Long Short - Term Memory (LSTM)

Sepp Hochreiter & Jürgen Schmidhuber (1997)

By: Rey Reza Wiyatno

Agenda

- Cool RNN/LSTM applications
- Background
- LSTM (Hochreiter & Schmidhuber, 1997) vs RNN architecture
- Forget, input, cell state, and output gates
- Backprop in LSTM
- Summary
- Demo RNN vs LSTM
- Other LSTM variants: Gated Recurrent Units or GRU (Cho et al., 2014)
- Open questions
- Q & A

The agenda for today is first we are going to see some cool RNN/LSTM applications to boost up interest in this topic. Next we will recall on standard RNN and backprop through time, then we will talk a little bit about the background on why we need LSTM and see how LSTM is different compared to standard RNN. Then we will see LSTM part by part (there are 4 main parts in LSTM and we will see how each of the part works). We will also see a simple demo on RNN vs LSTM on predicting next characters given some characters, and we will also take a quick look on a variant of LSTM called gated recurrent units or GRU, which is now being widely used. Then we will also see some of the open research questions in RNN and LSTM, so we can all start to think beyond LSTM, and have an idea of one example of research topic in this field and hopefully some of you will be interested to investigate more on this topic. Finally, we will have Q&A, but please feel free to stop me whenever you find something to be confusing. The point of this lecture is to get familiar with the concepts of LSTM, not to dig deep into the proofs since we won't have enough time to cover everything in 50 minutes. Once we get some intuitions on LSTM work, for those who are interested, I encourage you to read the papers listed in the "Reference" section.

Cool RNN/LSTM Applications

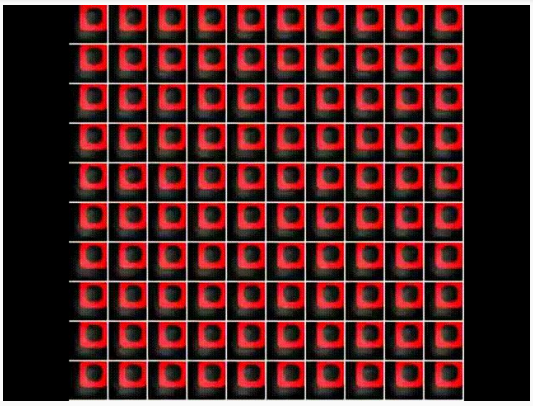
Show and Tell (Vinyal et al., 2014)

Image Captioning using Show and Tell (Vinyals et al., 2015)

One of the applications of RNN and LSTM is to do image captioning. Pretty much almost anything the relates with language, we can model it using RNN/LSTM. So you see here some of the images that are being captioned using combination of CNN (as a feature extractor) and LSTM.

Cool RNN/LSTM Applications

Deep Recurrent Attentive Writer (DRAW) (Gregor et al., 2015)



DRAW generating handwritten digits (Gregor et al., 2015)

This is from a paper called DRAW which stands for Deep Recurrent Attentive Writer. The idea is to be able to generate an image using attention mechanism per time step rather than generating an image as a whole. This actually mimics more how we (human) look at things and write. In this figure, we can see how DRAW architecture is drawing new digits that are not in the dataset, adding new stuffs on what they call as “canvas” by only looking at certain patch per time step. The redbox is the patch that it is currently looking at a certain time step. They actually use a standard RNN combined with Variational AutoEncoder (VAE), but don’t worry about VAE for now (it’s another neural network architecture for generative modelling). This is actually a pretty interesting example because it shows how neural networks can be used to model something that behaves more like a brain, so we can see how neural networks and neuroscience are starting to reinforce each other.

Cool RNN/LSTM Applications

Other Applications

- Language modelling
- Language translation
- Video (frame by frame) recognition
- Video captioning
- Question answering
- And many more!

Other applications: language modelling and translation, video recognition (we can imagine if we want to classify what is in a frame from a video, it may be more helpful if we know what is in the previous frame, because the difference between one frame usually should not look too much different from the previous frame), question answering task, and many more. So RNN/LSTM is a really useful tool.

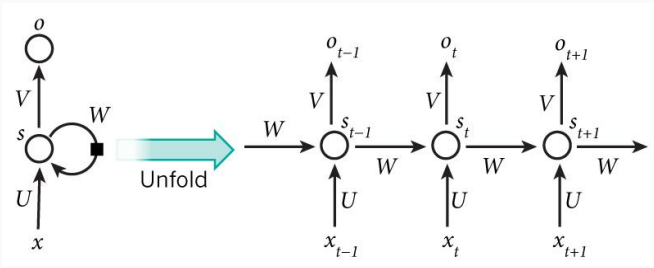
Background

- Recall: RNN + BPTT (next slide)
- RNN suffers from **vanishing** & exploding gradients (Bengio et al., 1994; Pascanu et al., 2013)
- Which makes RNN bad to perform task that rely on long term dependencies (i.e. “My friend is **annoying** …. for example …. Therefore I do not want to hang out with her.”)

The problem with RNN is that it suffers from vanishing and exploding gradients as the time dependencies gets larger and larger. So RNN is bad when we’re trying to perform task that would require long term dependencies. For example, at the time, many applications of RNN focused on natural language processing applications like predicting a word given previous words. But in real world, when we are trying to tell a story, we’re often not being to the point. So for example, here it says “My friend is annoying … “. The reason for her to not hangout with her friend is because she is annoying, but the word “annoying” may occur perhaps like 100 words before the conclusion and therefore the RNN needs to be able to handle this kind of long dependency. The problem is that as the time steps in RNN gets longer, the gradient often vanishes or explodes. It’s a bit easier to deal with exploding gradients because we can do things like gradient clipping, but it is a lot harder to deal with vanishing gradient at the time. And this is why we need some modifications to the standard RNN. Note that LSTM is not the only solution to this problem, but it was one of the major breakthroughs at the time.

Background

Vanishing/exploding gradient in RNN



Source: http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}$$

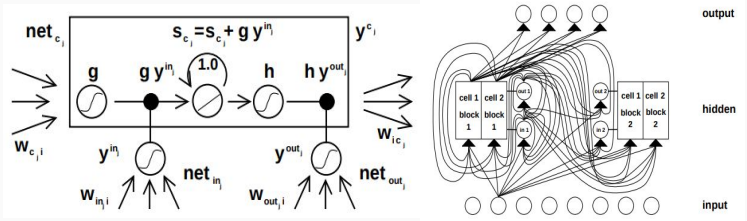
Note that:

- Every $\boldsymbol{ds_t/ds_{t-1}}$ contains $\boldsymbol{W}$
- If $\boldsymbol{W > 1}$, then $\boldsymbol{dE_\infty/dW}$ will go to ∞
- If $\boldsymbol{W < 1}$, then $\boldsymbol{dE_\infty/dW}$ will vanish to ≅ 0!

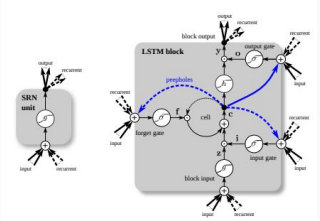
Exploding gradient -> gradient clipping
Vanishing gradient -> LSTM!

Recall on RNN and BPTT, here we see the derivative of the error at time step 3 with respect to the weights W. We can see that as the time steps grow, the gradient will be made up of bunch of multiplications between ds_t/ds_{t-1}, which contains W. We can imagine as time goes larger and larger, if W is less than 1 (in average), then dE_inf (at t = infinity) will vanish to 0. If the gradient of the error at t = infinity is 0 with respect to W, it means that we will not be able to update W such that the error at the later time steps decreases, which explains the problem with long term dependencies in standard RNN. This is the vanishing gradient problem. We can also imagine if W is > 1 (in average) then we have exploding gradient problem, but exploding gradient is easier to deal with. For example, we can just clip the gradient at a certain range (i.e. if gradient > threshold, clip to a constant value we want to be maximum). Unfortunately, vanishing gradient was hard to deal with, therefore we need LSTM.

LSTM vs RNN Architecture



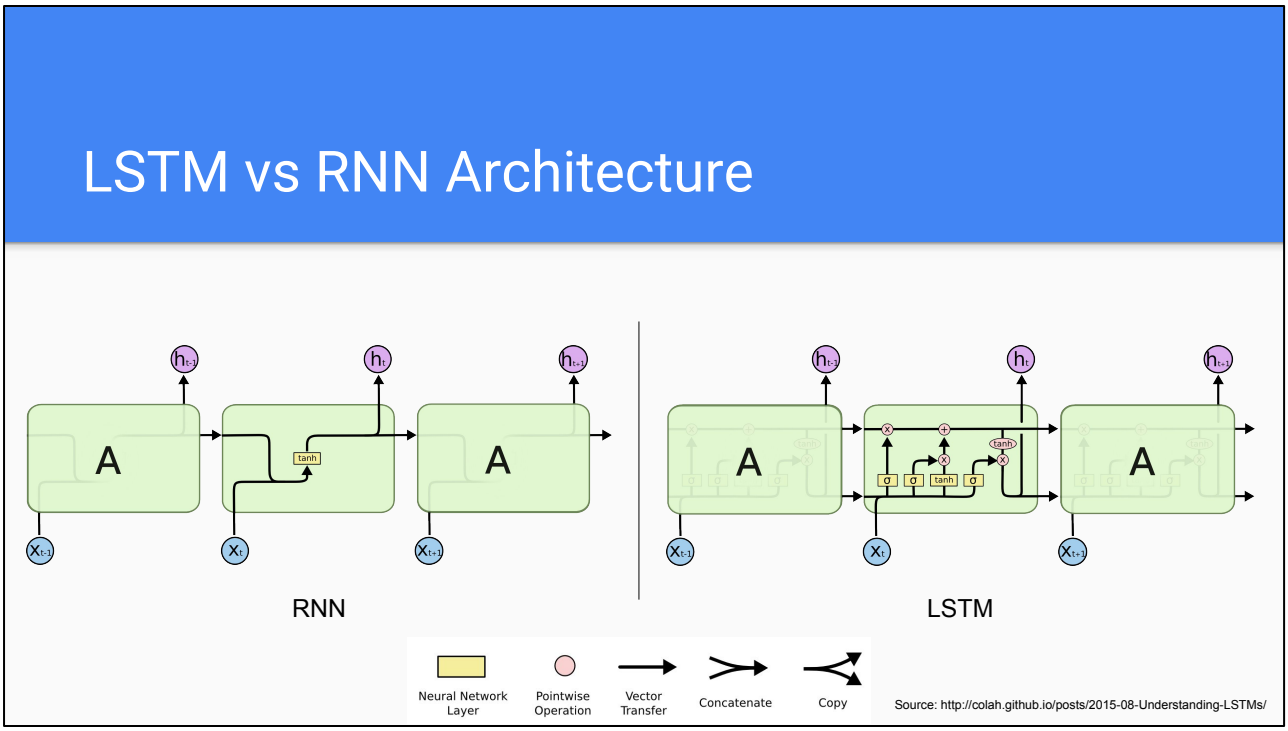
LSTM figures from the original paper (Hochreiter & Schmidhuber, 1997)



RNN and LSTM figures (Greff et al., 2015)

"The first time I saw LSTM they really scared me. I wasn't sure what's going on. I understand LSTM but I still don't know what these 2 diagrams are" - Andrej Karpathy (now Director of AI at Tesla), during CS231n W2016 lecture at Stanford

This is original LSTM figures from the 1997 paper, or if you google for LSTM figures, you will find other figures that look like these. These figures are hard to understand, especially for those who just started to learn about neural networks and RNN. Even Andrej Karpathy said this during CS231n lecture at stanford "see quote". He was referring to the right figure from original LSTM figure (left), and another figure which I am not sure which one, but I totally agree with him. The point here is, don't feel intimidated if you looked at these figures and don't understand them, sometimes they are just being targeted to different audience. Thankfully, many people in machine learning community are aware of this problem, and there have been many efforts to create simple and intuitive tutorials to understand concepts in machine learning through personal blog, medium (medium.com), or any other forms,. Which is why I'm not going to use the original figures, but instead I will use figures that are more intuitive to understand taken from Christopher Olah's blog, he is currently a research scientist at Google Brain.



As mentioned, most of the figures here I took from Christopher Olah's blog, you can check out his blog post through this link on the bottom right. I think he did a fantastic job explaining LSTM in a short blog post so it is accessible to many people because as we saw earlier, if you read the original LSTM paper, the figures were a bit harder to understand and its fairly long with all the proofs, which can be intimidating to many people. That being said, I encourage you to check out the original paper after this lecture and hopefully it will make so much more sense.

What I'm showing you here is the "unrolled" version of RNN and LSTM, by "unrolled" I mean the it is being "unrolled" with respect to time, so we can see what is happening at each time step here. So *t* here denotes the time step, and we can see the recurrency here and how each cell (green box) takes as an input information from previous time step. We can see how similar RNN and LSTM are. The difference is that rather than only having a single neural network layer with tanh activation, LSTM have four different neural network layers shown in the yellow boxes that act like "gate" as we will see soon. So LSTM is just RNN with more complex recurrency functions. The green box is usually referred as a cell. So if you see online some people talking about RNN cell or LSTM cell, they are referring to this green box. Note that all the yellow blocks are just another neural network layers, which means that the entire pipeline is differentiable. The gating here is not a piecewise function, rather it is controlled by some activation function. For example, the forget gate uses a sigmoid activation function. Intuitively we know that sigmoid outputs a number between 0 and 1, in this case 0 simply means "dont let anything through" and 1 means the opposite.

LSTM

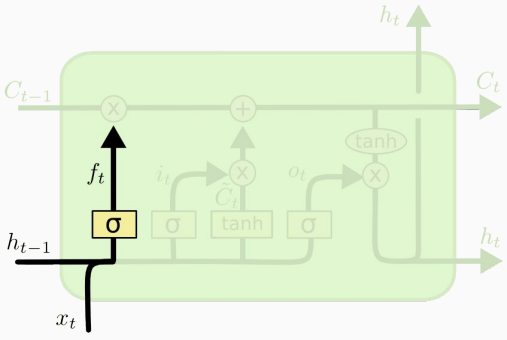
3 gates:

- Forget gate: controls how much of information from previous time steps do we care about.
- Input gate: controls how much of information from current time steps do we care about.
- Output gate: controls what the cell is outputting which depends on current cell state

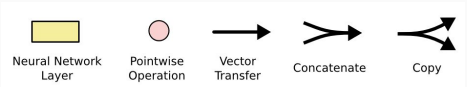
There are 3 gates in LSTM:

1. The first one is forget gate, which controls how much of information from previous time steps do we care about.
2. The second one is the input gate, which controls how much of information from current time steps do we care about.
3. Finally, there is an output gate that controls what the cell is outputting which depends on current cell state, as we will see it soon. Don’t worry if you dont know what a cell state is at this point, because we haven’t really talked about it.

Forget Gate



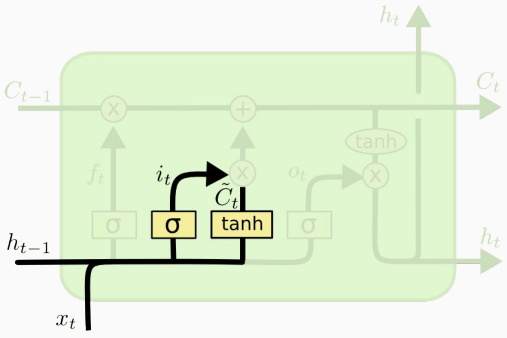
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$



Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

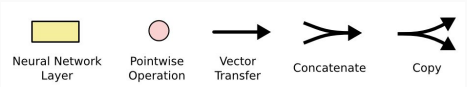
First gate is forget gate which is highlighted here so we can just focus on this particular gate. As mentioned in previous slides, the sigmoid here is just a neural network layer with sigmoid activation function. It takes as an input the concatenation of the hidden state of previous time step (h_{t-1}) and input from current time step (x_t). So we can imagine if we have 2 vectors, we just concatenate them together to become another vector. Then the rest is just like a typical neural network layer, we just need to multiply the concatenated vector with some weights plus some biases. We then apply the sigmoid to get the output of the forget gate, f_t, which is between 0 and 1 (since it is sigmoid). So again, we can see that all these operations are differentiable. Now we can see that the output of the forget gate is going to be used to scale the cell state coming from previous time step: the cell state from previous time step (C_{t-1}) is going to be multiplied by the output of forget gate (f_t). Let's leave the cell state here for now, as it will become more clear after we see how the input gate works.

Input Gate



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

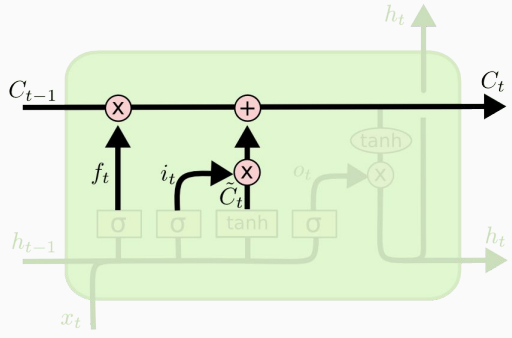
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$



Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

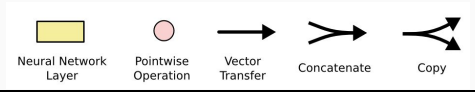
Next is the input gate, which as we mentioned before, controls how much information we need from current time step. For now, let's omit the sigmoid layer shown here. If we remember the standard RNN, the tanh layer here does exactly what the standard RNN cell does, that is to get information from X_t and h_{t-1} , but then there is an additional sigmoid layer here whose function is exactly the same as the forget gate, which is to control or scale the amount of information we need from the current time step. So C_t tilde here is just like what we get in standard RNN cell, which is then scaled by i_t. Next, using the output of the input gate, we will update the cell state by ADDING this as we will see in the next slide. We can actually probably just use 1 single layer neural network here rather than 1 sigmoid + 1 tanh, but by decoupling it into 2 neural network layers as shown here, it may probably become more expressive and can capture more information (obviously, since we're pretty much adding more parameters to the model). That said, there may be some proofs of why this is better that I am not aware of.

Cell State



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

$dC_t/dC_{t-1} = f_t$

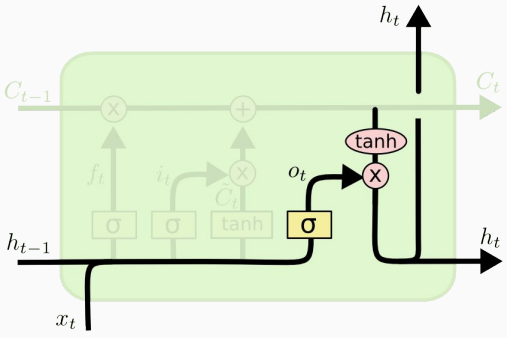


Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Here, we can see how the cell state for current time step is computed, using the output from the forget gate and input gate. This is the part why LSTM is more robust against vanishing gradient. We can see that the derivative of the current cell state with respect to previous cell state is just the output of the forget fate (f_t). This means we won't get gradient of 0 as long as the neural net thinks that the information is still relevant. The proof for this is not trivial, but if you are curious, you can check the paper as listed in the reference section at the end of these slides. But just intuitively, we can see that there is at least one path for the gradient to not vanish.

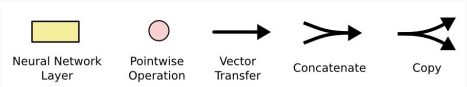
Bonus: usually the biases in forget gate is initialized as a positive number, because once the forget gate goes to 0 we can imagine the gradient will be cut off. So by initializing the biases to be positive, we let the forget gate to always output 1 at the beginning of the training process, and just let the network learn to shut it off if necessary. Remember that ideally forget gate should always output 1 or 0, but we choose sigmoid so that it is all differentiable.

Output Gate



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

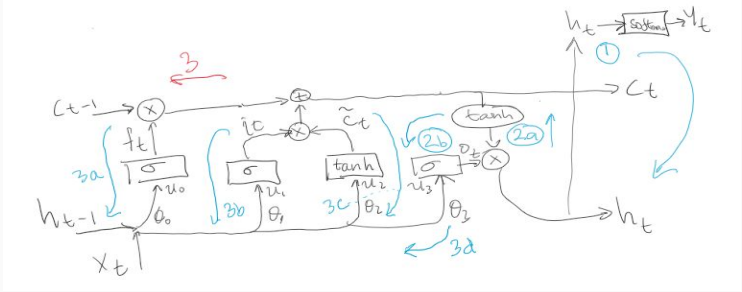
$$h_t = o_t * \tanh\left(C_t\right)$$

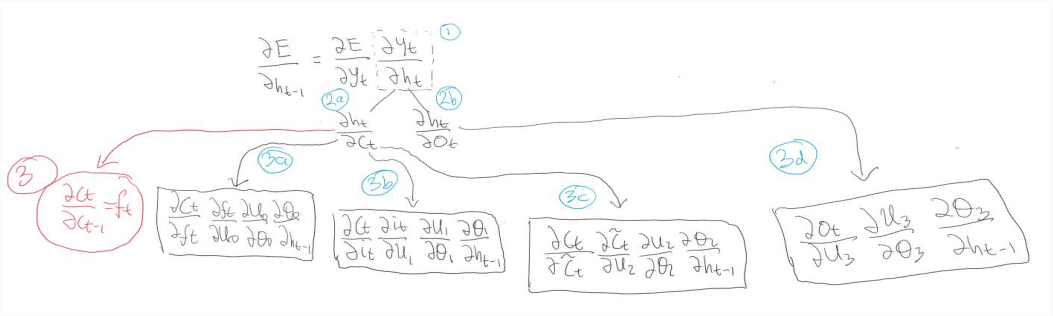


Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Finally, the output gate. The output depends on the current cell state that we just calculated but being passed to tanh (note that this tanh here is not a neural net layer, its just pointwise operation) and multiply that with o_t which is the output of this sigmoid neural net layer highlighted in the figure. This will give us the hidden state for the current time step (h_t) which we will use again as the input for the next time step. So that is how forward pass in LSTM cell works, now lets see how the backprop works.

Backprop in LSTM

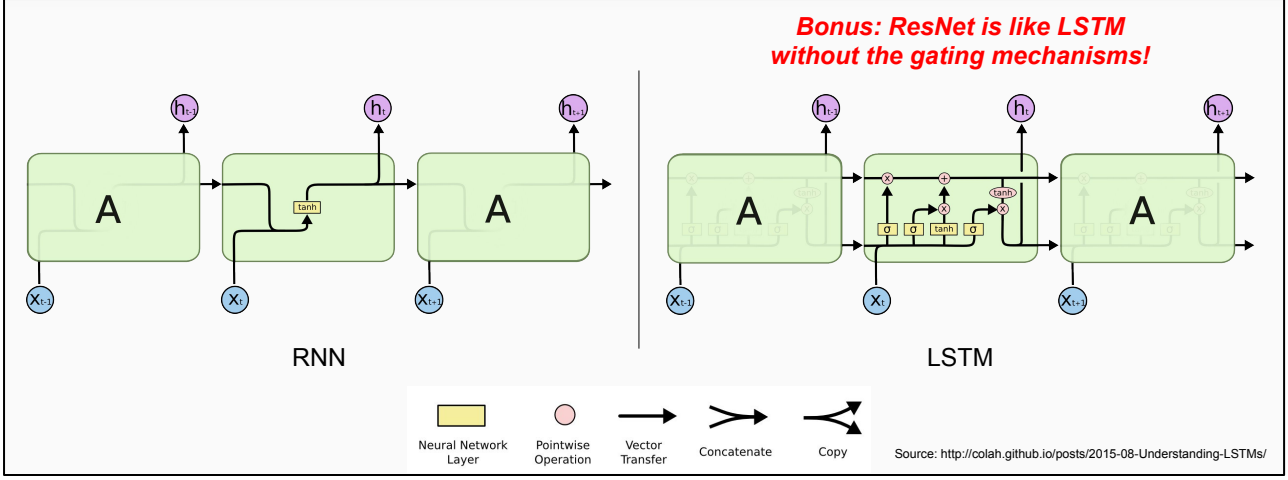




It's kind of hard trying to visualize the gradient flow in LSTM since there are many so things happening in a single LSTM cell. This slide is just to visualize the gradient flow in LSTM. We can see that the gradient flowing in (3) are always constant (equal to f_t) which will help to fight against vanishing gradient (imagine if the neural network is able to learn to always output ft = 1, then the gradient will never vanish). In practice LSTM can still suffer from vanishing gradient, but it is just less likely compared to standard RNN.

Note: sorry if my handwriting looks bad and confusing.

Summary



So just to summarize (we've seen these diagrams before from previous slides):

- LSTM has 3 gate mechanisms: forget gate, input gate, and output gate. It is very similar to RNN except that we have more complex functions to calculate the hidden state where each gate is a single layer neural network
- LSTM has an extra state called cell state denoted by C_t in previous slides in addition to the hidden states. Recall the for every time step, standard RNN only gives us a hidden state (no cell state)
- The key of LSTM is in the cell state because it gives us constant gradient which is equal to the output of the forget gate
- Bonus: If you're familiar with ResNet, it is actually very similar with LSTM because ResNet also requires us to perform addition between later layers and early layers. So, it is like LSTM without the gating mechanisms. Like LSTM, ResNet are more robust against vanishing gradient and was one of the early CNN architectures that allows us to build a very deep network (50 - 100 layers deep!)

Demo: RNN vs LSTM

input →

"Hasting to feed her fawn hid in some brake.
By this, she hears the hounds are at a bay;
Whereat she starts, like one that spies an adder
Wreathed *up in fatal folds just in his way,*
The fear whereof doth make him shake and ***shudder****;*
Even so the timorous yelping of the hounds
Appals her senses and her spirit confounds."

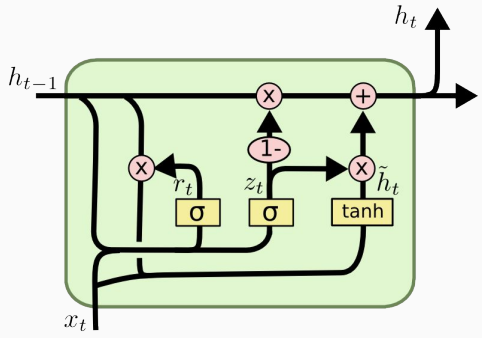
RNN's output (85 characters): by this, she hears the hounds are at a bay; whereat she starts,like one that spies an adder wreathed ***up in fatal folds just in his ears a hame the boar to-morrow.***
'what hard heart do se

LSTM's output (85 characters): by this, she hears the hounds are at a bay; whereat she starts,like one that spies an adder wreathed **up in fatal folds just in his way.**
the fear whereof doth make him shake and shudder;

Now we are going to see a simple demo and how LSTM performs compared to standard RNN. If we just compare the loss on shakespeare.txt, we can see that the loss after 250 epochs is lower when we replace the RNN cell as LSTM cell. This may have been for several different reasons. The first reason is that this maybe due to the fact that LSTM has almost 4x more parameters compared to RNN, but it's really hard to compare them head to head by just naively comparing the number of parameters.

One of more intuitive reason on why this may have happened is that, if we take a look at shakespeare.txt, at the bottom are some of poems from shakespeare. And usually poems have some rhymes, some sort of linguistic patterns, for example like shown above the word "adder" was then followed by "shudder" after one line. I suspect that RNN can't get this long term dependencies while LSTM can. This turns out to be true, If i feed as an input the characters shown on the above in bold "By this, she hears ...", we can see the prediction from RNN does not match with the ground truth even though these texts are in the dataset, so RNN can't even overfit to the dataset, while as we can see, LSTM can.

LSTM Variants - Gated Recurrent Unit (GRU) (Cho et al., 2014)

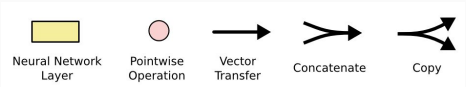


$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

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

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

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



Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Just to give another taste of LSTM. GRU is an LSTM variant that is more efficient and becoming widely used as well. We can see that it is also using some sort of “gating” mechanisms, and somewhat similar to LSTM but without having the cell state that exists in LSTM. We are not going to dig in to the detail of GRU, but curious students are encourage to read the paper or any blogs/tutorials that explain how GRU works. The point here is that LSTM is not a fixed design, you can always be creative and further improve on LSTM.

Open Questions

- Is LSTM an optimal design? Consider what they did not know in 1997, what can we do to avoid vanishing/exploding gradient?
  - For example, Le et al. (2015) shows that RNN + ReLU + proper initialization can produce similar effect as LSTM and the results are comparable with 4x less parameters
- Is GRU an optimal design?

This slide is to encourage us to think beyond LSTM. LSTM is stable and widely used in the industry, but it is certainly not an optimal design. Let's start with the motivation of LSTM, which as we mentioned before was to deal with vanishing gradient problem. However, it was in 1997, and as we all know, deep learning only took off in 2012 and we have learned a lot more about neural nets since 2012. For example, it has been shown that ReLU can also help to avoid vanishing gradient. How would this help standard RNN to escape from vanishing gradient? There is a paper from Le et al. (2015) that shows how RNN with ReLU (instead of tanh) with proper weight initialization can compete with the performance of LSTM with 4x less number of parameters for certain tasks. As a practitioner, we need to be able to answer this question. Hopefully this will give an example of an open research area in LSTM, and sparks some research interest in this particular field for some of you.

References

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw., 5(2):157–166, March 1994*. ISSN 1045-9227. doi: 10.1109/72.279181. URL http://dx.doi.org/10.1109/72.279181.

Kyunghyun Cho, Bart van Merrienboer, C¸alar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. *In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 1724–1734, Doha, Qatar, October 2014*. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/D14-1179.

Klaus Greff, Rupesh Kumar Srivastava, Jan Koutnık, Bas R. Steunebrink, and Jurgen Schmidhuber.LSTM: A search space odyssey. CoRR, abs/1503.04069, 2015. URL http://arxiv.org/abs/1503.04069.

Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. *In Francis Bach and David Blei (eds.), Proceedings of the 32nd International Conference on Machine Learning, volume 37 of Proceedings of Machine Learning Research, pp. 1462–1471, Lille, France, 07–09 Jul 2015*. PMLR. URL http://proceedings.mlr.press/v37/gregor15.html.

Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. *Neural Comput., 9(9):1735–1780, November 1997*. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. A simple way to initialize recurrent networks of rectified linear units. CoRR, abs/1504.00941, 2015. URL http://arxiv.org/abs/1504.00941.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *In Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, ICML'13, pp. III–1310–III–1318*. JMLR.org, 2013. URL http://dl.acm.org/citation.cfm?id=3042817.3043083.

Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. CoRR, abs/1411.4555, 2014. URL http://arxiv.org/abs/1411.4555.

This is the reference slides. Curious students are encouraged to check out these papers, especially if they want to learn more about the proof on why LSTM does not suffer from vanishing gradient and the historical background of LSTM, as well as how to improve LSTM/RNN.

Questions?

Thank You

Email:
reywiyatno@gmail.com

Github:
https://github.com/rrwiyatn

Feel free to contact me if there is anything about LSTM that you don't understand. All the codes are available on my github repo (deeplearning-ai), I also have some other open source deep learning projects like image segmentation, GANs, super resolution, style transfer, etc., so please feel free to check it out.