Introduction to Deep Learning (IT3320E)

8 - RNN: Recurrent Neural Network

Hung Son Nguyen

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Nov. 8, 2022

Agenda

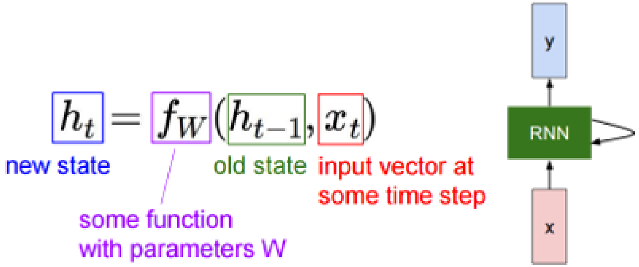
Section 1

Introduction

Recurrent Neural Networks

In **feedforward neural network** computation flows directly from input $\mathbf{x}$ through intermediate layers $\mathbf{h}$ to output $\mathbf{y}$.

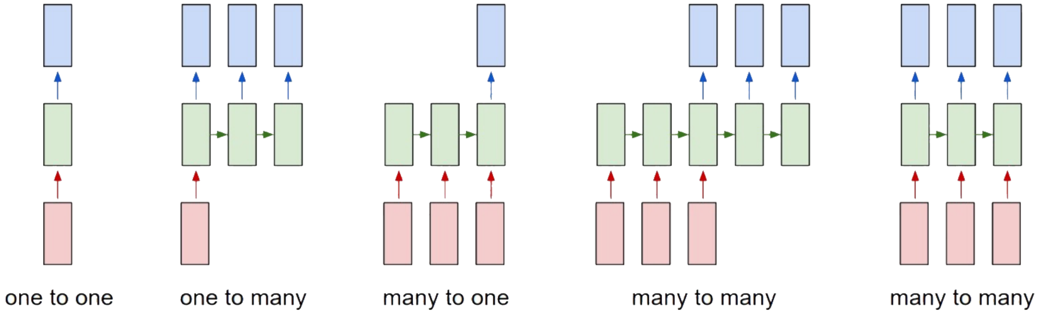
A **recurrent neural networks** (RNN) is a class of artificial neural network where connections between units form a directed cycle. This creates an internal state of the network which allows it to exhibit dynamic temporal behavior. Unlike feedforward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs. This makes them applicable to tasks such as unsegmented connected handwriting recognition or speech recognition.



Recurrent Neural Networks

Recurrent neural networks (RNN) are **specialized for processing sequences**. Similarly, we saw that convolutional neural networks feature specialized architecture for processing images.

RNNs boast a **much wider API with respect to feedforward neural networks**. Indeed, these models can deal with *sequences* in the input, in the output or even both.

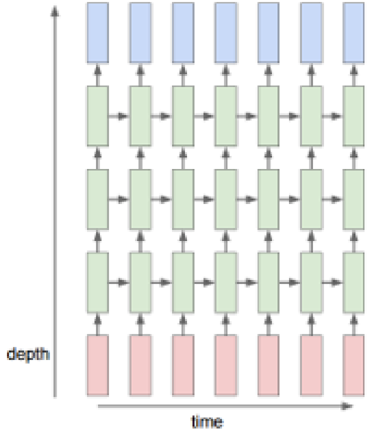


Example of RNN

Let's see some wire examples

1. Vanilla Neural Networks
2. Image Captioning (image to sequence of words)
3. Sentiment Classification (sequence of words to sentiment)
4. Machine Translation (seq of words to seq of words)
5. Video classification on frame level

We can stack RNNs together to produce deeper RNN. In figure 5 case we have stack 3 RNNs. It still works the same as before but now we have 3 weights.



Practical Example

Lets create a RNN that given a sequence of characters predicts the next one. The output is always the prob of each letter in the vocabulary to be the next character:

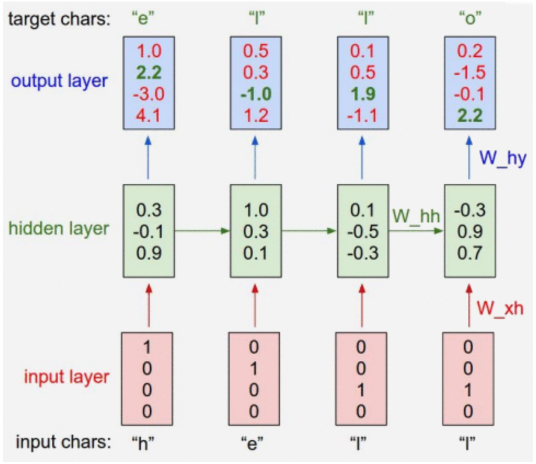
- Vocabulary: [h,l,e,o]
- Example training sequence: "hello"

For this example we will use a Vanilla RNN:

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

$$y_t = W_{hy}h_t \qquad (2)$$

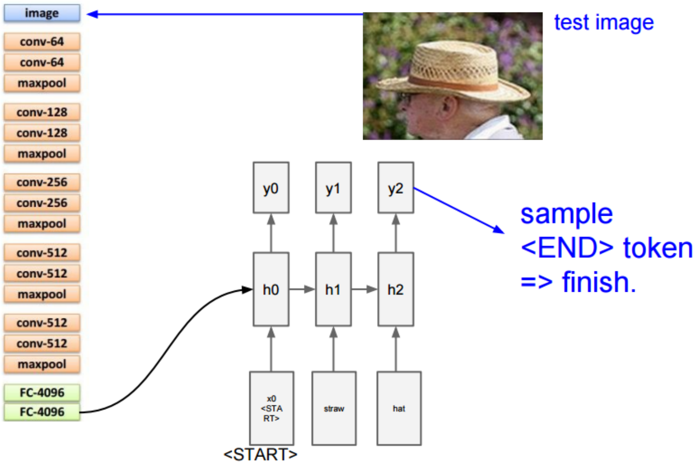
$$h_0 = 0 \qquad (3)$$



You can see this blue boxes being softmax classifier, in other words, at each time step there is a softmax classifier

Image Captioning

Example using RNN for image captioning

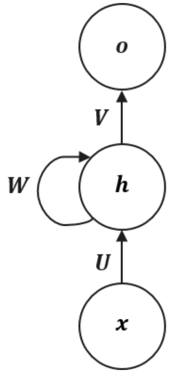


"Deep Visual-Semantic Alignments for Generating Image Descriptions", Karpathy and Fei-Fei

Section 2

Vanilla RNN

Vanilla RNN



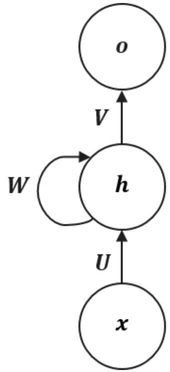
The vanilla RNN is provided with three sets of parameters:

- $\mathbf{U}$ maps inputs to the hidden state
- $\mathbf{W}$ parametrizes hidden state transition
- $\mathbf{V}$ maps hidden state to output

System dynamics is as simple as:

$$\begin{cases} \mathbf{h}^{(t)} = \phi(\mathbf{W}\,\mathbf{h}^{(t-1)} + \mathbf{U}\,\mathbf{x}^{(t)}) \\ \mathbf{o}^{(t)} = \mathbf{V}\,\mathbf{h}^{(t)} \\ \mathbf{y}^{(t)} = \mathbf{softmax}(\mathbf{o}^{(t)}) \end{cases} \tag{4}$$

Intuition about Hidden State



The hidden state $\mathbf{h}^{(t)}$ can be intuitively viewed as a *lossy* summary of the sequence of past inputs fed to the network, in which are stored the main task-relevant aspects of the past sequence of inputs up to time t.

Since the an input sequence of arbitrary length $(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(t)})$ is mapped into a fixed size vector $\mathbf{h}^{(t)}$, this summary is necessarily lossy.

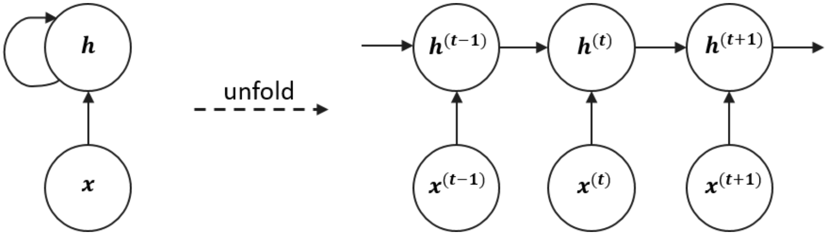
Section 3

Training a RNN

Unfolding the Computational Graph

A recurrent computational graph can be unfolded into a sequential computational graph with a repetitive structure.

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{t-1}, \mathbf{x}^{(t)}; \theta)$$



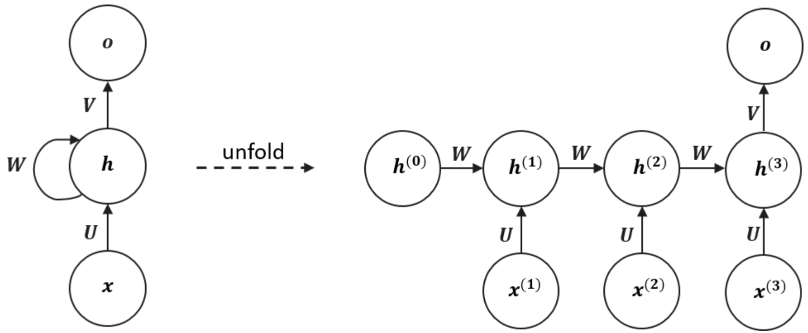
Backpropagation Through Time

The capacity to **unfold a recurrent graph into a DAG** (Directed Acyclic Graph) allows to train recurrent neural network by means of standard backpropagation.

Because the gradient conceptually flows backward though time instead of though layers, this algorithm is usually referred to as **backpropagation through time (BPTT)**.

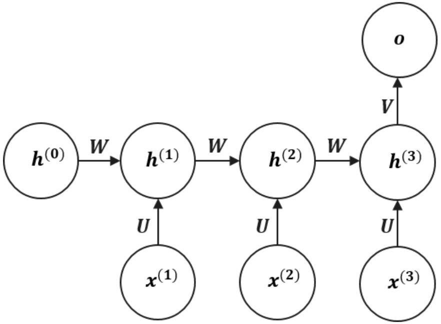
Backpropagation Through Time

Let's make an example for an simple architecture which processes sequences of length $\tau = 3$ and produces an output at the end of the sequence.



Backpropagation Through Time

Now, given a differentiable loss on final output $L(\mathbf{y}, \mathbf{o})$ let's compute the derivative of the objective L with respect to the weights $\mathbf{V}$, $\mathbf{W}$ and $\mathbf{U}$ and see what happens.



- $\frac{\partial L}{\partial \mathbf{V}} = \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{V}}$
- $\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{h}^{(3)}} \sum_{k=0}^{3} \left(\frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{h}^{(k)}} \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}} \right)$
- $\frac{\partial L}{\partial \mathbf{U}} = \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{h}^{(3)}} \sum_{k=0}^{3} \left(\frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{h}^{(k)}} \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{U}} \right)$

Consideration: while $\frac{\partial L}{\partial \mathbf{V}}$ depends only on current state, $\frac{\partial L}{\partial \mathbf{W}}$ and $\frac{\partial L}{\partial \mathbf{U}}$ depend on all previous sequence states.

The Challenge of Long-Term Dependencies

Depending on the network parameters and choice of activation functions, the opposite problem can arise. This is called **exploding gradient problem** and happens when gradients become bigger and bigger until numerical problems destroy the optimization process.

Both issues can be mitigated through proper weight initialization, accurate choice of activation functions and gradient clipping.

Note: these problems can also happen in deep feedforward networks: however, they are more common in recurrent architectures because these models are usually very deep (actually as deep as the length of the input sequence).

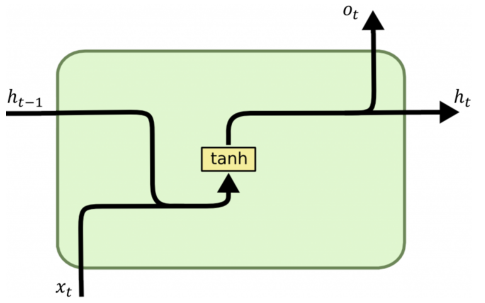
The Challenge of Long-Term Dependencies

Looking closer, we see that terms $\frac{\partial \mathbf{h}^{(\mathbf{t})}}{\partial \mathbf{h}^{(k)}}$ must be themselves computed through the chain rule. For example, we can obtain $\frac{\partial \mathbf{h}^{(\mathbf{3})}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(\mathbf{3})}}{\partial \mathbf{h}^{(2)}} \frac{\partial \mathbf{h}^{(\mathbf{2})}}{\partial \mathbf{h}^{(1)}}$.

Turns out [3] that when using *tanh* and *sigmoid* activations, the 2-norm of these Jacobian matrices is upper bounded by 1 and $1/4$ respectively. Thus, we can easily end up multiplying smaller and smaller numbers until the gradients become zero.

This problem is known as **vanishing gradient problem**, and causes serious trouble when trying to learn long-term dependencies in the input sequences, because contributions of "far-away" steps become zero.

Calculation with vanilla RNN:



Notations:

x_t : **input vector** $(m \times 1)$.

h_t : **hidden layer vector** $(n \times 1)$.

o_t : **output vector** $(n \times 1)$.

b_h : **bias vector** $(n \times 1)$.

U, W : **parameter matrices** $(n \times m)$.

V : **parameter matrix** $(n \times n)$.

σ_h, σ_y : **activation functions.**

Feed-Forward:

$$h_t = \sigma_h(i_t) = \tanh(U_h x_t + V_h h_{t-1} + b_h)$$
$$\hat{y}_t = \sigma_y(o_t) = \textbf{softmax}(W_y h_t + b_h)$$

Entropy loss:

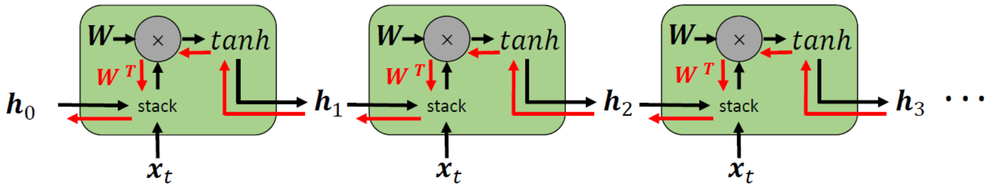
$$E(\hat{y}, y) = \sum_t E_t(\hat{y}_t, y_y) = -\sum_t y_t \log \hat{y}_t$$

Backpropagation: $\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$

$$\Pi_t = \frac{\partial E_t}{\partial o_t}\frac{\partial o_t}{\partial h_t} + \frac{\partial h_{t+1}}{\partial h_t}\Pi_{t+1}$$
$$\beta_t^U = \beta_{t+1}^U + \Pi_t \frac{\partial h_t}{\partial U_t} \qquad \beta_t^V = \beta_{t+1}^V + \Pi_t \frac{\partial h_t}{\partial V_t}$$
$$\frac{\partial E}{\partial X} \equiv \beta_0^x$$

Gradient flow in RNN

Gradient flow



Computing gradient of h_0 involves many factors of W^T (and repeated $tanh$)

- Largest singular value >1: **Exploding gradients**
- Largest singular value < 1: **Vanishing gradients**

$\rightarrow$ **Gradient clipping**: scale the gradient if its norm exceeds a threshold

$\rightarrow$ **Change RNN architecture**

The Challenge of Long-Term Dependencies

"the clouds are in the sky"

- If we are trying to predict the last word in this sentence, we don't need any further context – it's pretty obvious the next word is going to be sky.
- In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.

"I grew up in France... I speak fluent French."

- Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back.
- It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.
- Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

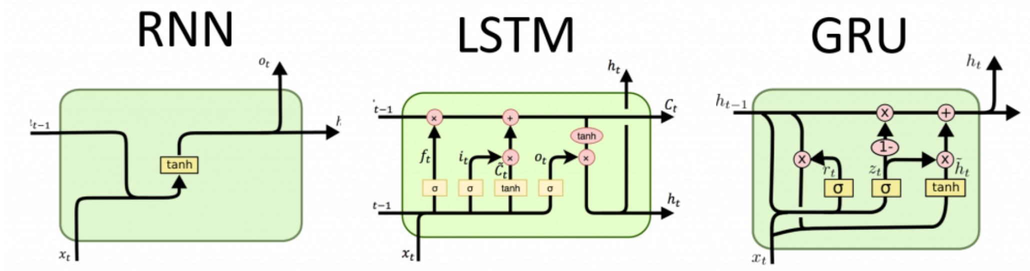
In theory, RNNs are absolutely capable of handling such "long-term dependencies." Sadly, in practice, RNNs don't seem to be able to learn them due to exploding and vanishing gradient.

Section 4

LSTM - An Advanced RNN Architecture

From RNN (above) to LSTM (below)

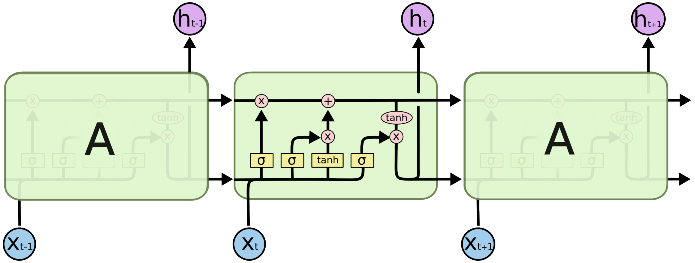
- **Long Short-Term Memory (LSTM)** and **Gated Recurrent Unit (GRU)** are more complex recurrent architectures that have been proposed [2, 1] to overcome the issues in the gradient flow and to ease the learning of long-term dependencies.
- Remembering information for long periods of time is practically their default behavior, not something they struggle to learn, thanks to the introduction of **learnable gating mechanisms**.



From RNN to LSTM

- It is very similar, it stills take into account the input and the last state but now the combination of both is more complex and works better.
- With RNN we have one vector h at each time step. But with LSTM we have two vectors at each time step h_t and c_t.
- Moreover in RNNs the repeating module only has one layer, *tanh*. Instead, LSTM has four.

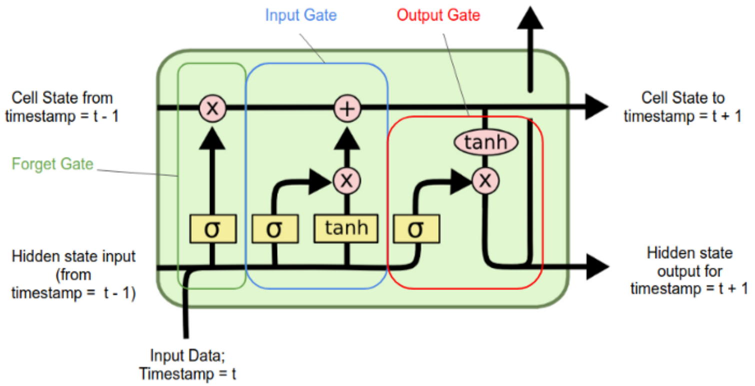
The repeating module in an LSTM contains four interacting layers.



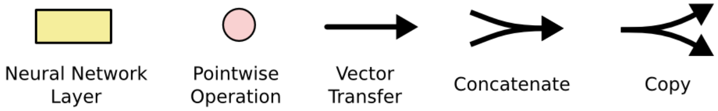
This entire rectangle is called an LSTM "cell"

These are the parts that make up the LSTM cell:

- The "Cell State"
- The "Hidden State"
- The Gates:
  "Forget" or also "Remember", "Input", and "Output".


Input Gate
Output Gate
Cell State from timestamp = t - 1
Cell State to timestamp = t + 1
Forget Gate
tanh
σ
σ
tanh
σ
Hidden state input (from timestamp = t - 1)
Hidden state output for timestamp = t + 1
Input Data; Timestamp = t

Notations:



"Cell State" vs "Hidden State"

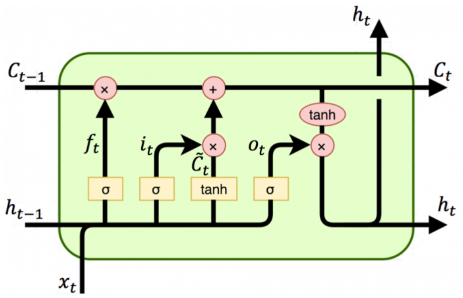
There is usually a lot of confusion between the "Cell State" and the "Hidden State". The two are clearly different in their function.

The cell state:

- The cell state is basically the global or aggregate memory of the LSTM network over all time-steps.
- the result of both forget and input gates are incorporated into the cell state from processing the previous time-step and gets passed on to get modified by the next time-step yet again.
- As a result, not all time-steps are incorporated equally into the cell state — some are more significant, or worth remembering, than others.
- This is what gives LSTMs their characteristic ability of being able to dynamically decide how far back into history to look when working with time-series data.

The hidden state:

- hidden state is more concerned with the most recent time-step;
- hidden state does not equal the output or prediction;
- It encodes a kind of characterization of the previous time-step's data.
- Characterization = abstract term that may have different meanings based on what you want the LSTM network to do



Notations:

h_t : **hidden layer vectors.**

C_t : **cell state vectors.**

x_t : **input vector.**

b_f, b_i, b_c, b_o : **bias vector.**

W_f, W_i, W_c, W_o : **parameter matrices.**

$\sigma, \tanh$: **activation functions.**

Feed-Forward:

$$
\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
\tilde{C}_t &= \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
h_t &= o_t \odot \tanh(C_t)
\end{aligned}
$$

Loss function:

$$
\begin{aligned}
L(\hat{y}, y) &= \sum_{t=1}^{T} L_t(\hat{y}_t, y_t) = -\sum_{t}^{T} y_t \log \hat{y}_t \\
&= -\sum_{t=1}^{T} y_t log\,[softmax(o_t)]
\end{aligned}
$$

Backpropagation:

We are given the following upstream gradients: $\frac{\partial L}{\partial c_t}$, $\frac{\partial L}{\partial h_t}$

The objective is to find $ifog$ gate gradients and matrix weight gradients: $\frac{\partial L}{\partial i}, \frac{\partial L}{\partial f}, \frac{\partial L}{\partial o},$ $\frac{\partial L}{\partial C}, \frac{\partial L}{\partial W_f}, \frac{\partial L}{\partial W_i}, \frac{\partial L}{\partial W_c}$, $\frac{\partial L}{\partial W_o}$, $\frac{\partial L}{\partial b_f}$, $\frac{\partial L}{\partial b_i}$, $\frac{\partial L}{\partial b_c}$, $\frac{\partial L}{\partial b_o}$, $\frac{\partial L}{\partial b}$, $\frac{\partial L}{\partial x}$, $\frac{\partial L}{\partial h_0}$

Backpropagation:

$$\frac{\partial C_{t+1}}{\partial h_t} = \frac{\partial C_{t+1}}{\partial \tilde{C}_{t+1}}\frac{\partial \tilde{C}_{t+1}}{\partial h_t} + \frac{\partial C_{t+1}}{\partial f_{t+1}}\frac{\partial f_{t+1}}{\partial h_t} + \frac{\partial C_{t+1}}{\partial t_{t+1}}\frac{\partial i_{t+1}}{\partial h_t}$$

$$\frac{\partial C_{t+1}}{\partial C_t}$$

$$\frac{\partial h_{t+1}}{\partial C_t} = \frac{\partial h_{t+1}}{\partial C_{t+1}}\frac{\partial C_{t+1}}{\partial C_t}$$

$$\frac{\partial h_{t+1}}{\partial h_t} = \frac{\partial h_{t+1}}{\partial C_{t+1}}\frac{\partial C_{t+1}}{\partial h_t} + \frac{\partial h_{t+1}}{\partial o_{t+1}}\frac{\partial o_{t+1}}{\partial h_t}$$

$$\Pi_t = \frac{\partial E_t}{\partial h_t} + \frac{\partial h_{t+1}}{\partial h_t}\Pi_{t+1} + \frac{\partial C_{t+1}}{\partial h_t}\mathcal{T}_{t+1}$$

$$\mathcal{T}_t = \frac{\partial E_t}{\partial h_t}\frac{\partial E_t}{\partial C_t} + \frac{\partial h_{t+1}}{\partial C_t}\Pi_{t+1} + \frac{\partial C_{t+1}}{\partial C_t}\mathcal{T}_{t+1}$$

$$\beta_t^f = \beta_{t+1}^f + \frac{\partial C_t}{\partial f_t}\frac{\partial f_t}{\partial W_t^f}\left(\frac{\partial h_t}{\partial C_t}\Pi_t + \mathcal{T}_t\right)$$

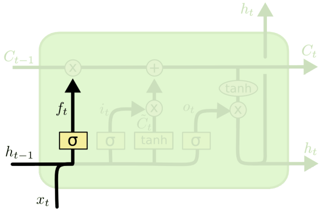
$$\beta_t^i = \beta_{t+1}^i + \frac{\partial C_t}{\partial i_t}\frac{\partial i_t}{\partial W_t^i}\left(\frac{\partial h_t}{\partial C_t}\Pi_t + \mathcal{T}_t\right)$$

$$\beta_t^c = \beta_{t+1}^c + \frac{\partial C_t}{\partial \tilde{C}_t}\frac{\partial \tilde{C}_t}{\partial W_t^c}\left(\frac{\partial h_t}{\partial C_t}\Pi_t + \mathcal{T}_t\right)$$

$$\beta_t^o = \beta_{t+1}^o + \frac{\partial h_t}{\partial o_t}\frac{\partial o_t}{\partial W_t^o}(\Pi_t)$$

First Step

Decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A 1 represents "completely keep this" while a 0 represents "completely get rid of this." For example, in a language model trying to predict the next word based on all the previous ones the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.

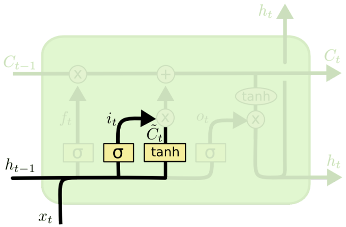


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

First: decide what information we're going to throw away from the cell state

Second

Decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a *tanh* layer creates a vector of new candidate values, $\widetilde{C}_t$, that could be added to the state. In the next step, we'll combine these two to create an update to the state. For example, in the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.

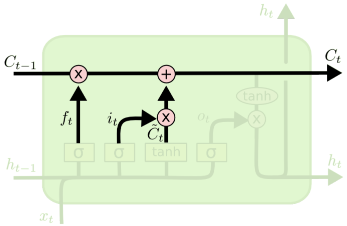


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Second: decide what information we're going to store in the cell state

Third

Update the old cell state, C_{t-1}, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it. We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * \widetilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value. In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.

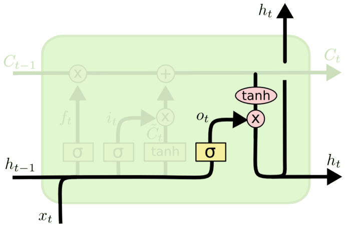


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Third, update the old cell state

Fourth

Finally, we decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through *tanh* (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to. For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.

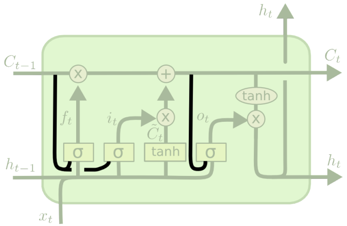


$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$

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

Peephole

Not all LSTMs are the same as the above. In fact, it seems like almost every paper involving LSTMs uses a slightly different version. The differences are minor, but it's worth mentioning some of them. Greff, et al. (2015) do a nice comparison of popular variants, finding that they're all about the same.

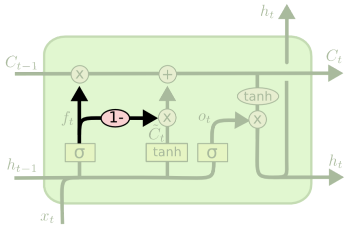


$$
\begin{aligned}
f_t &= \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_f\right) \\
i_t &= \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \ + \ b_i\right) \\
o_t &= \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \ + \ b_o\right)
\end{aligned}
$$

One popular LSTM variant, introduced by Gers & Schmidhuber (2000), is adding "peephole connections." This means that we let the gate layers look at the cell state. The diagram adds peepholes to all the gates, but many papers will give some peepholes and not others.

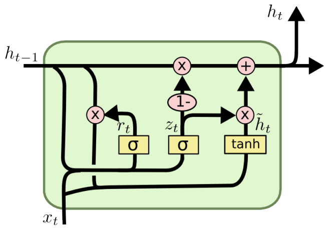
Compound forget-input gates

Another variation is to use coupled forget and input gates. Instead of separately deciding what to forget and what we should add new information to, we make those decisions together. We only forget when we're going to input something in its place. We only input new values to the state when we forget something older.



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

Update gates



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

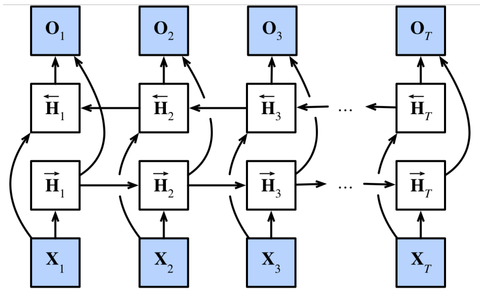
A slightly more dramatic variation on the LSTM is the Gated Recurrent Unit, or GRU, introduced by Cho, et al. (2014). It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.

These are only a few of the most notable LSTM variants. There are lots of others, like Depth Gated RNNs by Yao, et al. (2015). There's also some completely different approach to tackling long-term dependencies, like Clockwork RNNs by Koutnik, et al. (2014).

Bidirectional RNN

- A Bidirectional Recurrent Neural Network is a type of Neural Network that contains two RNNs going into different directions.
- The forward RNN reads the input sequence from start to end, while the backward RNN reads it from end to start.
- The two RNNs are stacked on top of each others and their states are typically combined by appending the two vectors.
- Bidirectional RNNs are often used in Natural Language problems, where we want to take the context from both before and after a word into account before making a prediction.
- Formally for any time step t, we consider a minibatch input $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs in each example: d).
- In the bidirectional architecture, the forward and backward hidden states for this time step are $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ and $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ respectively, where h is the number of hidden units.

Bidirectional RNN


$\mathbf{O}_1$
$\mathbf{O}_2$
$\mathbf{O}_3$
$\mathbf{O}_T$
$\overleftarrow{\mathbf{H}}_1$
$\overleftarrow{\mathbf{H}}_2$
$\overleftarrow{\mathbf{H}}_3$
$\overleftarrow{\mathbf{H}}_T$
$\overrightarrow{\mathbf{H}}_1$
$\overrightarrow{\mathbf{H}}_2$
$\overrightarrow{\mathbf{H}}_3$
$\overrightarrow{\mathbf{H}}_T$
$\mathbf{X}_1$
$\mathbf{X}_2$
$\mathbf{X}_3$
$\mathbf{X}_T$

The forward and backward hidden state updates are as follows:

$$\overrightarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}),$$
$$\overleftarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}),$$

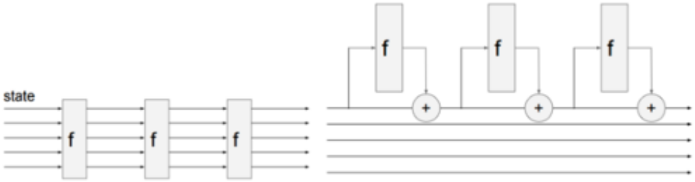
where $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d\times h}$, $\mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h\times h}$, $\mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d\times h}$, and $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h\times h}$ and biases $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1\times h}$ and $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1\times h}$ are all the model parameters.

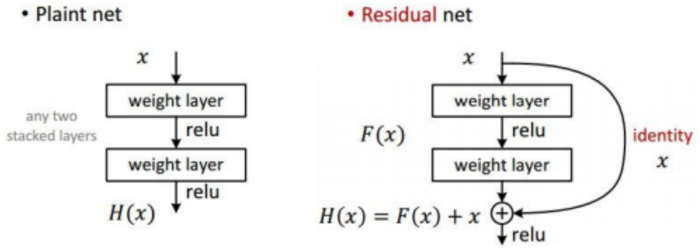
Next, $\mathbf{H}_t = \overrightarrow{\mathbf{H}}_t \oplus \overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n\times 2h}$ (concatenation) is fed into the output layer. In deep bidirectional RNNs with multiple hidden layers, such information is passed on as input to the next bidirectional layer. Last, the output layer computes the output $\mathbf{O}_t \in \mathbb{R}^{n\times q}$ (number of outputs: q):

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q.$$

where $\mathbf{W}_{hq} \in \mathbb{R}^{2h\times q}$ and bias $\mathbf{b}_q \in \mathbb{R}^{1\times q}$ are the model parameters of the output layer.

RNN vs. LSTM





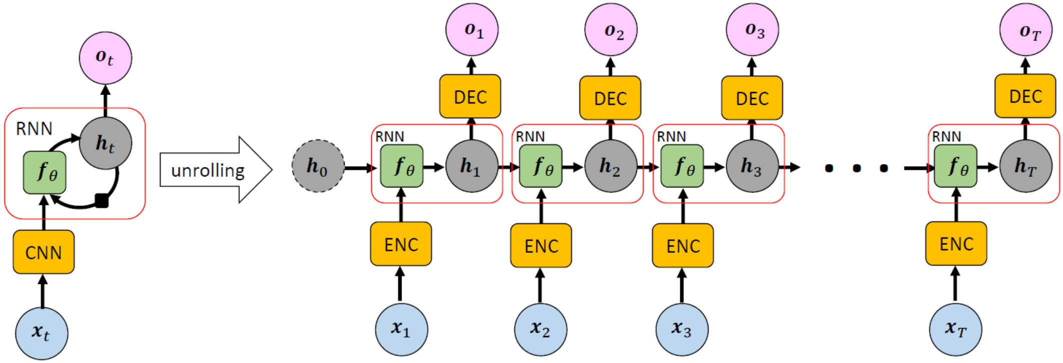
Difference between RNN (**left**) and LSTM (**right**)

RNN vs. LSTM

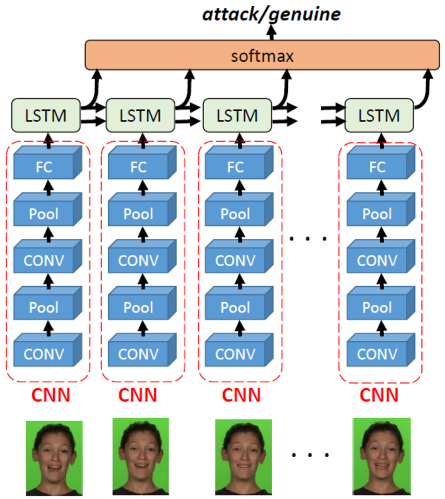
- RNN transformative interaction of the state, LSTM additive interaction of the state.
- In RNN you are operating and transforming the state vector. So you are changing your hidden state from time step to time step. Instead, LSTM has these cells states flowing through. A subset of cells (or all of them) to compute the hidden state. Then, based on hidden state, we decide how to operate over the cell. We can reset it and/or adding interaction.
- RNNs look identical to plain nets, LSTMs to ResNets.
- RNNs has vanishing gradients, so you can not learn dependencies between distant time steps.LSTMs do not have the problem of vanishing gradients. In a video in evernote they introduce gradient noise to a layer and show how it evolves. We can see that RNN automatically kills it while LSTM maintains it more time. This means that RNN can not learn long term relationships.

A CNN learns data representation to the RNN input

- A first CNN (encoder) learns data representation to the RNN input.
- A second CNN (decoder) provides the RNN output.



Combining CNN and RNN: example



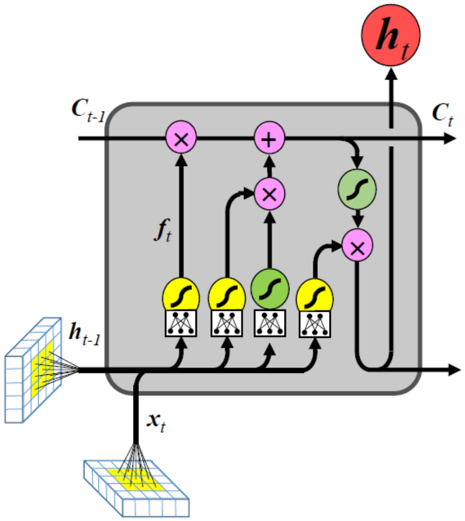
Example: Face anti-spoofing from video:

- A CNN learns a representation and provides the input to a LSTM
- LSTM learns temporal structure
- The softmax decides if it is an ***attack*** or ***genuine***.

Xu et al., 2015: https://ieeexplore.ieee.org/document/7486482

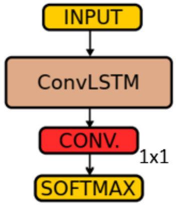
ConvLSTM

- The internal matrix multiplications are replaced by convolution operations.
- Data flows through the ConvLSTM cells keeping the input dimension (e.g., 3D) instead of just being a 1D feature vector.
- The same idea can be applied to GRU.



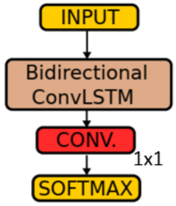
Fully Convolutional Recurrent Networks

(a)

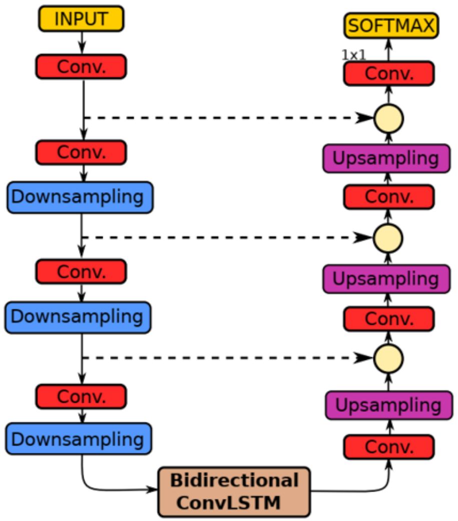


UConvLSTM

(b)



BConvLSTM



BUnetConvLSTM

Section 5

Summary and Credits

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.

Credits i

These slides heavily borrow from a number of awesome sources. I'm really grateful to all the people who take the time to share their knowledge on this subject with others.

In particular:

- Stanford CS231n Convolutional Neural Networks for Visual Recognition
  http://cs231n.stanford.edu/
- Stanford CS20SI TensorFlow for Deep Learning Research
  http://web.stanford.edu/class/cs20si/syllabus.html
- Deep Learning Book (GoodFellow, Bengio, Courville)
  http://www.deeplearningbook.org/

Credits ii

- Marc'Aurelio Ranzato, "Large-Scale Visual Recognition with Deep Learning"
  www.cs.toronto.edu/~ranzato/publications/ranzato_cvpr13.pdf
- Convolution arithmetic animations
  https://github.com/vdumoulin/conv_arithmetic
- Andrej Karphathy personal blog
  http://karpathy.github.io/
- WildML blog on AI, DL and NLP
  http://www.wildml.com/
- Michael Nielsen Deep Learning online book
  http://neuralnetworksanddeeplearning.com/

References i

[1] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio.
Learning phrase representations using rnn encoder-decoder for statistical machine translation.
arXiv preprint arXiv:1406.1078, 2014.

[2] S. Hochreiter and J. Schmidhuber.
Long short-term memory.
Neural computation, 9(8):1735–1780, 1997.

[3] R. Pascanu, T. Mikolov, and Y. Bengio.
On the difficulty of training recurrent neural networks.
In *International Conference on Machine Learning*, pages 1310–1318, 2013.