Advanced Topics in Machine Learning

Alejo Nevado-Holgado

Lecture 13 (NLP 5) - Vanishing gradients and fancy RNNs

V 0.4 (23 Feb 2020 - final version)

Feedback so far

- Ask students if they have questions and ask questions to them [+1/ 0]
- Define all the technical terms that you use (e.g. hyperparameters) [+1/ 0]
- Sometimes you talk too fast [+1/ 0]
- The speed/amount of material is good [+3/ 0]
- The spoken descriptions of the equations, and why they are used, are very useful [+1/ 0]

Course structure

- **Introduction:** What is NLP. Why it is hard. Why NNs work well ← Lecture 9 (NLP 1)
- **Word representation:** How to represent the meaning of individual words
  - Old technology: One-hot representations, synsets ← Lecture 9 (NLP 1)
  - Embeddings: First trick that boosted the performance of NNs in NLP ← Lecture 9 (NLP 1)
    - Word2vec: Single layer NN. CBOW and skip-gram ← Lecture 10 (NLP 2)
    - Co-occurrence matrices: Basic counts and SVD improvement ← Lecture 10 (NLP 2)
    - Glove: Combining word2vec and co-occurrence matrices idea ← Lecture 10 (NLP 2)
    - Evaluating performance of embeddings ← Lecture 10 (NLP 2)
- **Named Entity Recognition (NER):** How to find words of specific meaning within text
  - Multilayer NNs: Margin loss. Forward- and back-propagation ← Lecture 11 (NLP 3)
  - Better loss functions: margin loss, regularisation ← Lecture 11 (NLP 3)
  - Better initializations: uniform, xavier ← Lecture 11 (NLP 3)
  - Better optimizers: Adagrad, RMSprop, Adam... ← Lecture 11 (NLP 3)

Course structure

- **Language modelling:** How to represent the meaning of full pieces of text
  - Old technology: N-grams ← Lecture 12 (NLP 4)
  - Recursive NNs language models (RNNs) ← Lecture 12 (NLP 4)
  - Evaluating performance of language models ← Lecture 12 (NLP 4)
  - Vanishing & exploding gradients: Problem. Gradient clipping ← Lecture 13 (NLP 5)
  - Improved RNNs: LSTM, GRU, Bidirectional... ← Lecture 13 (NLP 5)
- **Machine translation:** How to translate text
  - Old technology: Georgetown–IBM experiment and ALPAC report ← Lecture 16 (NLP 6)
  - Seq2seq: Greedy decoding, encoder-decoder, beam search ← Lecture 16 (NLP 6)
  - Attention: Simple attention, transformers, reformers ← Lecture 16 (NLP 6)
  - Evaluating performance: BLEU ← Lecture 16 (NLP 6)

Introduction: Why Neural Networks?

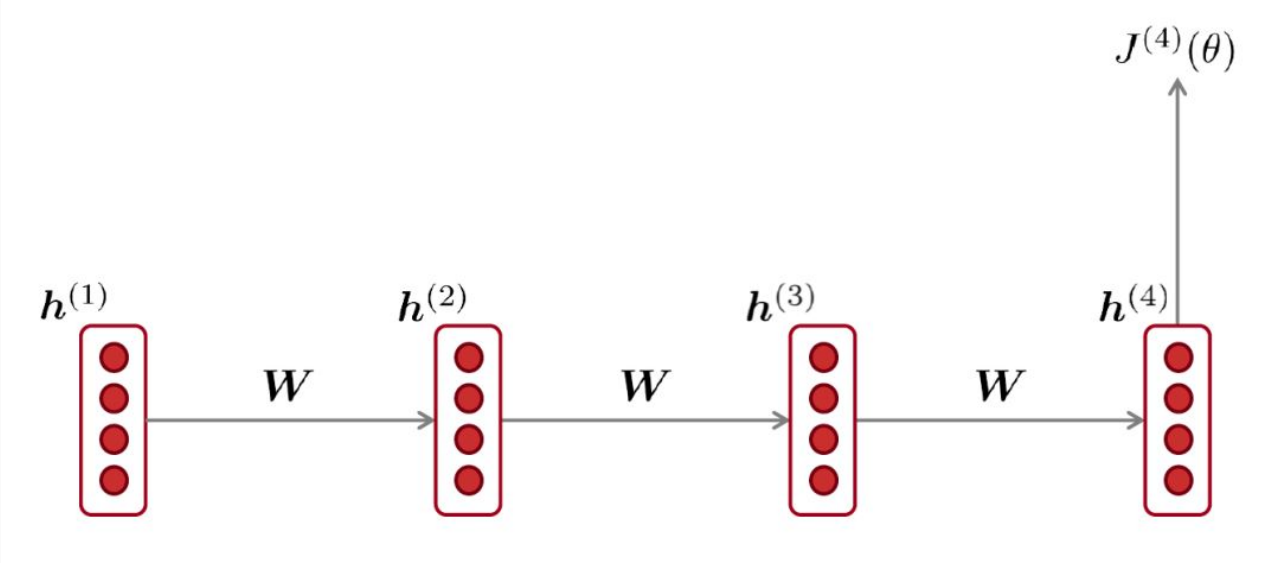
- In about 2010, deep learning techniques started outperforming other machine learning techniques. Why this decade?
  - Better **data**, or rather, much much more data is available
  - Better **hardware**, such as GPUs, had drastically developed during 2000s
  - Better **software**, such as new models, algorithms, and ideas:
    - better, more flexible learning of intermediate representations
    - effective end-to-end joint system learning
    - effective learning methods for using contexts and transferring between tasks
    - better regularization and optimization methods

⇒ improved performance (first in speech and vision, then NLP)

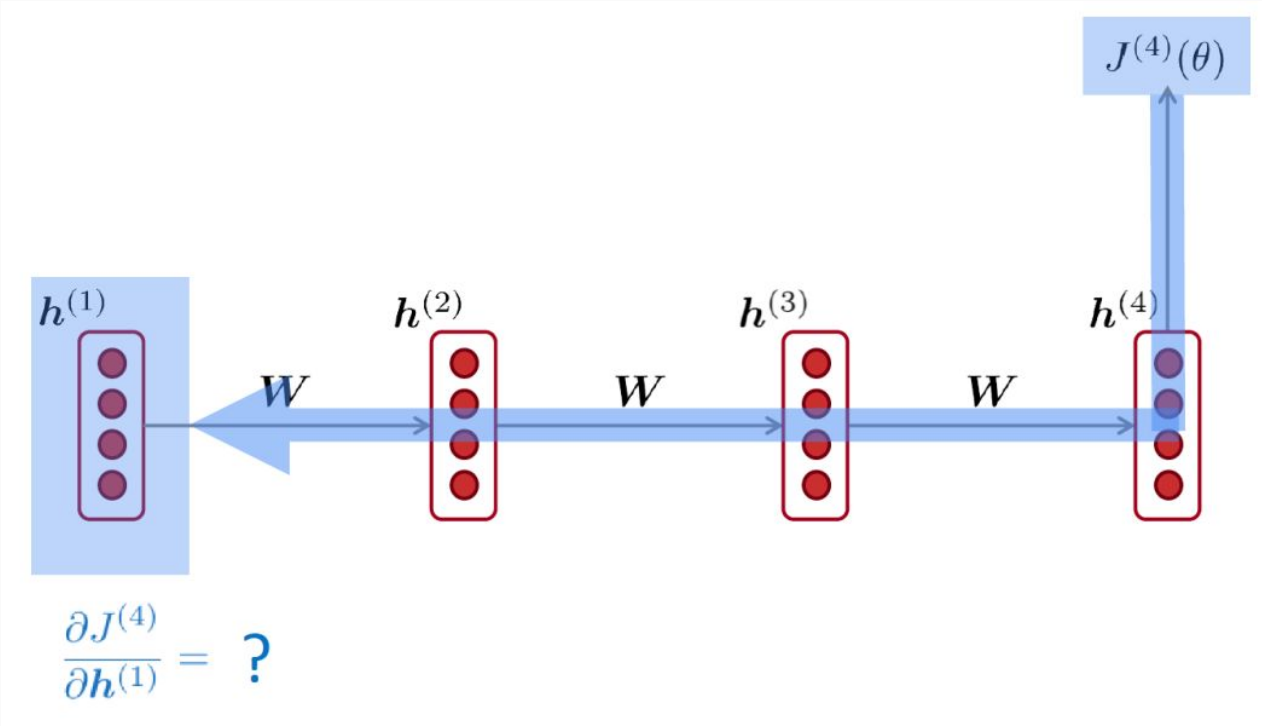
Vanishing & exploding gradients

- **Problem:** When NNs become deep, gradients tend to either vanish or explode. This is the vanishing and exploding gradients problem, and it is specially serious in RNNs
- **A bit of history:** In the "old times of NNs" (until ~2010!) this was the core **software** reason why NN did not perform as well as now. The first method they (the deep mafia) designed to deal with the problem was layerwise training. Later improvements (careful weight initialisation like Xavier's, batch normalisation, non saturating activation functions, residual networks) further reduced the problem in feed forwards NNs, even in very deep ones. However, RNNs are ∞ deep, and their weights are re-applied over and over through the recurrent connection, making it impossible to eliminate the problem.
- What finally solved the problem were memory cells where writing and reading are controlled by internal activation functions → gradient clipping, LSTMs and GRUs

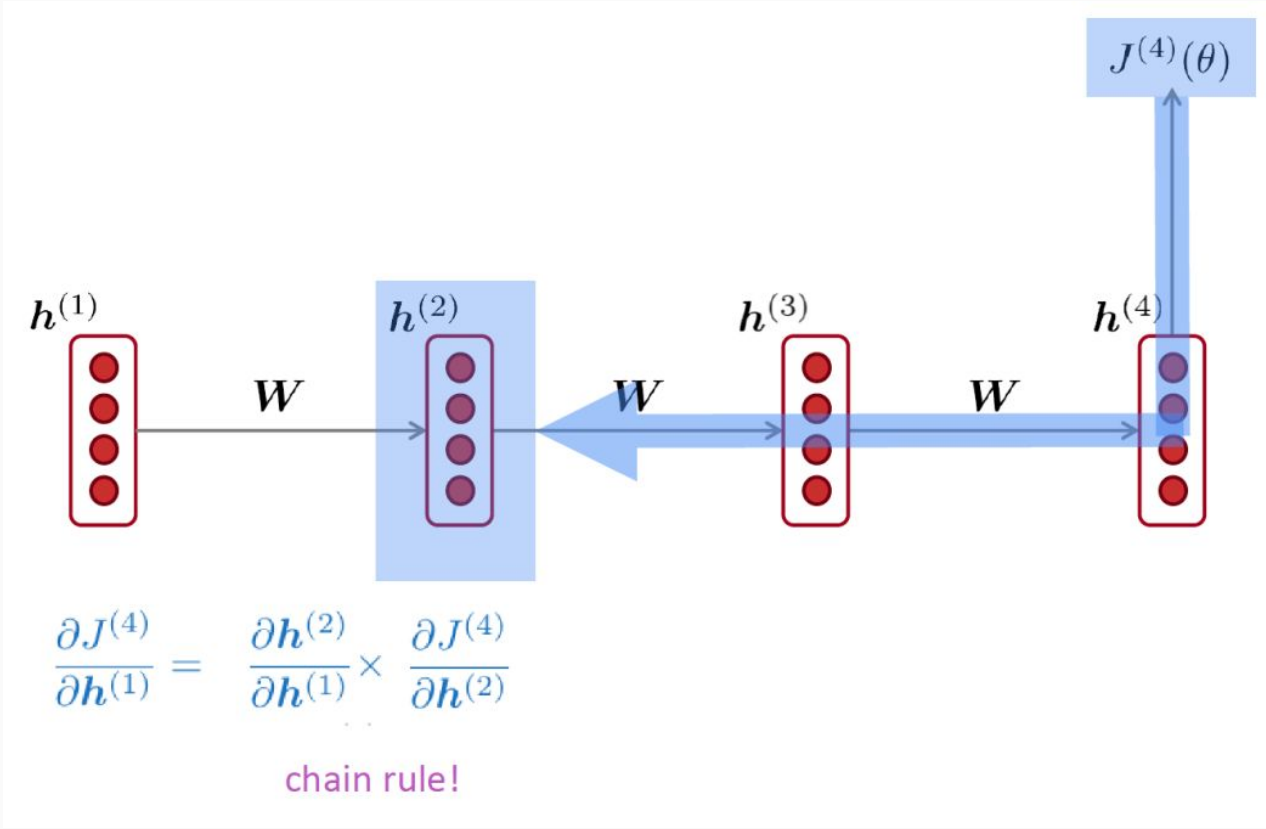
Van & exp gradients: The problem



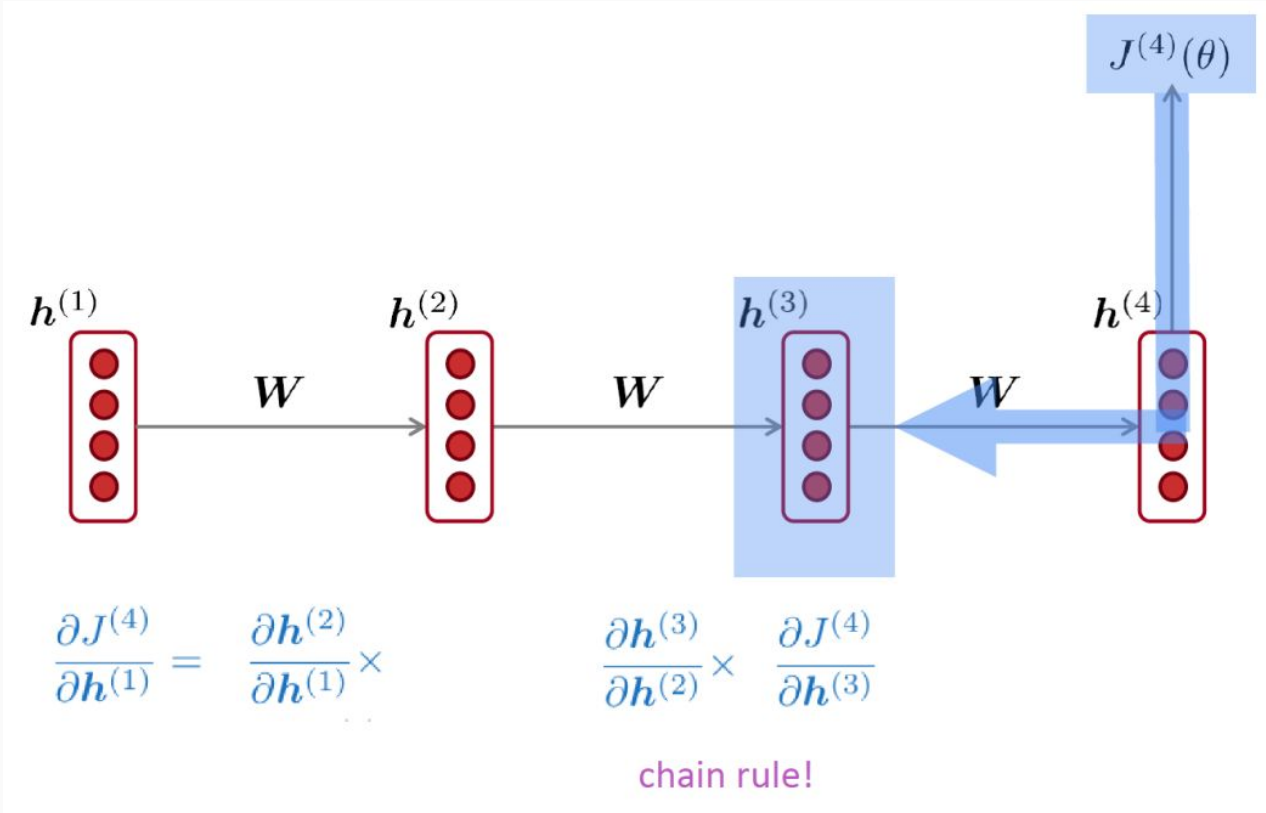
Van & exp gradients: The problem



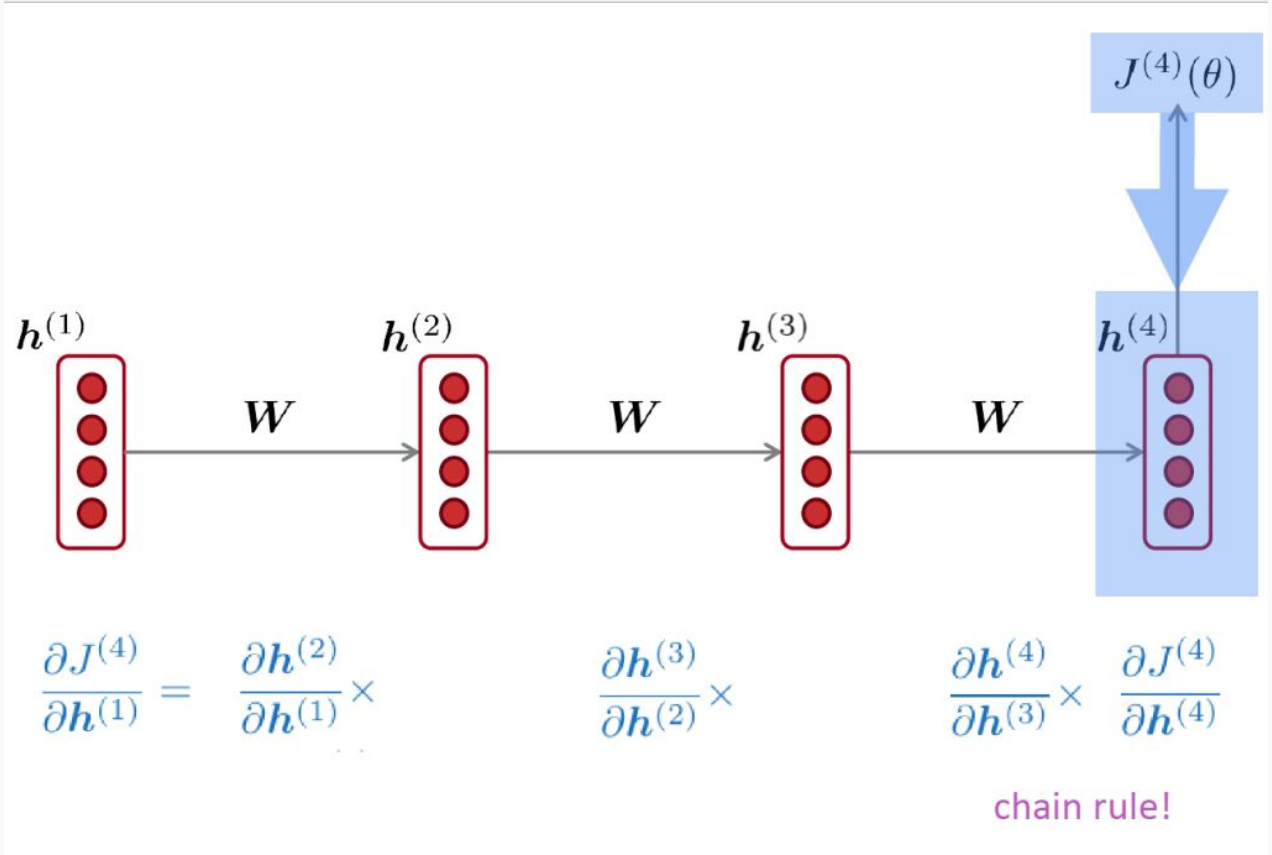
Van & exp gradients: The problem



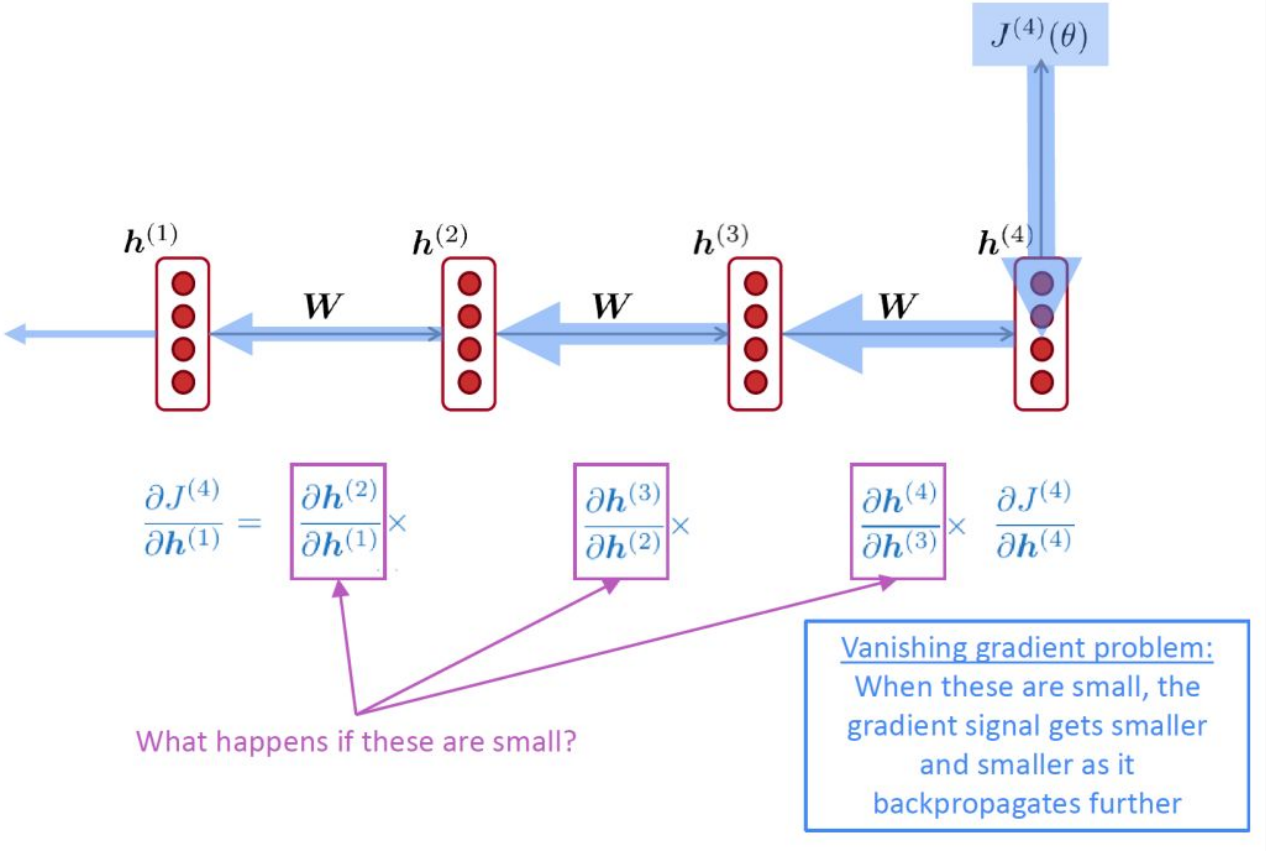
Van & exp gradients: The problem



Van & exp gradients: The problem



Van & exp gradients: The problem



Van & exp gradients: The problem

- If we calculate the gradient of J at step 'i' (i.e. $J^{(i)}(\theta)$) with respect to the hidden state that the RNN had in step 'j' ($\mathbf{h}^{(j)}$):

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}$$

we apply the chain rule many times

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)$$

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} = \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right) \boldsymbol{W}_h$$

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{(i-j)} \prod_{j<t\leq i} \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right)$$

If W_h is small, then this term gets vanishingly small as *i* and *j* get further apart

Multiplying a number by itself many times is very unstable[arXiv:1211.5063v2]

Van & exp gradients: The problem

If the largest eigenvalue of $\mathbf{W_h}$ is less than 1, the gradient $J^{(i)}(\theta)$ will decrease to 0 exponentially → Vanishing gradient

If the largest eigenvalue of $\mathbf{W_h}$ is larger than 1, then the gradient $J^{(i)}(\theta)$ will grow to ±∞ exponentially → Exploding gradient

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}$$

we apply the chain rule many times

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)$$

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} = \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right) \boldsymbol{W}_h$$

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{(i-j)} \prod_{j<t\leq i} \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right)$$

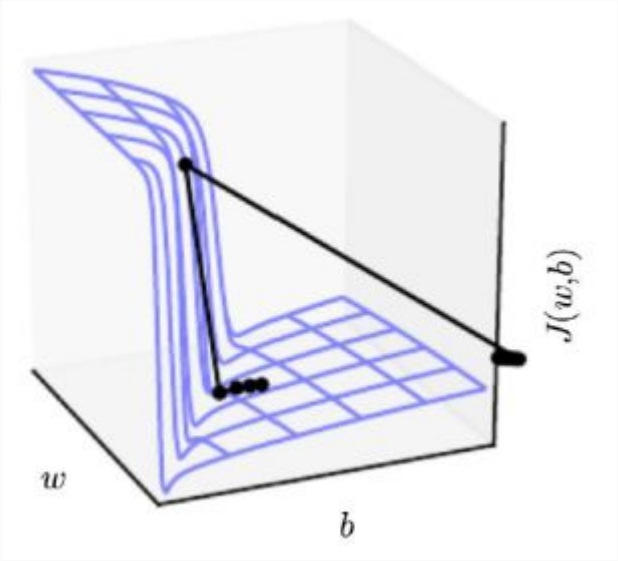
If W_h is small, then this term gets vanishingly small as *i* and *j* get further apart

Exploding gradients: The problem

- **Problem:** If the gradient ($\nabla_\theta J(\theta)$) becomes too big, updates grow too large and throw the model out of the basin of attraction of good minima in the error surface defined by $J(\theta)$ (i.e. in the parameter space). This happens when you apply the weights update rule

$$\theta^{new} = \theta^{old} - \underbrace{\alpha}_{\text{learning rate}} \underbrace{\nabla_\theta J(\theta)}_{\text{gradient}}$$

In extreme cases, this may even trigger register overflow and create NaNs or Infs

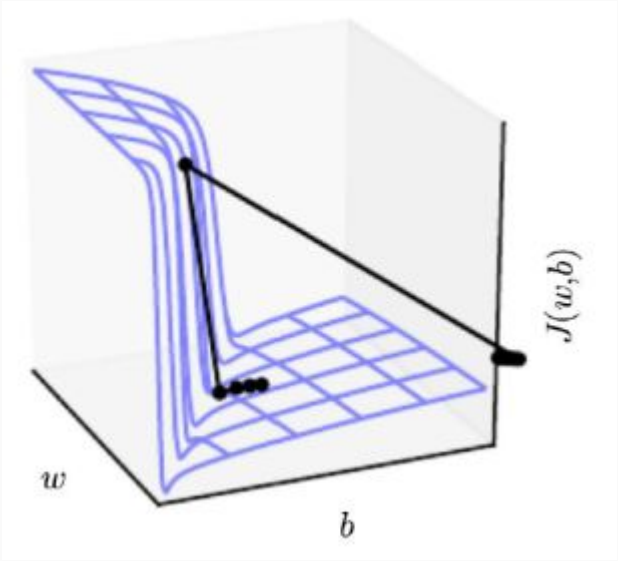


Exploding gradients: Solution

- **Idea:** Simply select a threshold (th), and cut any gradient (grad) whose absolute value is becoming too large

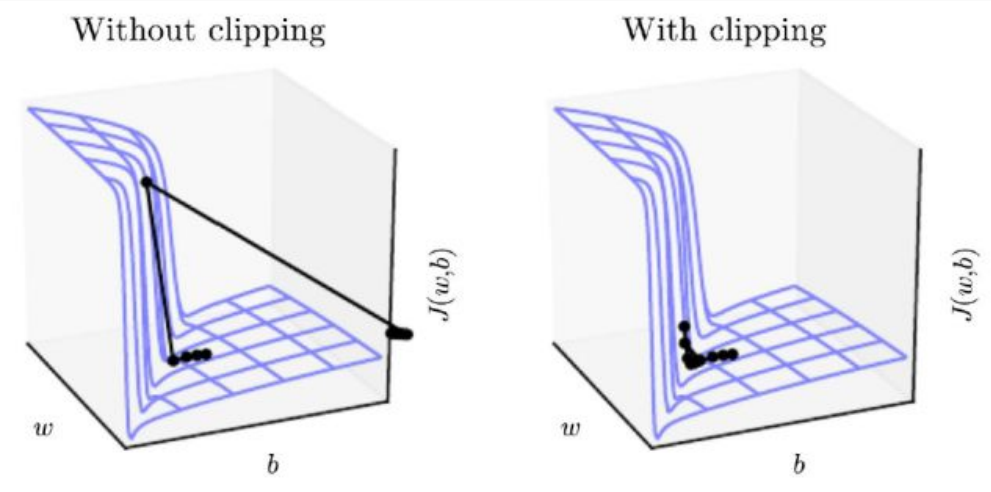
```
grad ← calculate_gradient( nn )
if norm( grad ) > th :
        grad ← th × ( grad / norm( grad ) )
nn.weight ← nn.weight + lr × grad
```

- Thanks to cliping, the update rule will apply a smaller update to the weights of the NN, but this update will still have the same sign



"On the difficulty of training recurrent neural networks", Pascanu et al., 2013.
http://proceedings.mlr.press/v28/pascanu13.pdf

Exploding gradients: Solution



- The figure shows the error surface of a RNN. This is the graphical representation of $J(\theta)$
- The "cliff" has a high gradient ($\nabla_\theta J(\theta) >> 0$). When the weights of the NN (w and b in the figure) falls there, the update rule '$\theta_{new} = \theta_{old} - \alpha \nabla_\theta J(\theta)$' throws the weights far away
- Gradient clipping avoids this by reducing the value of $\nabla_\theta J(\theta)$ in the update rule

"Deep Learning", Goodfellow, Bengio, and Courville, 2016. Chapter 10.11.1.
https://www.deeplearningbook.org/contents/rnn.html

Van & exp gradients: The problem

If the largest eigenvalue of $\mathbf{W_h}$ is less than 1, the gradient will decrease to 0 exponentially → Vanishing gradient

If the largest eigenvalue of $\mathbf{W_h}$ is larger than 1, then the gradient will grow to ±∞ exponentially → Exploding gradient

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}$$

we apply the chain rule many times

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)$$

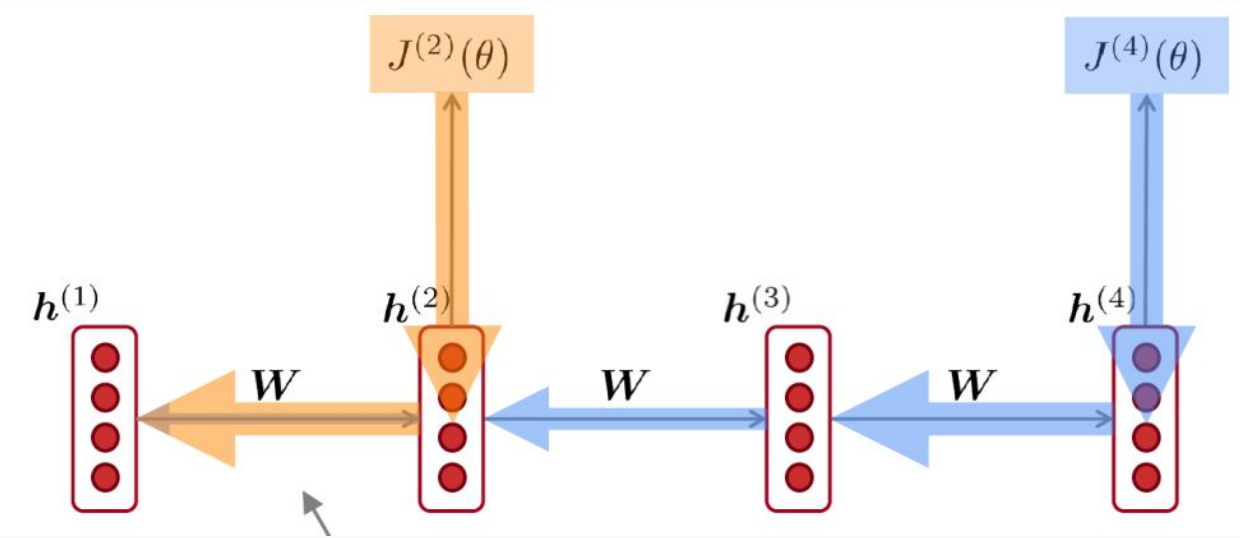
$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} = \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right) \boldsymbol{W}_h$$

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{(i-j)} \prod_{j<t\leq i} \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right)$$

If W_h is small, then this term gets vanishingly small as *i* and *j* get further apart

Vanishing gradients: The problem

- Far away gradients easily become smaller than gradients from nearby gradients
- The RNN ends up learning only short time effects, not long time effects



Vanishing gradients: The problem

- **Example:** *When she tried to print her tickets, she found that the printer had no more tonner. She went to the stationery shop to buy more. It was overpriced, but she had no more time to try other shops. After installing the toner into the printer, she finally printed her _____*
- To learn from this example (if we are using the example during training), or to correctly guess the missing word (if we are using the example during testing to measure the accuracy of the RNN), the NN needs to learn the dependency existing between the 7^{th} step (tickets) and the last step (_____ should be tickets again)
- But the gradient vanishes when being backpropagated through so many steps, so the RNN will never be able to learn this dependency

Vanishing gradients: The problem

- **Example:** *The food of the cats _____* ← is / are
- There are two ways of using the verb "to be" here:
  - By syntactic recency: use the form of the verb that refers to the closes noun or pronoun and makes the sense syntactically correct → is (correct in this example)
  - Sequential recency: use the form of the verb that refers to the closes noun or pronoun → are (incorrect in this example)
- Due to vanishing gradients, RNNs will tend to assign sequential recency rather than syntactic recency[arXiv:1611.01368v1], even in examples where this is wrong.

Vanishing gradients: The problem

- **Problem (backpropagation's take):** From the point of view of backpropagation, the problem is that, when you apply the chain rule to $\nabla_w J(W_h)$ (with W_h being the recurrent connection), you get an expression with the terms $W_h^{(i-j)}$, which tends to 0 for large 'i-j'

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}$$

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{(i-j)} \prod_{j<t\leq i} \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right)$$

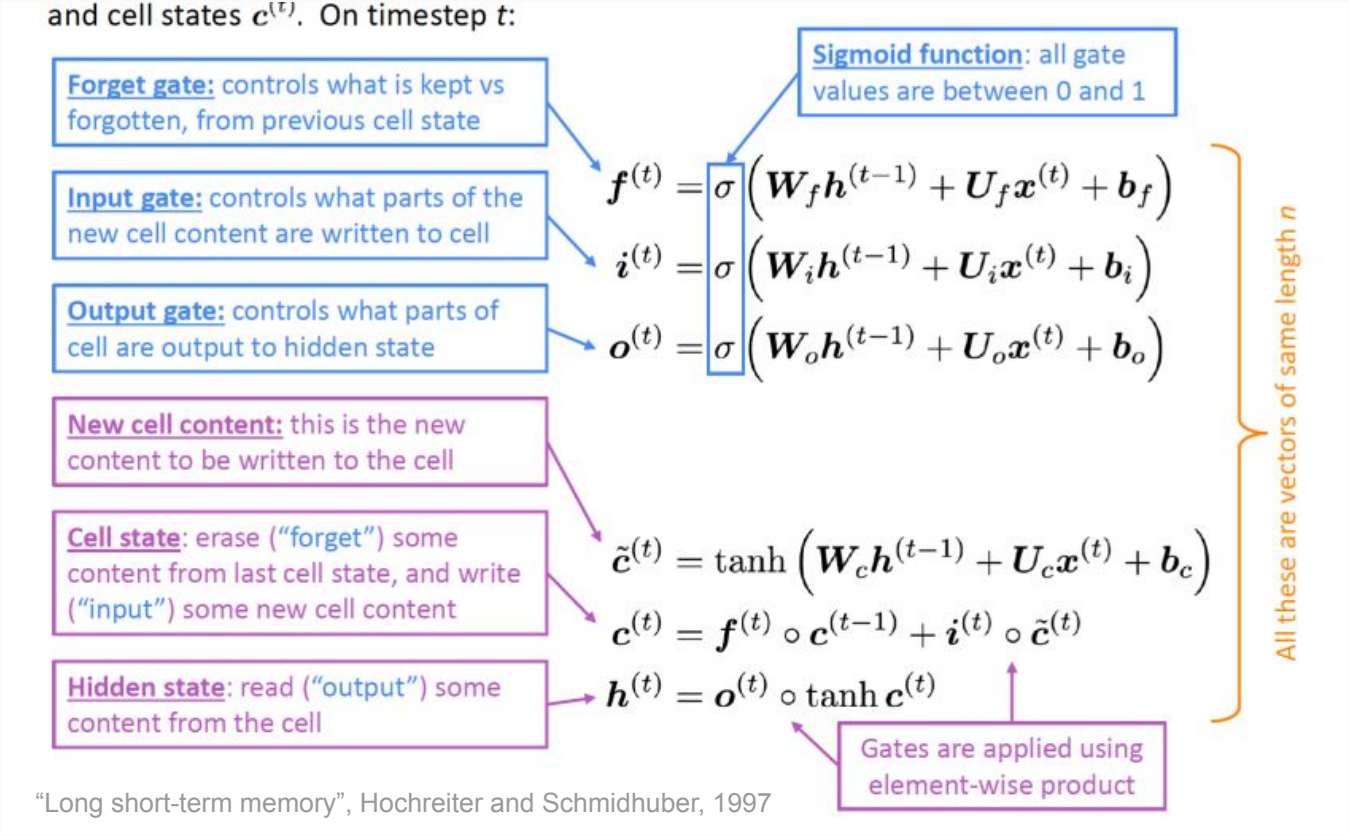
- **Problem (feedforwards' take):** From the point of view of the feed-forwards step, the problem is that the hidden state $h^{(t)}$ is constantly being re-written

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}\right)$$

Vanishing gradients: The solution

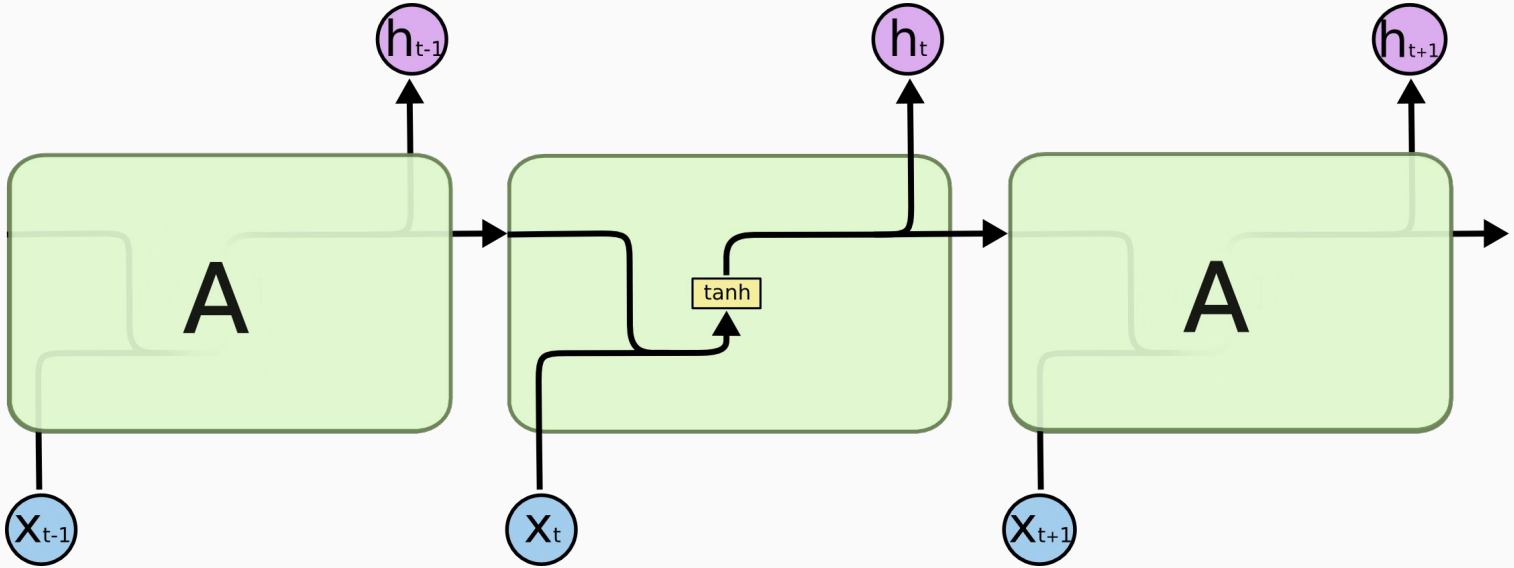
Idea: Besides the hidden state $h^{(t)}$, why do not introduce an extra 'super hidden' state where it is more difficult to write? A hidden state where the NN will write only when he/she are really really sure that he/she want to write something. This is a memory cell $c^{(t)}$.

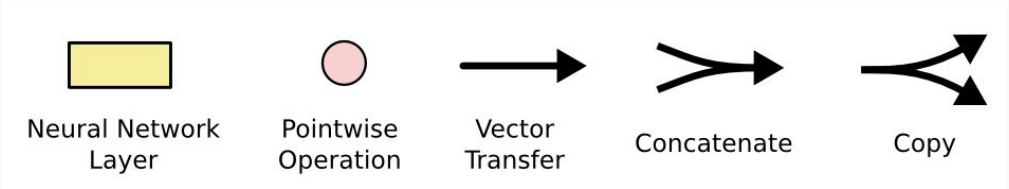
[10.1162/neco.1997.9.8.1735]



"Long short-term memory", Hochreiter and Schmidhuber, 1997

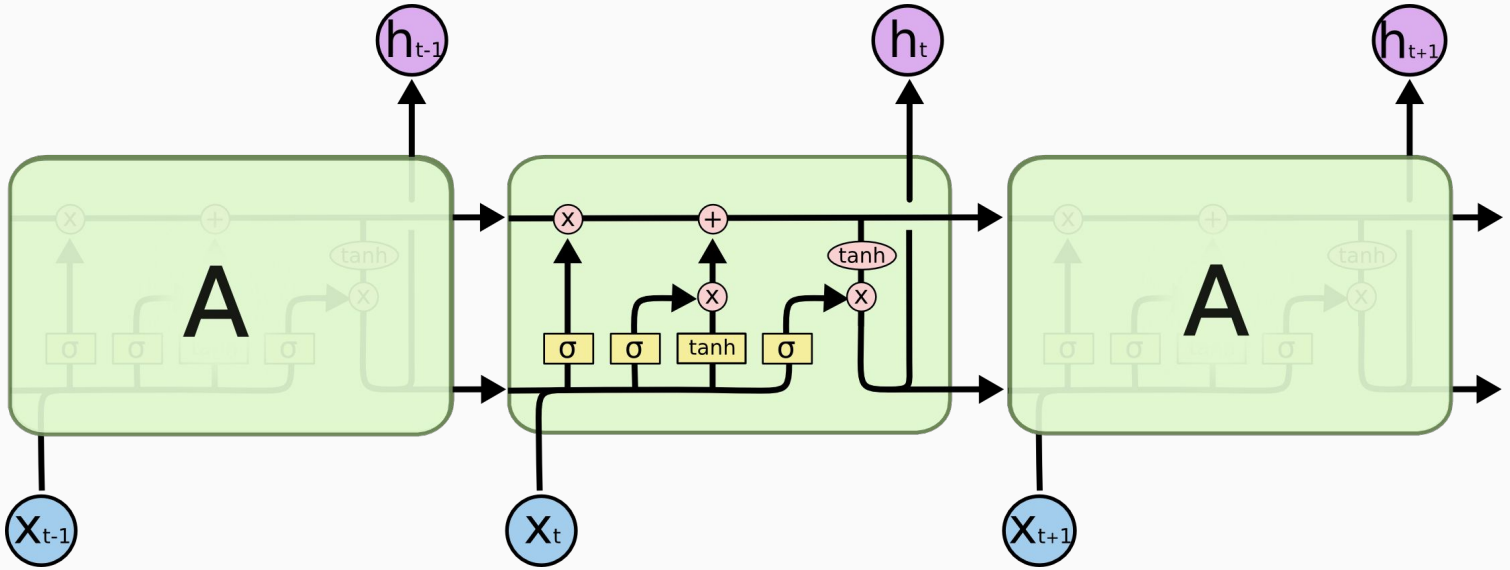
Vanishing gradients: The solution

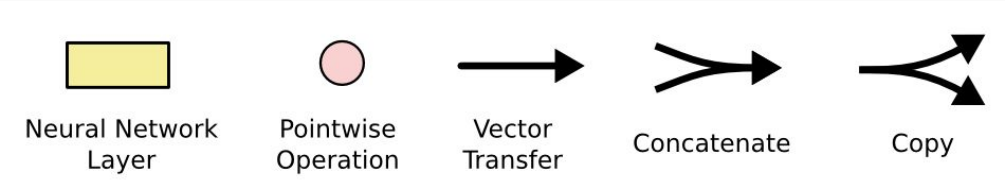




http://colah.github.io/posts/2015-08-Understanding-LSTMs/

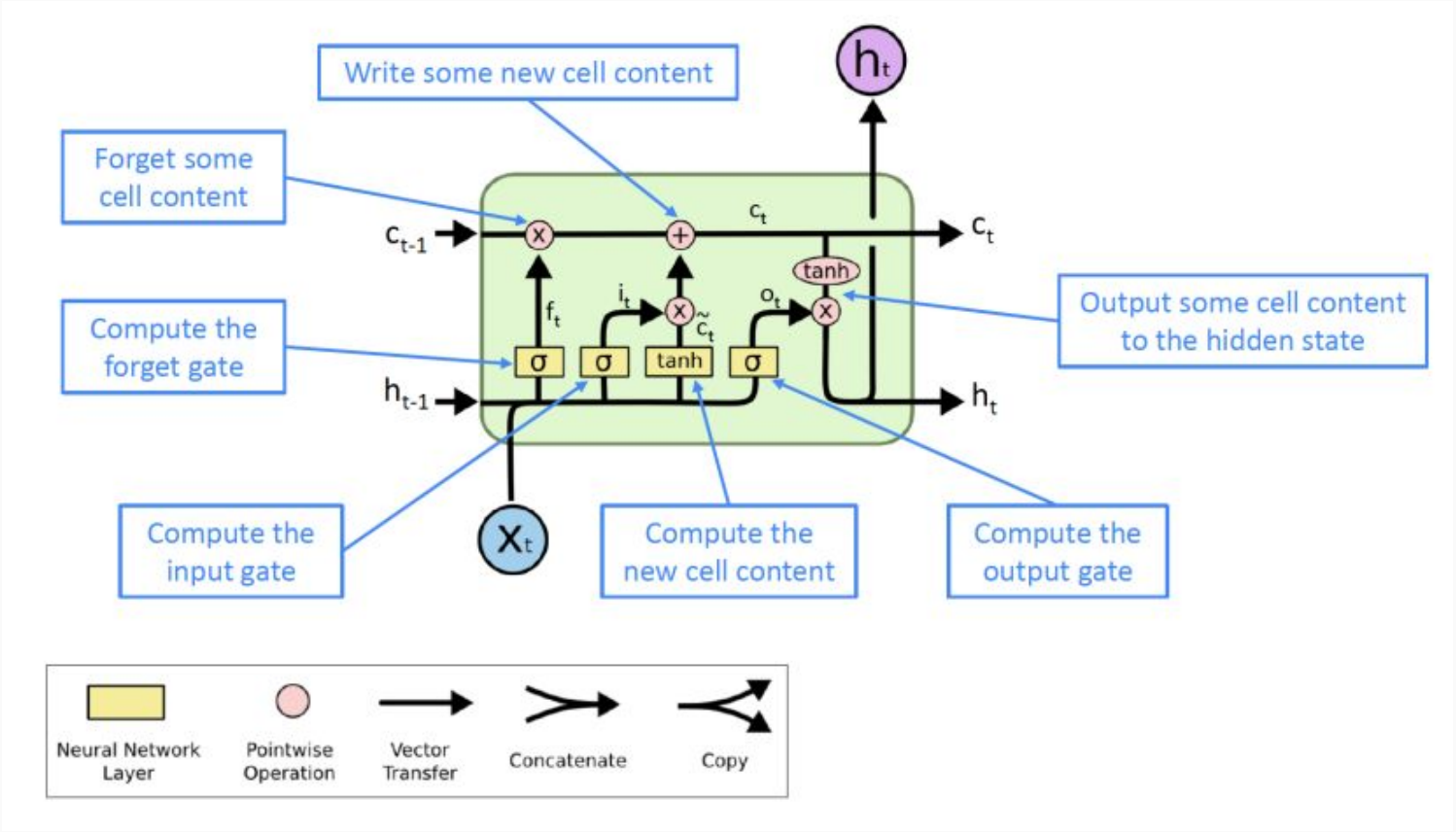
Vanishing gradients: The solution





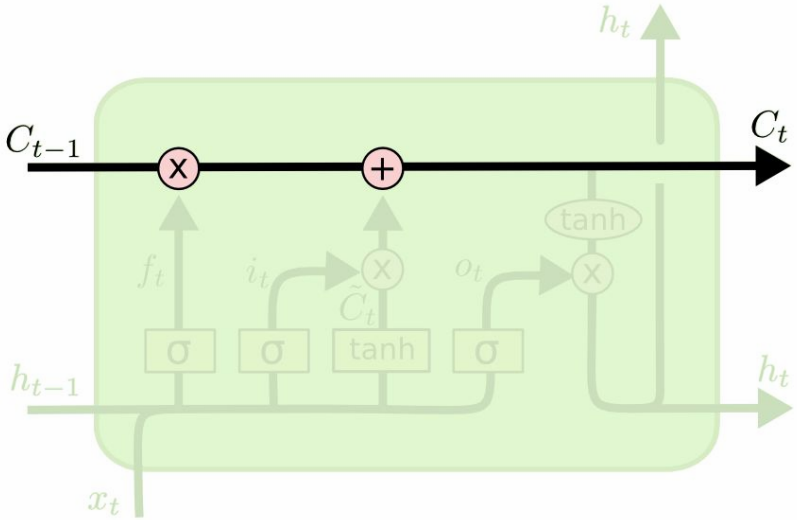
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Vanishing gradients: The solution



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

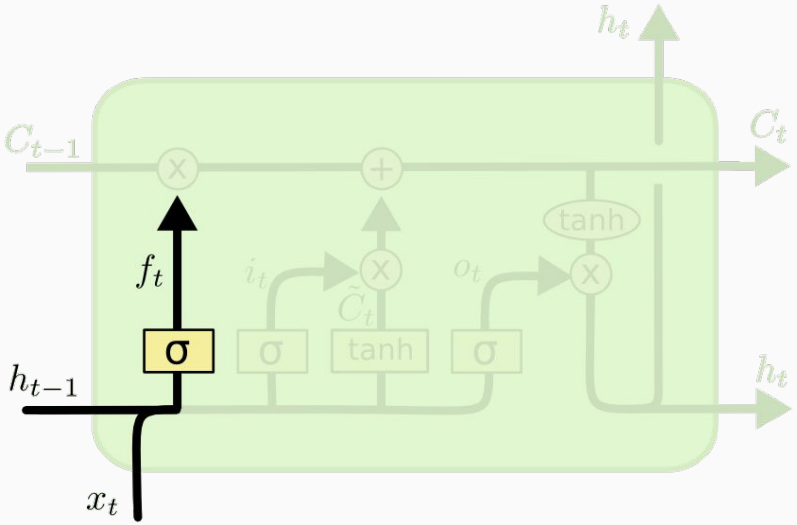
Vanishing gradients: The solution



- The ‘super hidden state’ $c^{(t)}$ keeps information from one recurrent step to the next, very much like the standard hidden state $h^{(t)}$. The difference is that for the NN it is more difficult to change its value - $c^{(t)}$ is more stable than $h^{(t)}$.

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

Vanishing gradients: The solution

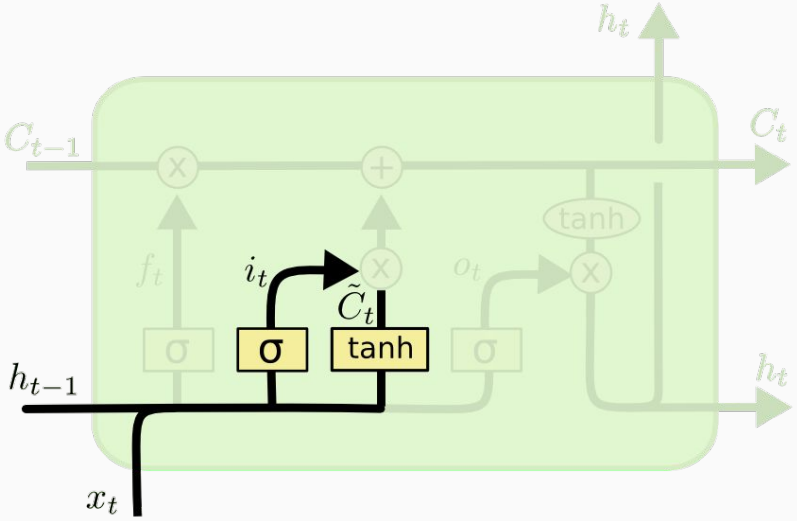


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

- The forget gate $f^{(t)}$ decides when to erase information from $c^{(t)}$
- The forget date looks at $h^{(t-1)}$ and $x^{(t)}$, outputs a value between 0 (forget) and 1 (remember), and multiplies this with $c^{(t-1)}$.

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

Vanishing gradients: The solution



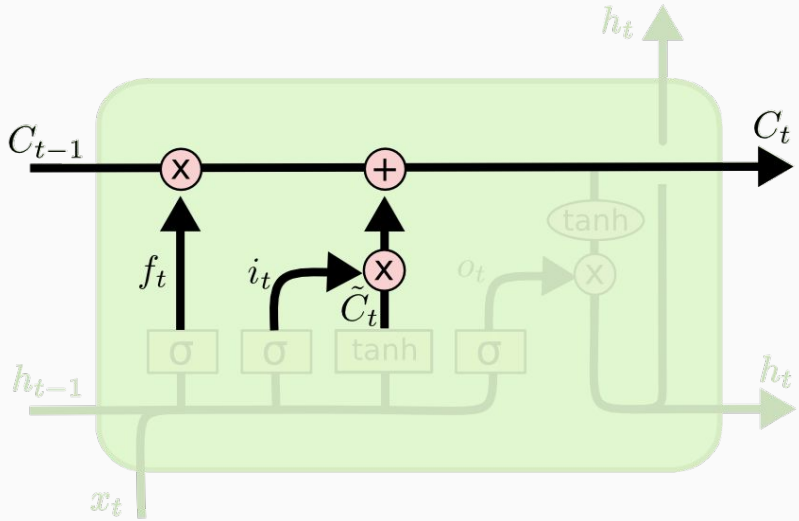
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

- The input gate $i^{(t)}$ decides when and what new information to introduce in $c^{(t)}$
- The forget gate looks at $h^{(t-1)}$ and x^t two times. A first time with a σ activation function to decide 'when' to input new information, a second time with a tanh activation function to decide 'what' information to input. … [next slide]

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

Vanishing gradients: The solution

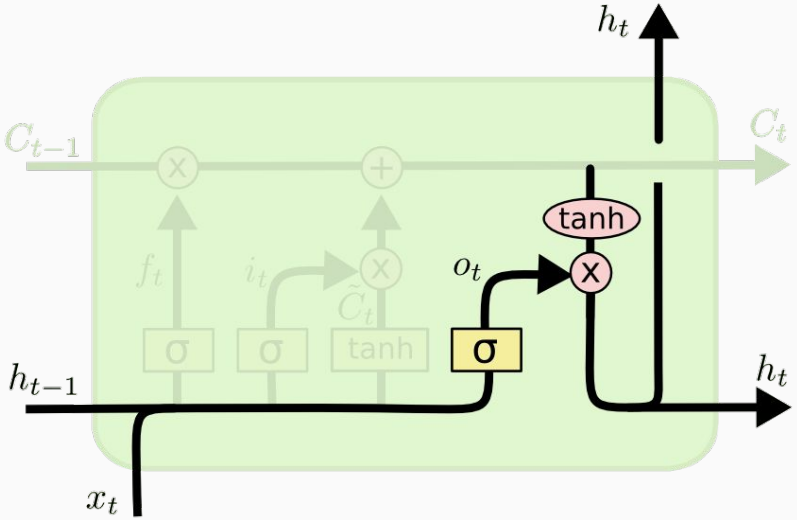


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- The input gate $i^{(t)}$ decides when and what new information to introduce in $c^{(t)}$
- [continuation] … Then the forget gate multiplies the outputs of σ and tanh to decide when AND what information to input. Then inputs that information to the cell $c^{(t)}$

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

Vanishing gradients: The solution



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

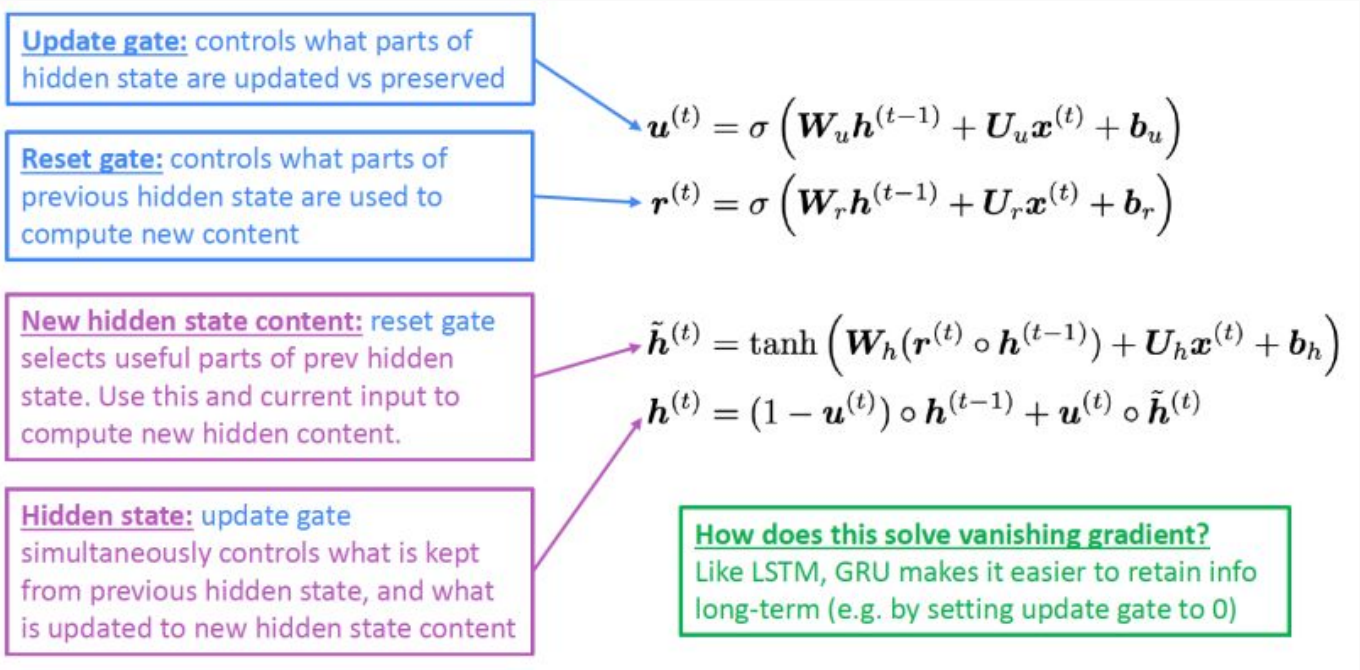
$$h_t = o_t * \tanh\left(C_t\right)$$

- The output gate $o^{(t)}$ decides when and what information to output in $h^{(t)}$
- The output date looks at $h^{(t-1)}$ and x^t two times to decide 'when' and 'what' to output. Exactly in the same way as the input gate did, but this time to decide the next value of $h^{(t)}$, not what to add to the current value of $c^{(t)}$

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

Vanishing gradients: The solution

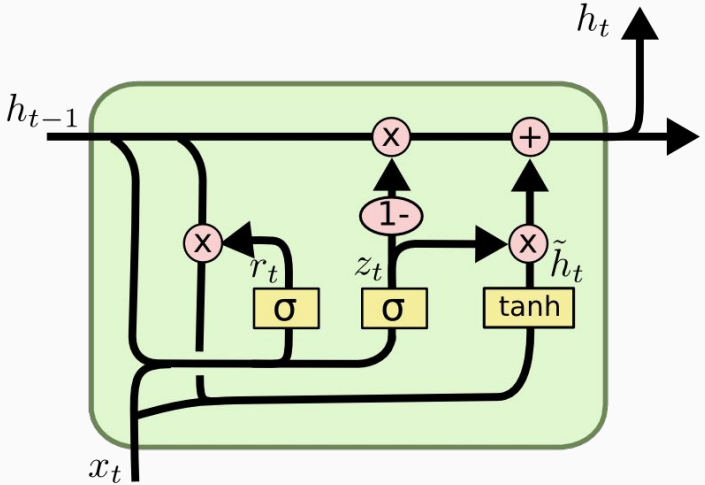
Idea: Why don't we simplify the architecture of the LSTM? We could merge the forget and input gates, because one of these gates is simply doing the opposite of the other. We could also merge $c^{(t)}$ and $h^{(t)}$, because they are simply doing slightly different things[arXiv:1406.1078v3]



"Learning phrase representations using RNN encoder-decoder for statistical machine translation", Cho et al., 2014.

Vanishing gradients: The solution

- Main simplifications:
  - Merge forget $f^{(t)}$ and input $i^{(t)}$ gates into a single gate → update gate $z^{(t)}$
  - Merge the hidden state $h^{(t)}$ and memory cell $c^{(t)}$ → new single $h^{(t)}$
  - Some other tweaks



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

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

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

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

Improved RNNs: LSTM, GRU

- Thanks to the memory cell $c^{(t)}$ (the 'supper hidden' state), the LSTM architecture makes it easier for the RNN to preserver information over many time steps.
  - e.g. if the forget gate is set to remember everything on every timestep, then the information in the cell is preserved indefinitely
  - e.g. by contracts, it is harder for vanilla RNN to learn a recurrent weight matrix W_h that preserves information for long in a hidden state

- LSTM & GRU however do not fully guarantee that there is not vanishing/exploding gradients, but it does provide an easier way for the model to learn long-distance dependencies

Improved RNNs: LSTM, GRU

- Researchers have proposed many gated RNNs variants, but LSTM and GRU are the most widely used
- The biggest difference is that GRU is quicker to computer, and has fewer parameters
- There is no conclusive evidence that one consistently outperforms the other
- LSTM is a good default choice (specially if your data has particularly long dependencies, or you have lots of training data)
- **Rule of thumb**: start with LSTM, but switch to GRU is you want something more efficient

Vanishing grads: The problem outside NLP

➢ **Problem:** The equation suggesting vanishing-exploding gradients is true for all NNs.

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{(i-j)} \prod_{j<t\leq i} \text{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right)$$

↳backpropagation's take

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}\right)$$

↳ feedforwards's take

In non recurrent NNs is that the first layers (closest to input) learn very slowly

➢ **Solution:** Changes in architecture

- Residual connections (**ResNets**), also known as skip-connections or peepholes: Allow feedforwarded signals and backpropagated gradients to skip layers. Memory cells in RNNs do kind of the same trick, but across time steps rather than across layers
- Batch Normalisation: Get neuron's inputs away from the domain where activation functions have low gradients.

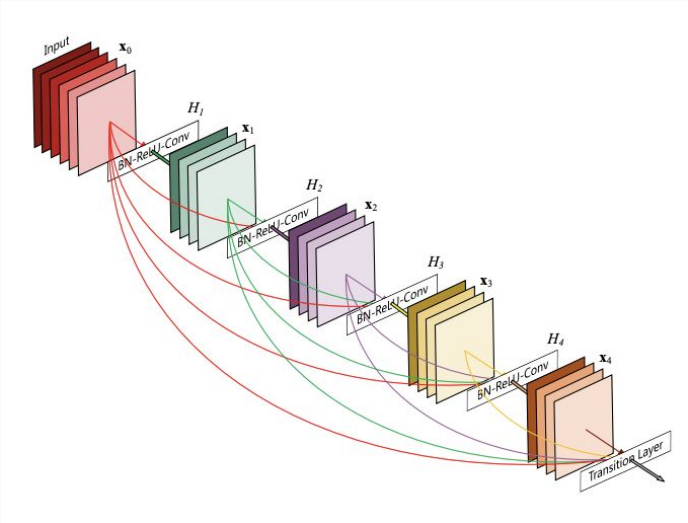
Vanishing grads: The solution outside NLP

➢ **Solution:** Changes in architecture

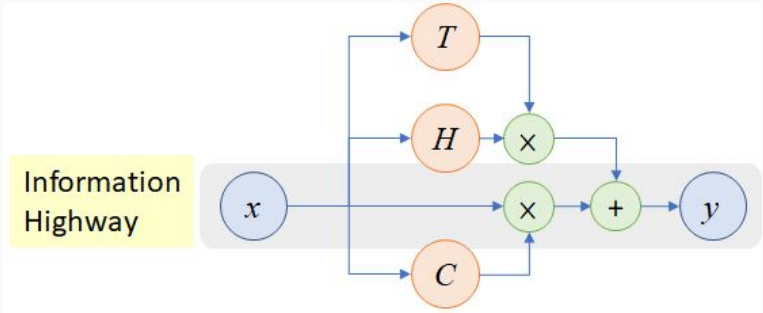
- … [more stuff here]
- Residual connections (**ResNets**), also known as skip-connections or peepholes: Allow feedforwarded signals and backpropagated gradients to skip layers
- Batch Normalisation: Get neuron's inputs away from the domain where activation functions have low gradients.
- Careful initialisation: Initial weights promote inputs away from low gradient domain (Glorot, He, Xavier)
- Better activation functions: In older activation functions the gradient is close to 0 in most of the domain (sigmoida, tanh). Newer activation functions avoid gradients close to 0 (ReLu, leaky ReLy, ELU)
- Gradient clipping: Simply cap gradients that are getting too high (also in RNNs)

Vanishing grads: The solution outside NLP

- Dense skip connections (**DenseNet**): Make skip connections from every layer to every other layer[arXiv:1608.06993v5]



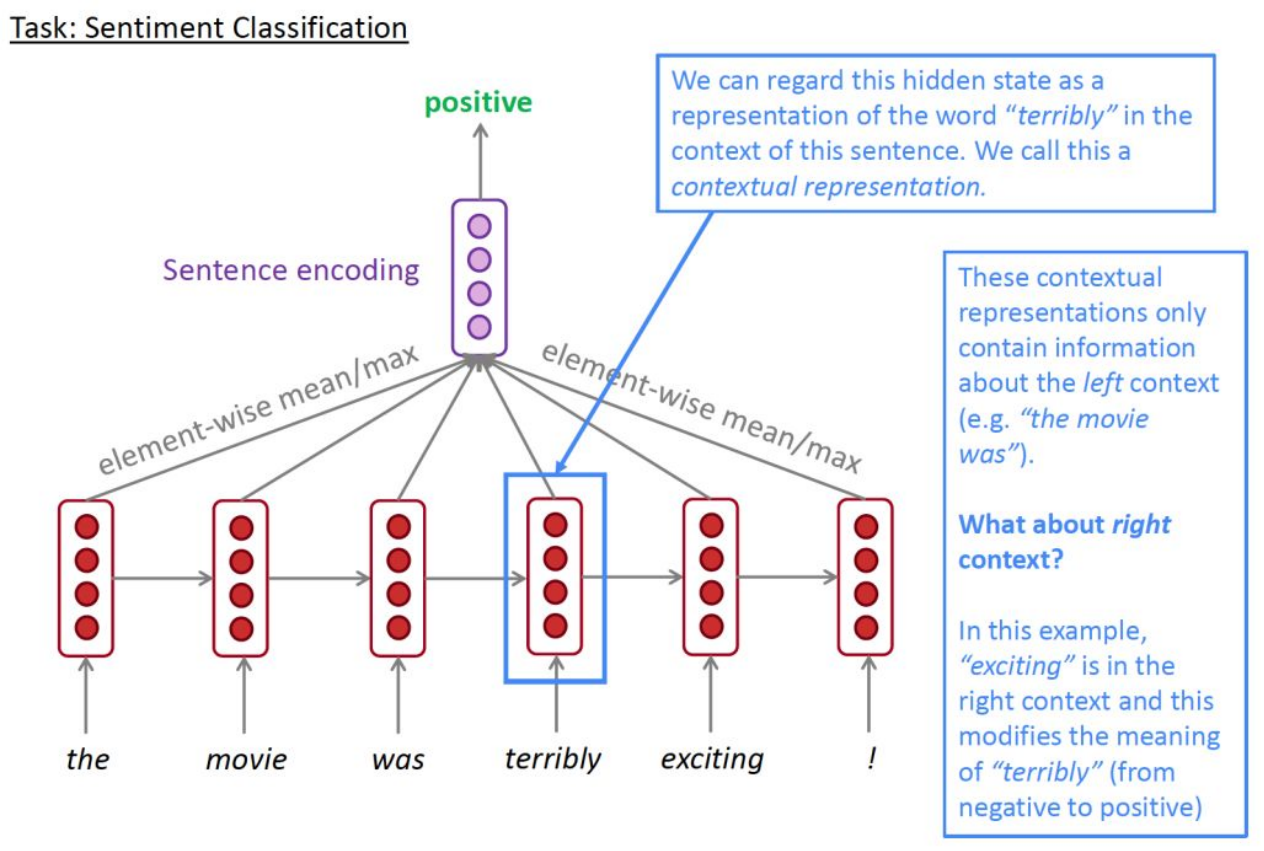
- Highway connections (**HighwayNet**): Make gated skip connections[arXiv:1505.00387v2]. Similar to the forget $f^{(t)}$, input $i^{(t)}$ and output $o^{(t)}$ gates of memory cells in LSTM (or update $z^{(t)}$ and reset $r^{(t)}$ in GRU)



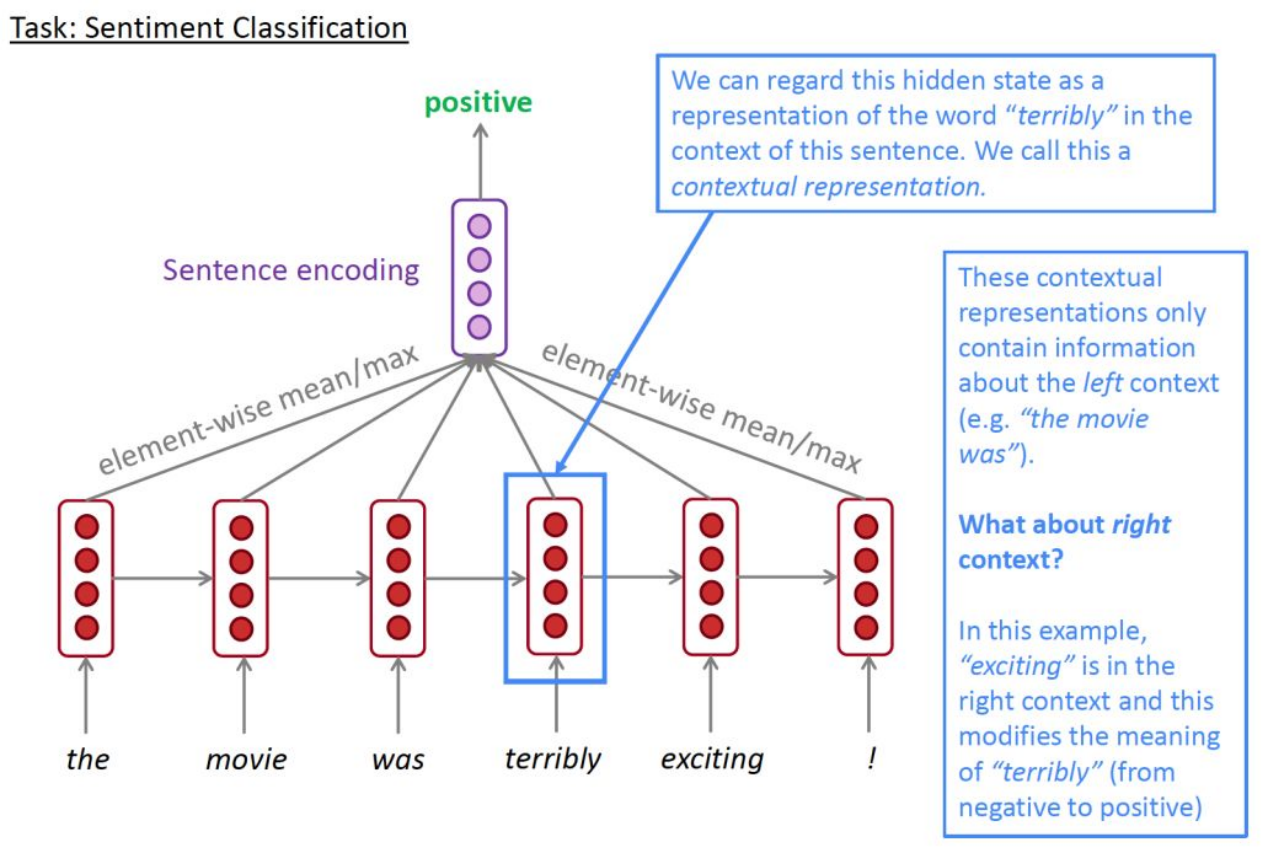
Bidirectional RNNs

Idea: Correlations across time steps do not only occur from previous steps $h^{(t-k)}$ to current step $h^{(t)}$ - they also occur from future steps $h^{(t+k)}$ to current one. Why don't we feed into the current step information from future ones?

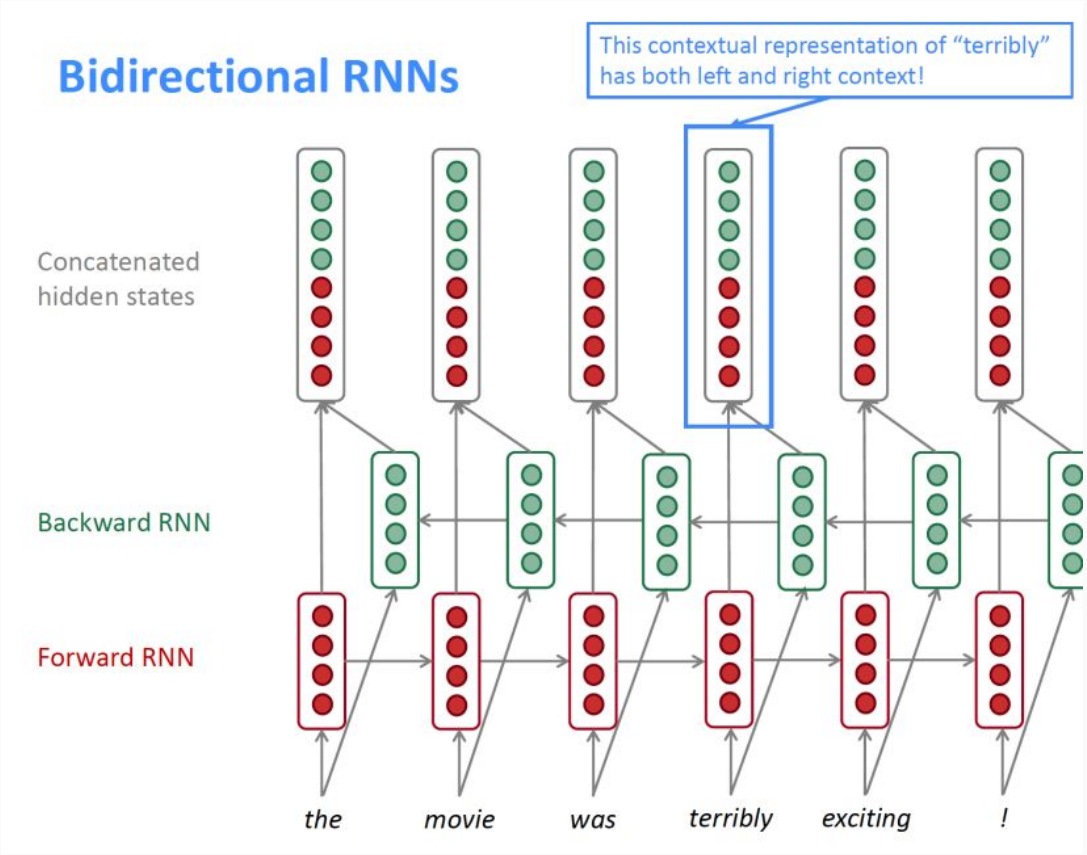
[10.1162/neco.1997.9.8.1735]



Bidirectional RNNs



Bidirectional RNNs



Bidirectional RNNs

On timestep t:

This is a general notation to mean "compute one forward step of the RNN" – it could be a vanilla, LSTM or GRU computation.

Forward RNN $\overrightarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{FW}}(\overrightarrow{\boldsymbol{h}}^{(t-1)}, \boldsymbol{x}^{(t)})$

Backward RNN $\overleftarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{BW}}(\overleftarrow{\boldsymbol{h}}^{(t+1)}, \boldsymbol{x}^{(t)})$

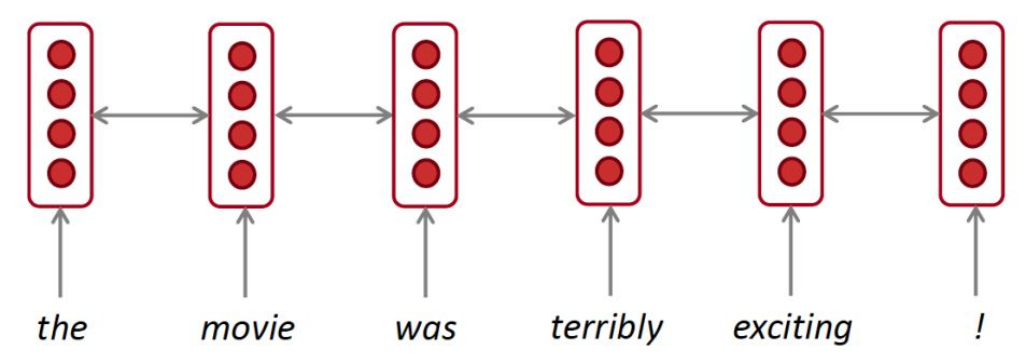
Generally, these two RNNs have separate weights

Concatenated hidden states $\boldsymbol{h}^{(t)} = [\overrightarrow{\boldsymbol{h}}^{(t)}; \overleftarrow{\boldsymbol{h}}^{(t)}]$

We regard this as "the hidden state" of a bidirectional RNN. This is what we pass on to the next parts of the network.

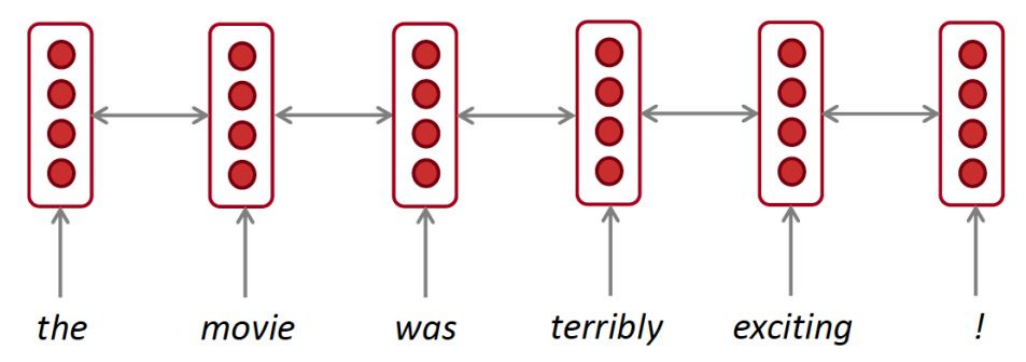
Bidirectional RNNs

- Bidirectional RNNs are virtually always better than monodirectional RNNs. You should use them by default if they are applicable to your problem
- They are sometimes not applicable when you don't have access to the full sequence - e.g. online-learning
- **BERT**: Bidirectional Encoder Representations from Transformer. A very powerful bidirectional RNNs based on Transformer architecture. Very standard now[arXiv:1810.04805v2]

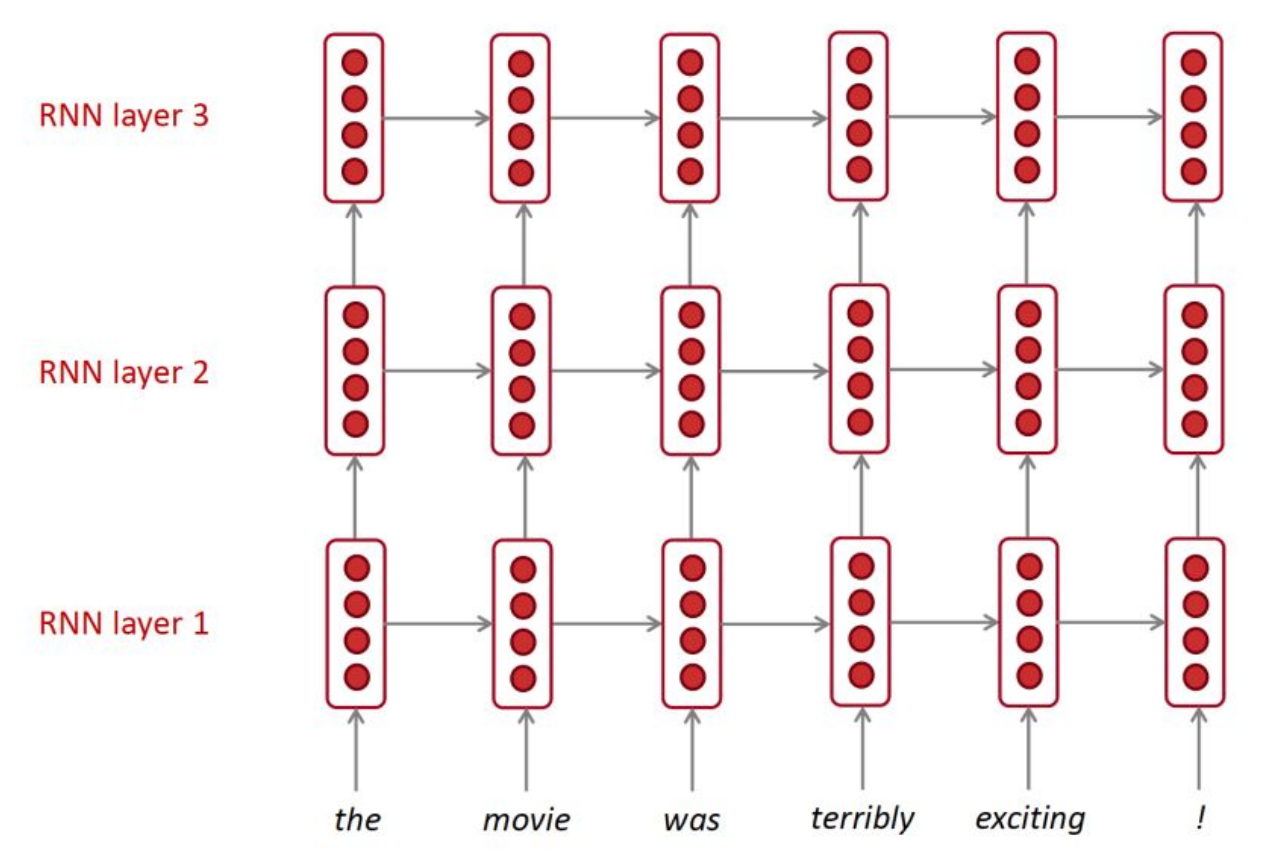


Multilayer RNNs

- **Idea:** A simple RNNs is already deep in the time dimension. Why don't we make them deep in another dimension by stacking them, like in feed forward NNs (FNNs)? We can simply feed the hidden state $h^{(t)}$ from one layer as the input $x^{(t)}$ of the next
- These are called multi-layer NNs:
- As in FNNs, they can represent more complex input-output relationships. Lower layers computer lower-level features, while higher layers compute more complex ones

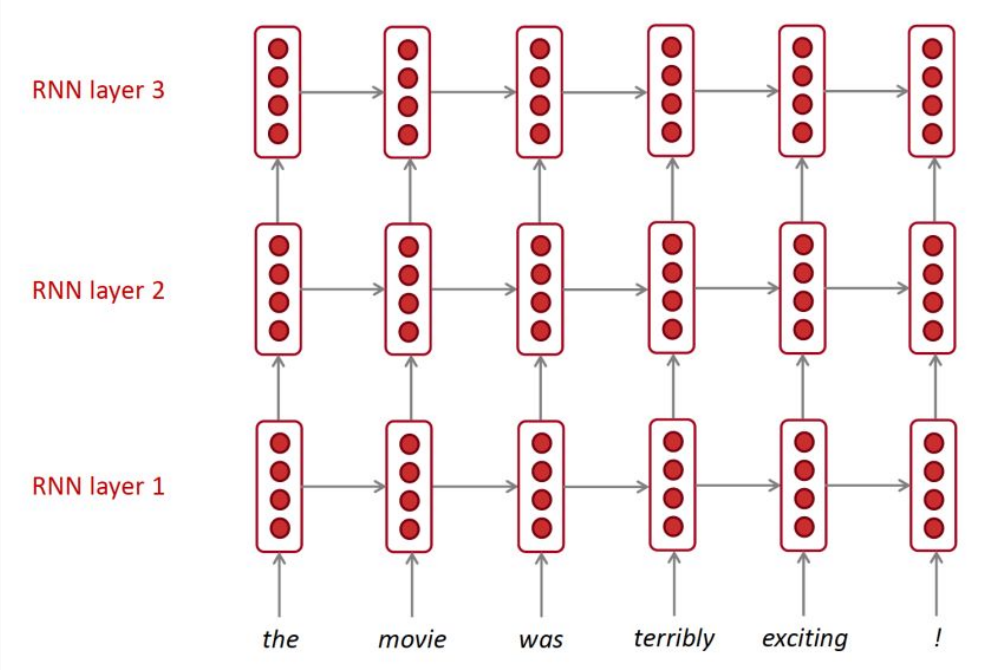


Multilayer RNNs



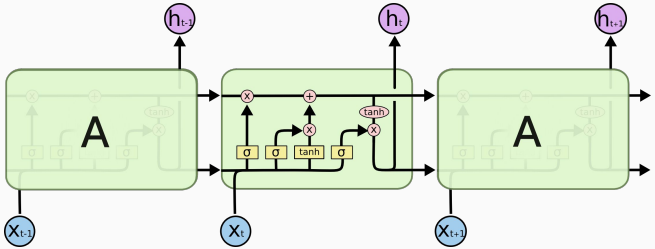
Bidirectional RNNs

- State of the art RNNs are often bidirectional and multi-layer. They are however not as deep as FNNs or convolutional NNs (CNNs), often used used in vision
- Non transformed RNNs seem to perform best with only a few layers. For the encoder part of a RNN this seems to be 2 to 4 layers, for the decoder part 4. Deeper RNNs (e.g. 8) need skip connections
- Transformer RNNs (e.g. **BERT**) can have up to 24 layers.

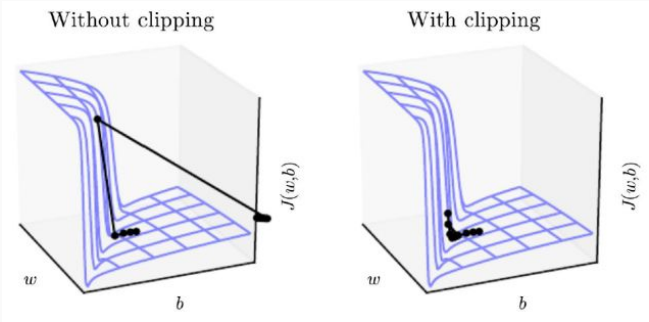


Bidirectional RNNs

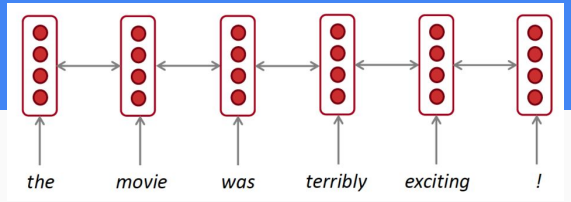
Lots of information today. Take away messages:



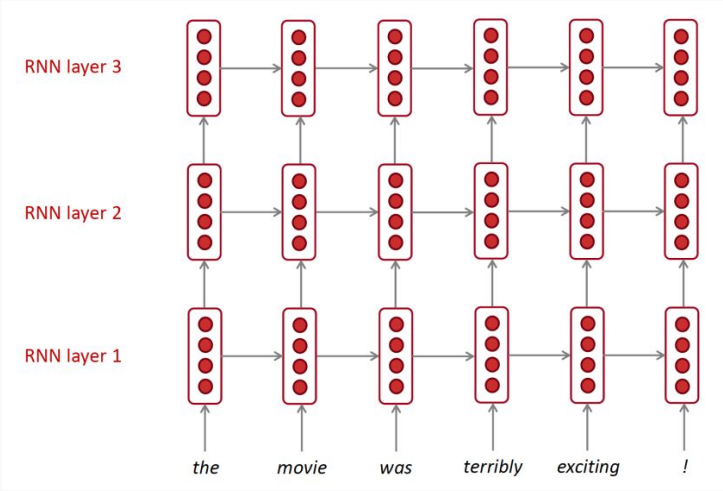
1) LSTMs are powerful but GRUs are faster



2) Clip your gradients



3) Use bidirectional connections when possible



4) Multilayer RNNs are powerful, but you will need skip-connections if deep

Literature

- **Papers** =
  - Sections 10.3, 10.5, and 10.7-10.12 of \Deep Learning", Goodfellow et al., 2016. http://www.deeplearningbook.org/contents/rnn.html
  - “Learning long-term dependencies with gradient descent is difficult", Bengio et al., 1994. http://ai.dinfo.unifi.it/paolo//ps/tnn-94-gradient.pdf
  - “On the difficulty of training recurrent neural networks", Pascanu et al., 2013. https://arxiv.org/pdf/1211.5063
  - “Understanding LSTM networks", Olah, 2015. http://colah.github.io/posts/2015-08-Understanding-LSTMs/