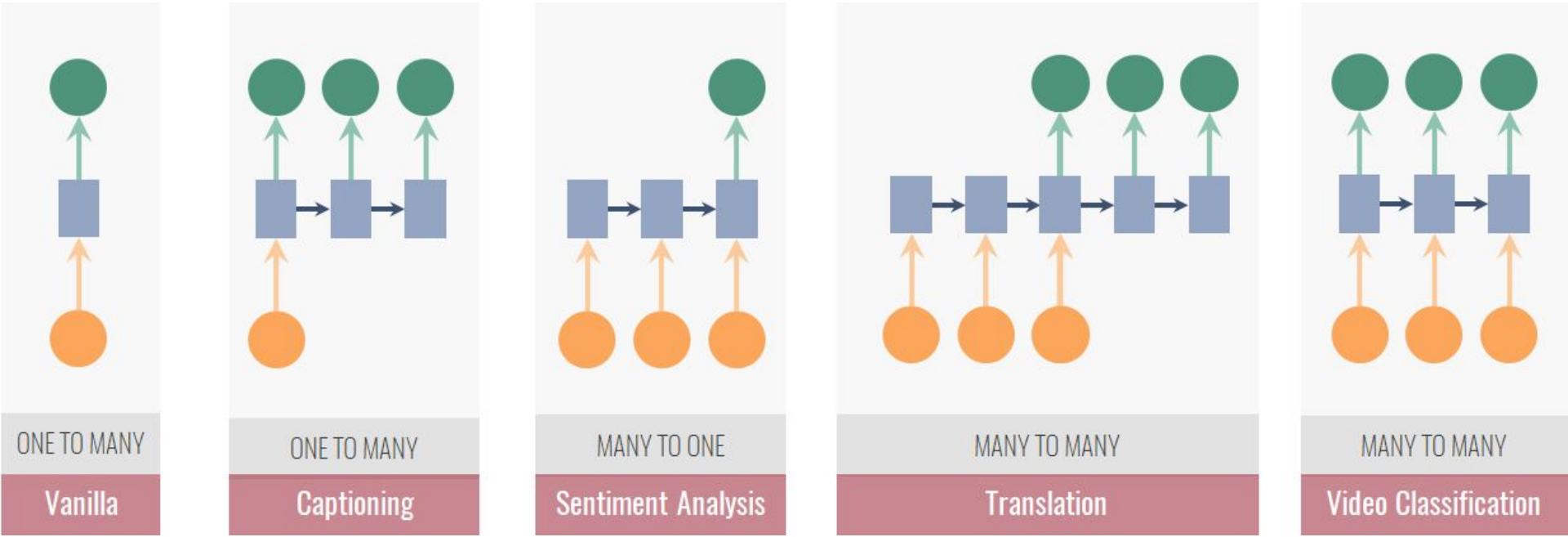
This diagram shows the activations in the forward pass when the RNN is fed the characters "hell" as input

The output layer contains confidences the RNN assigns for the next character (vocabulary is "h,e,l,o")

We want the green numbers to be high and red numbers to be low.*

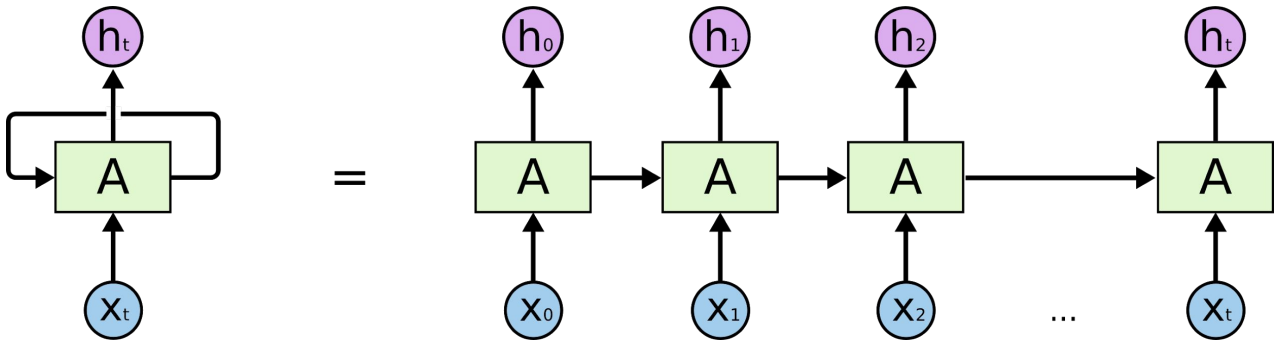
*Image and example from: The Unreasonable Effectiveness of Recurrent Neural Networks, Karpathy 2015

Different sequence relations for different use cases



The network architecture can have one or many inputs/outputs.
Regardless of the networks structure, sequence information (recurrence) is passed along the hidden layer.

An unrolled recurrent neural network



RNNs propagate earlier information to the present

Appeal of RNNs is the idea that they can connect previous information to the present tasks, examples include:

- Earlier video frames
- Word context
- Time series data

The idea behind RNNs is that they should be able to understand context given previous words: example

“Jim entered the room through the *door*”

RNNs don’t empirically work well for long sequences

The trouble with RNNs is that longer phrases or more sophisticated contexts require more information to predict the next word. For example:

“Jim entered the room through the *door*. Rose joined shortly after to discuss the news with Jim. Both Jim and Rose and in the _____? ”

Empirical results suggest that while simple RNNs function reasonably well for short sentences they perform poorly for longer-term dependencies more complex sentences

Among the reasons for their performance degradation include the dissipation of the gradient across neurons

RNNs model the entire sequence probability

RNNs model: p(x1)p(x2|x1)p(x3|x2,x1)...= p(x1...xn)

Intuitively, RNNs appear to have local Markov structure

Their global expansion is however *non-linear as the multiplication of activation functions with weight matrices* is accumulated:

$$
\begin{aligned}
h_t &= f(W_h h_{t-1} + W_x x_t + b_h) \\
&= f(W_h\, f(W_h h_{t-2} + W_x x_{t-1} + b_h) + W_x x_t + b_h) \\
&= f(W_h\, f(\cdots W_h f(W_h h_o + W_x x_1 + b_h)\cdots) + W_x x_t + b_h)
\end{aligned}
$$

That goal of RNNs is to retain contextual information from very far away (the past) and inform the present context

- W_h stores important information whereas the non-linear nature of RNNs ensures information is passed from past to present

Training RNNs is challenging

Training RNNs however is difficult because of the “Vanishing/ Exploding” gradient problem (Hochreiter 1991)*

In the calculation the W matrix’s gradient the product of the previous hidden layer gradients are multiplied with respect to one another

$$\frac{\partial J_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial J_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \underbrace{\frac{\partial h_t}{\partial h_k}}_{\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}} \frac{\partial h_k}{\partial W}$$

For the computer, if the gradient of these time steps

- Is very small their product will -> 0
- Is very large their product will -> ∞

Solutions**

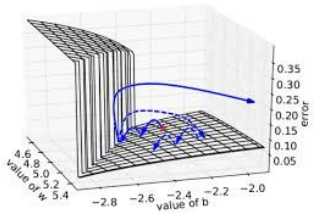
Norm clipping for exploding gradients

Proper initialization of the W matrix

Regularization

Use ReLU instead of tanh or sigmoid activation functions

- -ReLU derivative is a constant
- - So it isn’t as likely to suffer from vanishing gradients



* “Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients” Britz (2015)

**”On the difficulty of training recurrent neural networks” Pascanu, Mikolov, Bengio 2013

Training RNNs is challenging

Training RNNs however is difficult because of the “Vanishing/ Exploding” gradient problem (Hochreiter 1991)*

In the calculation the gradient the product of the previous hidden layer gradients are multiplied with respect to one another

$$\frac{\partial J_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial J_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \underbrace{\frac{\partial h_t}{\partial h_k}}_{\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}} \frac{\partial h_k}{\partial W}$$

For the computer, if the gradient for these time steps

- Is very small their product will -> 0
- Is very large their product will -> ∞

Solutions**

Norm clipping for the expling gradient

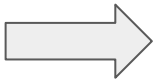
Proper initialization of the W matrix

Regularization

Use ReLU instead of tanh or sigmoid activation functions

-ReLU derivative is a constant

- So it isn’t as likely to suffer from vanishing gradients



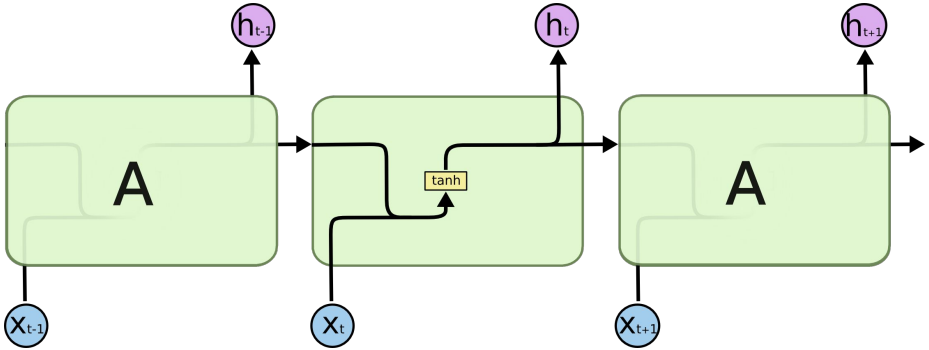
OR use LSTMs/ GRUs

* “Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients” Britz (2015)

**”On the difficulty of training recurrent neural networks” Pascanu, Mikolov, Bengio 2013

Long Short Term Memory Networks

- Introduced by Hochreiter and Schmidhuber in 1997
- Designed to remember long-term dependencies
- Similar to RNNs in that they have a repeatable modular chain structure
- Differences in internal structure however and information transmission is what makes the LSTMs different to standard RNNs



Cell structure of RNN with tanh activation

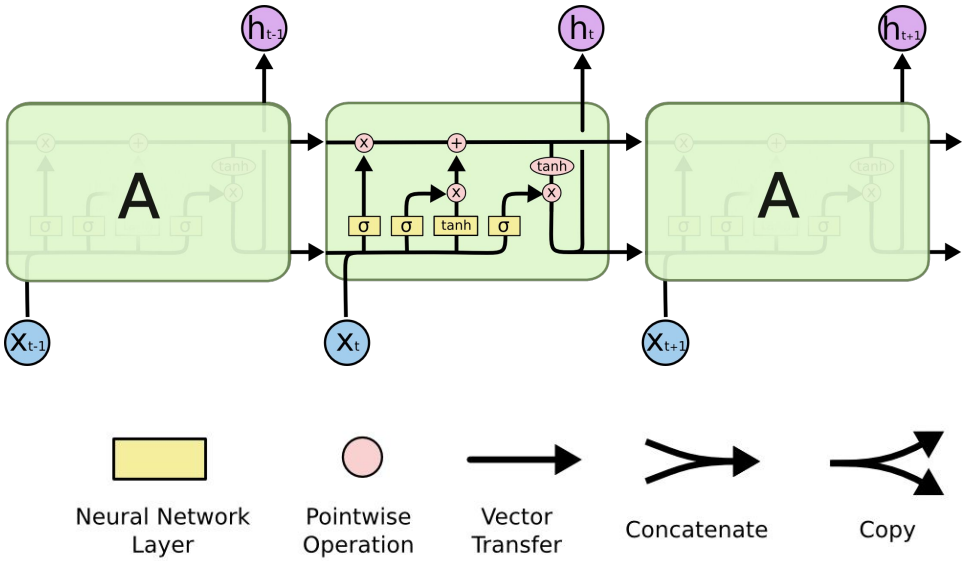
Pictures from Chris Olah's Understanding LSTM Networks

High-level view of LSTM cells

We'll go over the internals of an long short term memory (LSTM) network

Instead of dealing with neurons we'll now discuss "cells" which are a composition of several operations

The LSTM may look complicated but each operation has



Cell structure of LSTM

Gated units control information flow through the cell

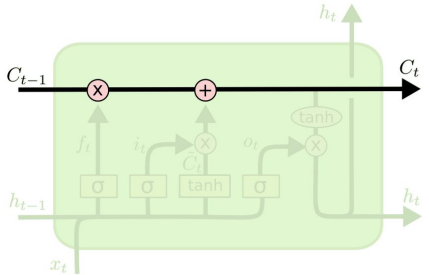
C_t is the cell state

It is a conveyor belt that transports information through the cell from C_{t-1} to C_t

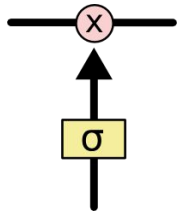
Cells are updated by gates which allow information to be
- Added or removed (additive)
- Amplified or attenuated (multiplicative)

Cell gates let a proportion of information through
- Their effect is a value between 0-1

An LSTM cell has 3 gates



Cell structure of LSTM



Multiplicative sigmoid gate

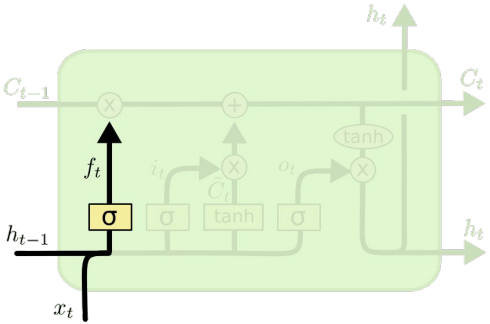
Step 1: What do we keep from the cell state?

The first gate in the cell is the “forget gate”

The forget gate is responsible for deciding the relevance of the previous states information

It allows some proportion (between 0-1) of the past cells to propagate to the next cell

In this way states that are more distant are down weighted whereas those that are contextually more relevant in the present are retained



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

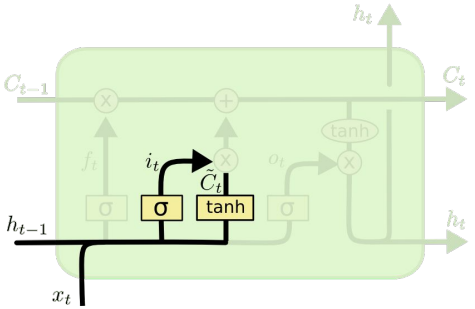
Step 2: What new information is shared?

The second gate decides what new information will be stored in the cell state

First a sigmoid gate called the “input gate layer” decides on the which values to update

The next (tanh) layer creates a candidate vector of new values

The two values are combined to create an update state



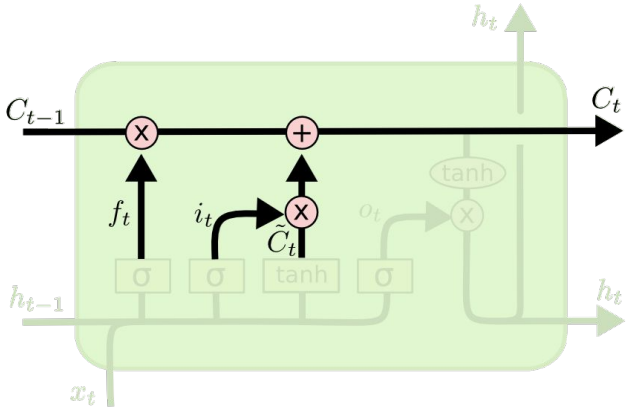
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Step 3: Combining information through

The forget and input gates are next combined to generate the new state of C_t

The new cell state C_t therefore propagates new information forward while forgetting older less relevant information

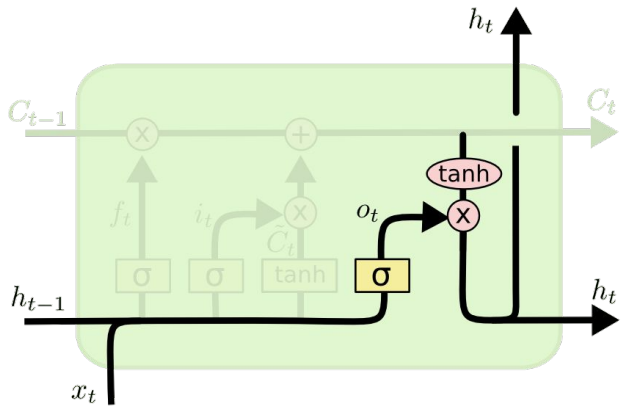


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Step 4: Value resolution and information passing

Once the cell state is updated 3 things occur:

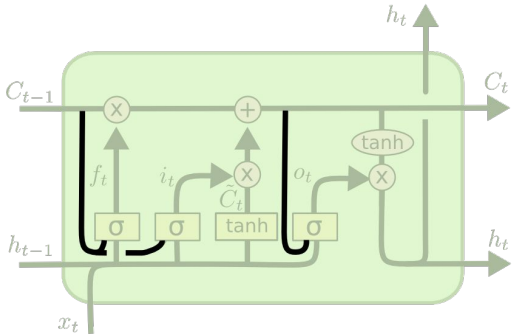
1. C_t is multiplied by a sigmoid “output gate” (sigmoid) which decides on the relevant information to output
2. The outpute gate is then scaled by a tanh function
3. Finally the new hidden state is output and also passed along to C_{t+1}



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

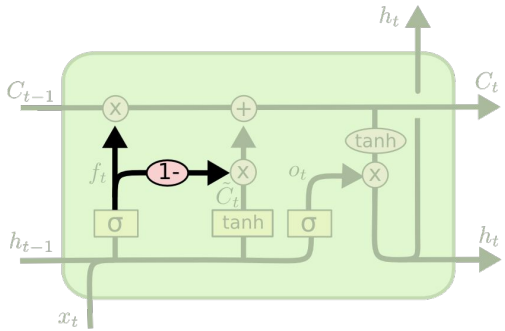
$$h_t = o_t * \tanh\left(C_t\right)$$

Variants of LSTM



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

The peephole LSTM modulates all gates through peep hole connections



$$C_t = f_t * C_{t-1} + (\mathbf{1} - \boldsymbol{f_t}) * \tilde{C}_t$$

The coupled LSTM ensures that if we are going to forget something that it is then replaced by something else

Sometimes also called a rated network

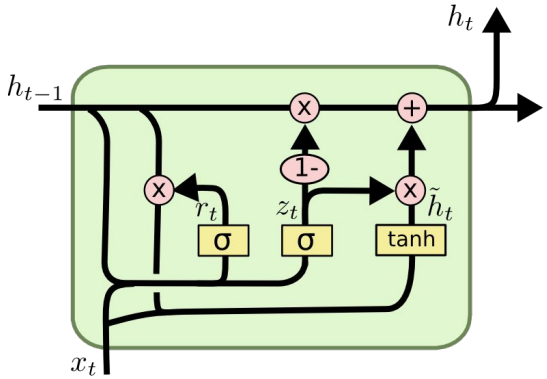
Gated Recurrent Unit

In 2014 Cho et. al introduced the Gated Recurrent Unit or GRU*

GRUs have become popular alternatives to LSTMs because the combine the forget and input gates into an “update gate”

GRUs also have a “reset gate” which update the amount of new information included the candidate proposal

The resulting cell is simpler and has comparable accuracy to LSTMs



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

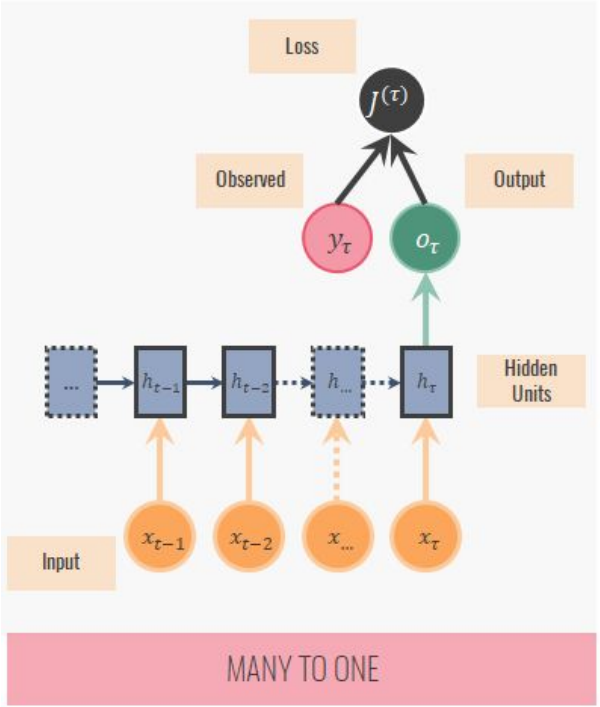
*Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation, Cho et al 2014

Pragmatics: Training a many to one network

This is a typical supervised learning problem, i.e. sentiment classification

The loss function has only one output per sequence

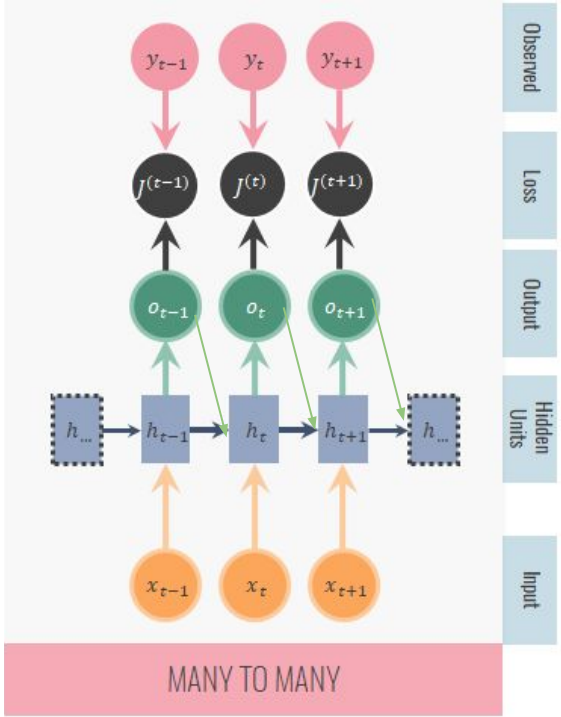
In the case of training a text model inputs are first embedded in some D-dimensional space



Pragmatics: Many to many

A typical architecture like this is used in language generation where the goal is to predict the next word given a series of previous words

As a practice:
- Words are embedded
  - This means both input and outputs require dictionaries to reverse out the embeddings
- The network is trained using “Teacher forcing”
  - Teacher forcing is a fast and effective way to train a recurrent neural network that uses output from prior time steps as input to the model
  - Teacher forcing is when the output sequences are used as the next words in the phrase rather than predicted outputs



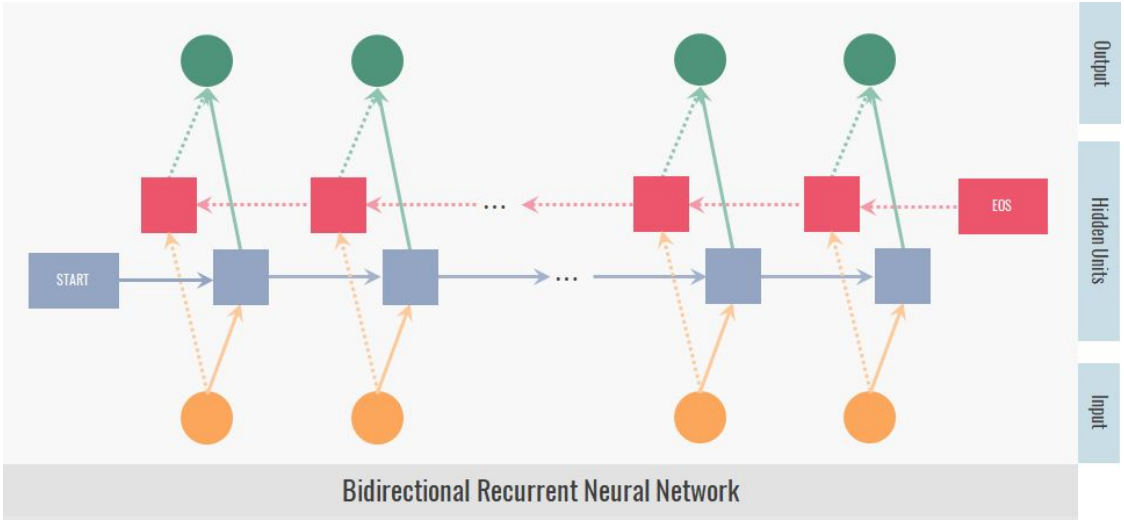
Sequences contain structure when parsed in reverse

A bidirectional network trains a model on a forward and reversed sequence

The idea is that by training phrase on reversed format that details missed by the forward pass are picked up by reversing the sequence

For example the opening and closing of a parentheses

Accordingly reading a phrase in reverse also imparts knowledge

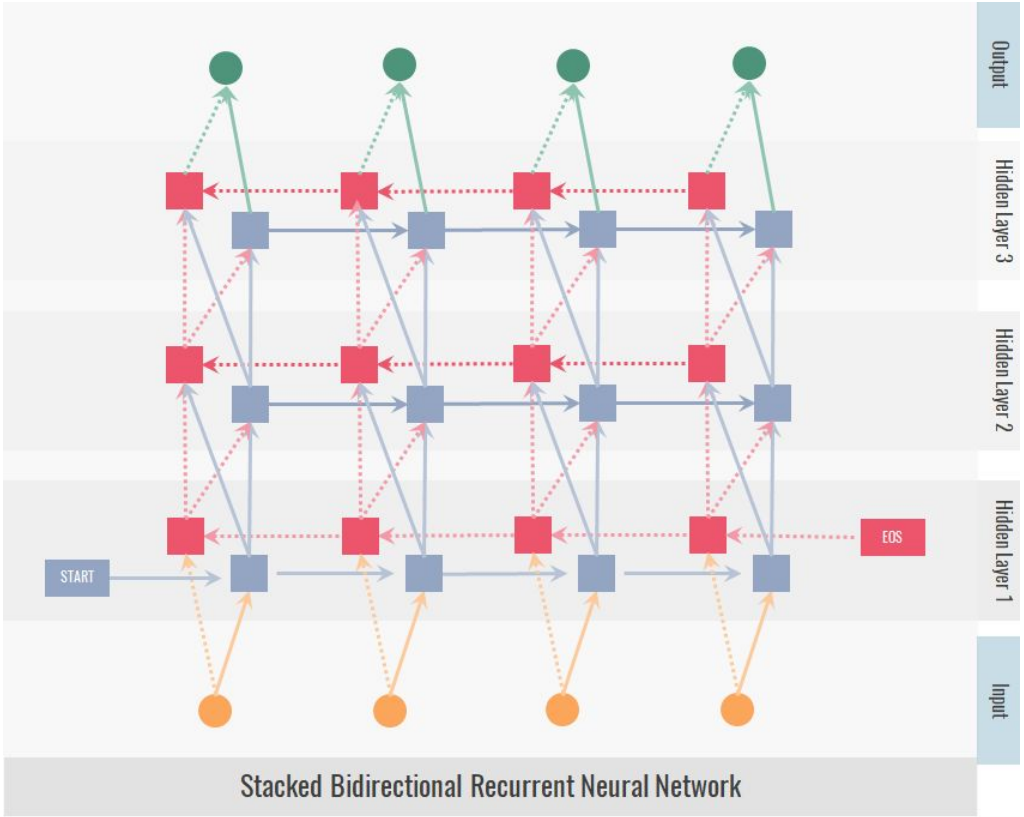


Stacked networks increase representational capacity

Stacking networks whether mono or bidirectional increases their representational capacity

There is a however a limit to the number of stacked layers as the computational burden and number of trainable parameters increase rapidly

Given the richness of problems that RNNs address there are a multitude of ways to build networks



Sample Code Snippets

```
# --- Training model: Embedding + RNN ---
model = Sequential()
model.add(Embedding(max_features, 32))
#model.add(SimpleRNN(32))
#model.add(GRU(32))
model.add(LSTM(32))
model.add(Dense(1, activation ='sigmoid'))
```

Embedding features in input

API is straight forward for RNN, LSTM and GRUs

Output will be in embedded space

```
# --- Rudnning the model ----
# - Here we use the functional API -
inp = Input(shape=(MAX_SEQUENCE_LENGTH,))
x = embedded_sequences = embedding_layer(inp)
x = Bidirectional(SimpleRNN(100,
                            return_sequences=False,
                            dropout=0.1,
                            recurrent_dropout=0.1))(x)
x = Dense(3, activation="softmax")(x)
```

Embedding features in input

Bidirectional SimpleRNN with dropout

Output will be in embedded space