# Recurrent neural networks

Herman Kamper

2024-03, CC BY-SA 4.0

A fixed-window neural language model

A simple RNN language model

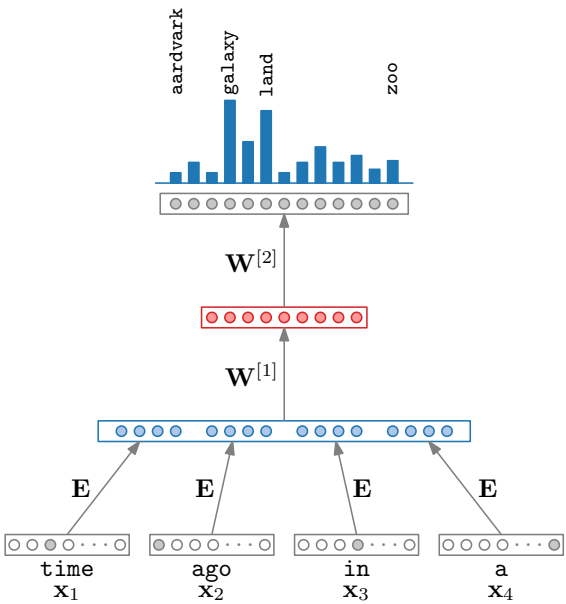
Recurrent neural networks (RNNs)

Backpropagation through time

Vanishing and exploding gradients

Extensions of RNNs

# A fixed-window neural language model




Output distribution

$$\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_{1:4}) = \hat{\mathbf{y}}$$
$$= \mathrm{softmax}(\mathbf{W}^{[2]}\mathbf{h} + \mathbf{b}^{[2]})$$
$$\in [0,1]^{|\mathcal{V}|}$$

Hidden layer

$$\mathbf{h} = g(\mathbf{W}^{[1]}\mathbf{e} + \mathbf{b}^{[1]})$$

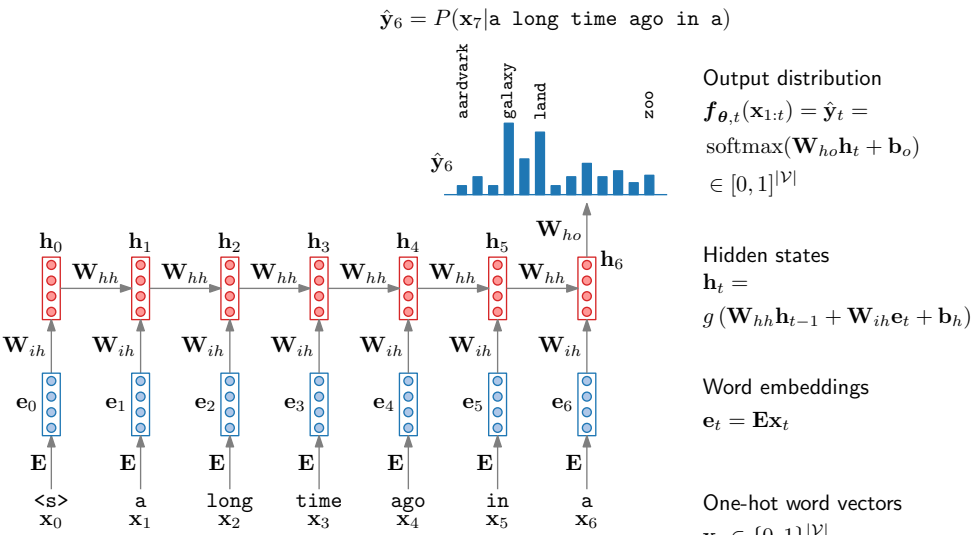
Concatenated word embeddings

$$\mathbf{e} = [\mathbf{e}_1; \mathbf{e}_2; \mathbf{e}_3; \mathbf{e}_4]$$

One-hot word vectors

What about a neural network that could process inputs of arbitrary length?

# A simple RNN language model




An RNN language model outputs a $\hat{\mathbf{y}}_t$ at every time step $t$:[1]

$$\hat{y}_{t,k} = P_{\boldsymbol{\theta}}(w_{t+1} = k|w_{1:t}) = P_{\boldsymbol{\theta}}(\mathbf{x}_{t+1,k} = 1|\mathbf{x}_{1:t})$$

[1] In the figure, $\mathbf{x}_t$ is the one-hot vector representing word $x_t$.

## Training loss

You can think of the output at $t$ as having its own loss: $J_t(\boldsymbol{\theta})$

The overall loss is the sum of the losses at every time step:

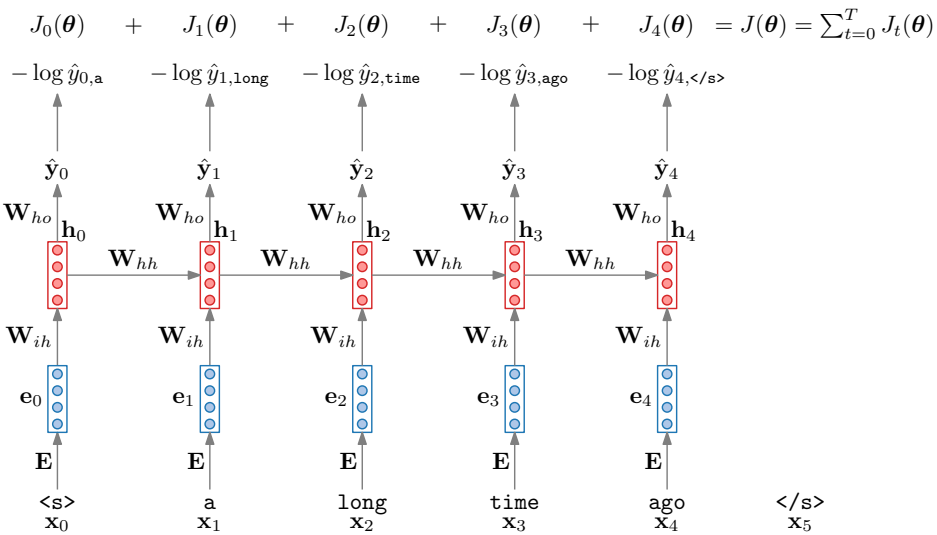
$$J(\boldsymbol{\theta}) = \sum_{t=0}^{T} J_t(\boldsymbol{\theta})$$

We use the negative log likelihood for $J(\boldsymbol{\theta})$:

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\sum_{t=0}^{T} \log P_{\boldsymbol{\theta}}(w_{t+1}|w_{1:t}) \\ &= -\sum_{t=0}^{T} \sum_{k=1}^{|\mathcal{V}|} x_{t+1,k} \log \hat{y}_{t,k} \\ &= -\sum_{t=0}^{T} \log \hat{y}_{t,j} = -\sum_{t=0}^{T} \log \hat{y}_{t,w_{t+1}} = \sum_{t=0}^{T} J_t(\boldsymbol{\theta}) \end{aligned}$$

where $j$ is the position where $\mathbf{x}_{t+1}$ is on and

$$J_t(\boldsymbol{\theta}) = -\log \hat{y}_{t,w_{t+1}}$$



Note that $J_t(\boldsymbol{\theta})$ is also the cross entropy between the one-hot $\mathbf{x}_{t+1}$ and $\hat{\mathbf{y}}_t$, if we think of these as discrete distributions:

$$\begin{aligned} H(\mathbf{x}_{t+1}, \hat{\mathbf{y}}_t) &= \sum_{k=1}^{|\mathcal{V}|} x_{t+1,k} \log \hat{y}_{t,k} \\ &= -\log \hat{y}_{t,j} = -\log \hat{y}_{t,w_{t+1}} \end{aligned}$$

In PyTorch you will probably use the cross entropy loss class.

In practice we normally get the overall loss by averaging over the per-time-step losses:[2]

$$\begin{aligned} J(\boldsymbol{\theta}) &= \frac{1}{T+1} \sum_{t=0}^{T} J_t(\boldsymbol{\theta}) \\ &= -\frac{1}{T+1} \sum_{t=0}^{T} \log \hat{y}_{t,w_{t+1}} \\ &= -\frac{1}{T+1} \sum_{t=0}^{T} \log P_{\boldsymbol{\theta}}(w_{t+1} | w_{1:t}) \end{aligned}$$

How does this averaged loss compare to the definition of perplexity given in the note on N-gram language modelling?

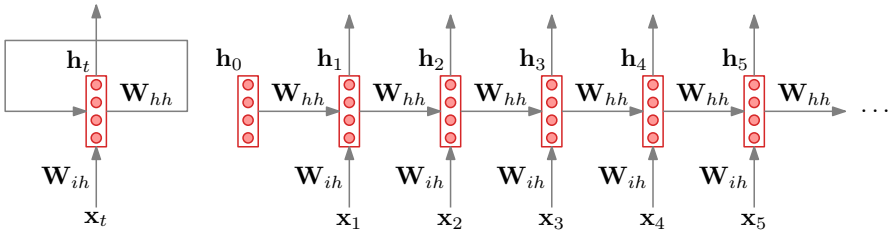
[2] I normalise by $T+1$ here since I explicitly include an end-of-sentence token at $T+1$. Some explanations do not do this, then we would normalise by $T$.

# Recurrent neural networks (RNNs)

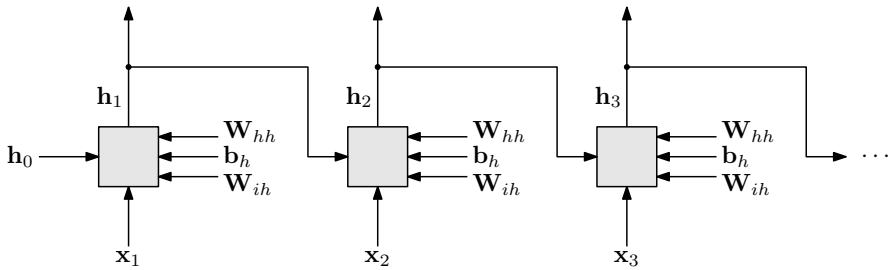
The core idea is to process each time step in the same way and influence the next time step through a hidden memory.

$$\mathbf{h}_t = g\left(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{ih}\mathbf{x}_t + \mathbf{b}_h\right)$$

Vector diagram:




Computational graph:

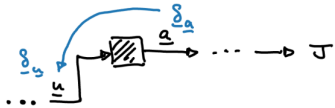



The shaded nodes represent the recurrent operation, as in the equation.

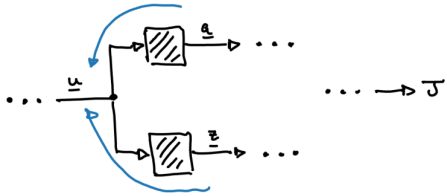
The same weights are applied at every time step:

- Apply $\mathbf{W}_{ih}$ to every input $\mathbf{x}_t$
- Apply $\mathbf{W}_{hh}$ to every hidden state $\mathbf{h}_{t-1}$ to affect the next hidden state $\mathbf{h}_t$

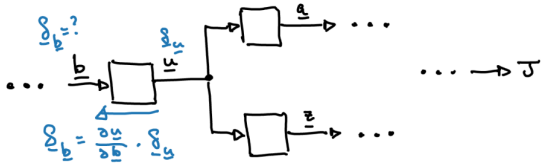
# BPTT preliminaries: A quick summary of backprop




$$\frac{\partial J}{\partial u} = \delta_u = \frac{\partial a}{\partial u} \cdot \underbrace{\frac{\partial J}{\partial a}}_{\delta_a} = \text{prod}\left(\frac{\partial a}{\partial u}, \delta_a\right)$$



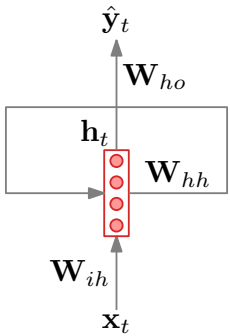

$$\delta_u = \text{prod}\left(\frac{\partial a}{\partial u}, \delta_a\right) + \text{prod}\left(\frac{\partial z}{\partial u}, \delta_z\right)$$



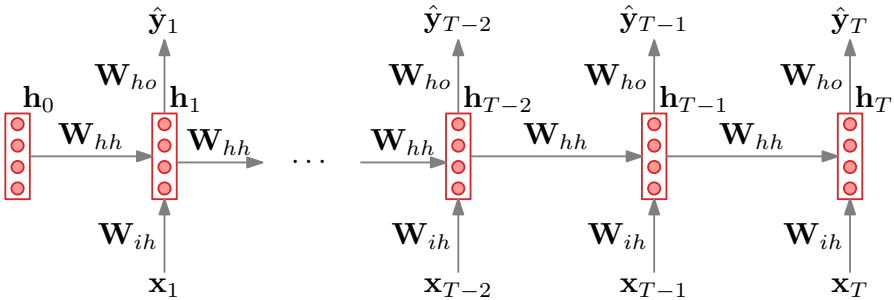

$$\delta_u \leftarrow 0$$
$$\delta_u \leftarrow \delta_u + \frac{\partial a}{\partial u} \cdot \delta_a$$
$$\delta_u \leftarrow \delta_u + \frac{\partial z}{\partial u} \cdot \delta_z$$

# Backpropagation through time (BPTT)

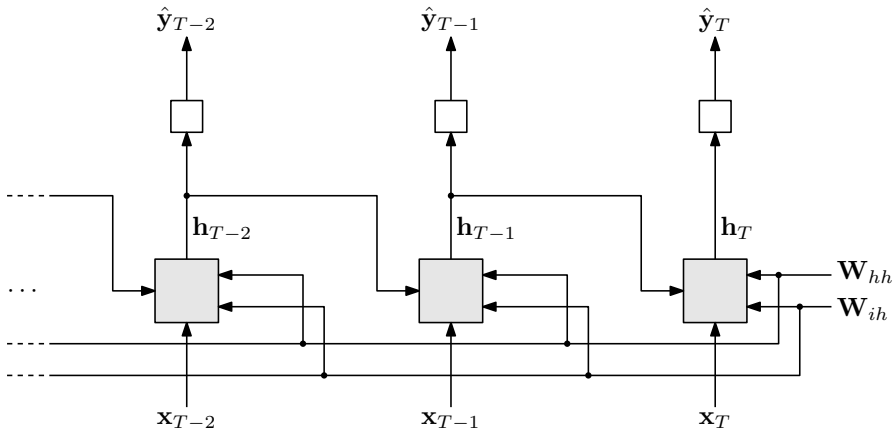
Consider the vector diagram of the general RNN:




If we unroll this RNN, we get the vector diagram:




The computational graph (not showing all the parameters) are given at the top of the next page.




Below we considered how we would determine the gradients $\boldsymbol{\delta}_{\mathbf{W}_{hh}}$ for $\mathbf{W}_{hh}$. The other gradients would be determined in a very similar way.

## The gradients in an RNN

Since $\mathbf{W}_{hh}$ is the input to more then one operation (node) in the graph (i.e. it forks), we would get the error term as the sum:

$$\begin{aligned}\boldsymbol{\delta}_{\mathbf{W}_{hh}} &= \text{prod}\left(\frac{\partial \mathbf{h}_T}{\partial \mathbf{W}_{hh}}, \boldsymbol{\delta}_{\mathbf{h}_T}\right) + \text{prod}\left(\frac{\partial \mathbf{h}_{T-1}}{\partial \mathbf{W}_{hh}}, \boldsymbol{\delta}_{\mathbf{h}_{T-1}}\right) + \\ &\quad \ldots + \text{prod}\left(\frac{\partial \mathbf{h}_1}{\partial \mathbf{W}_{hh}}, \boldsymbol{\delta}_{\mathbf{h}_1}\right) \\ &= \sum_{t=1}^{T} \text{prod}\left(\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}, \boldsymbol{\delta}_{\mathbf{h}_t}\right)\end{aligned}$$

This is from the generalised chain rule (scalars: $\frac{\partial J}{\partial u} = \frac{\partial z}{\partial u}\frac{\partial J}{\partial z} + \frac{\partial a}{\partial u}\frac{\partial J}{\partial a}$).

However, practically speaking, we won't be calculating $\boldsymbol{\delta}_{\mathbf{W}_{hh}}$ in one shot. Instead, we do exactly what the backpropagation algorithm says: Starting at the final operation, we will *accumulate* terms systematically as we proceed backwards through the graph:

$$\boldsymbol{\delta}_{\mathbf{W}_{hh}} \leftarrow \boldsymbol{\delta}_{\mathbf{W}_{hh}} + \text{prod}\left(\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}, \boldsymbol{\delta}_{\mathbf{h}_t}\right)$$

for $t = T, T-1, \ldots, 1$

This is why I sometimes call $\boldsymbol{\delta}_{\mathbf{W}_{hh}}$ the accumulator for $\mathbf{W}_{hh}$.

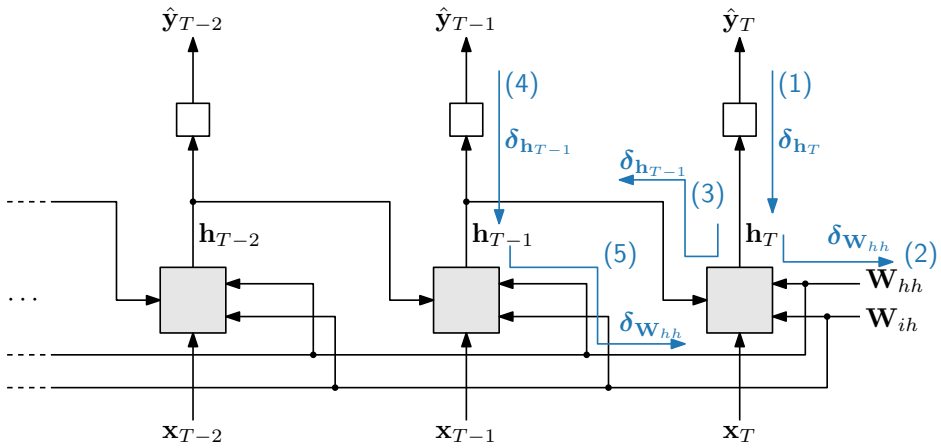
Note that we need to make sure that the accumulator for $\mathbf{h}_t$ is final before using it to update $\boldsymbol{\delta}_{\mathbf{W}_{hh}}$. Stated differently, before we can use $\boldsymbol{\delta}_{\mathbf{h}_t}$ all operations taking $\mathbf{h}_t$ as input should have been backpropped.

In this case $\mathbf{h}_t$ is involved as the input to two operations: one with output $\hat{\mathbf{y}}_t$ and the other with output $\mathbf{h}_{t+1}$. So before updating $\mathbf{W}_{hh}$, we should have already finished the accumulation for $\boldsymbol{\delta}_{\mathbf{h}_t}$:

$$\boldsymbol{\delta}_{\mathbf{h}_t} = \text{prod}\left(\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t}, \boldsymbol{\delta}_{\hat{\mathbf{y}}_t}\right) + \text{prod}\left(\frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}, \boldsymbol{\delta}_{\mathbf{h}_{t+1}}\right)$$

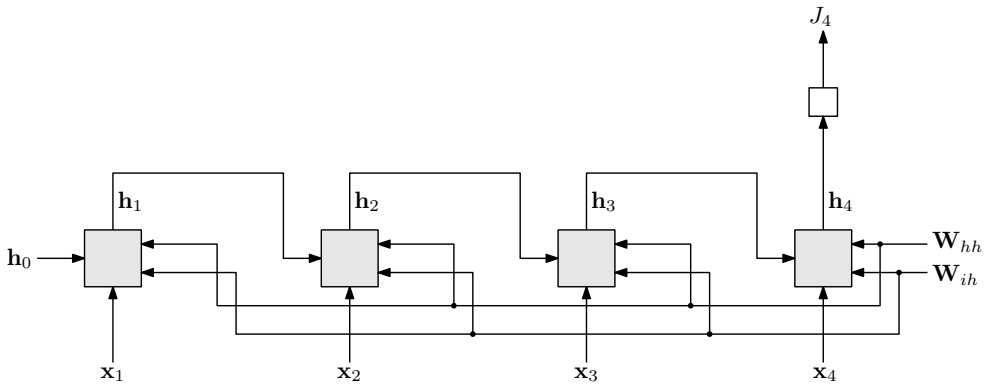
## The backprop order

The above implies a particular order for backpropagation through time:



# Vanishing and exploding gradients

Consider the RNN graph where we have an output after the fourth time step:




$$
\begin{aligned}
\frac{\partial J_4}{\partial \mathbf{h}_1} &= \text{prod}\left(\frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}, \frac{\partial J_4}{\partial \mathbf{h}_2}\right) \\
&= \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} \times \frac{\partial J_4}{\partial \mathbf{h}_2} \\
&= \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} \times \left[\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2} \times \frac{\partial J_4}{\partial \mathbf{h}_3}\right] \\
&= \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} \times \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2} \times \frac{\partial \mathbf{h}_4}{\partial \mathbf{h}_3} \times \frac{\partial J_4}{\partial \mathbf{h}_4}
\end{aligned}
$$

What happens if the elements in the matrices $\dfrac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$ become small? Or very large?

## Proof for linear RNNs

Consider an RNN with an identity activation: $g(x) = x$

Recurrence:

$$\mathbf{h}_t = \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{ih}\mathbf{x}_t + \mathbf{b}_h$$

Derivative of output of recurrence block w.r.t. $\mathbf{h}_t$:

$$\begin{aligned}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} &= \frac{\partial}{\partial \mathbf{h}_{t-1}} \left(\mathbf{W}_{hh}\mathbf{h}_{t-1}\right) \\ &= \mathbf{W}_{hh}^\top\end{aligned}$$

using the identity $\frac{\partial \mathbf{A}\mathbf{x}}{\partial \mathbf{x}} = \mathbf{A}^\top$.

Let's say our RNN has a single output only at the final time step $T$. What is the gradient of this loss $J_T(\boldsymbol{\theta})$ w.r.t. some intermediate hidden state $\mathbf{h}_i$?

$$\begin{aligned}\frac{\partial J_T}{\partial \mathbf{h}_i} &= \frac{\partial \mathbf{h}_{i+1}}{\partial \mathbf{h}_i} \times \frac{\partial \mathbf{h}_{i+2}}{\partial \mathbf{h}_{i+1}} \times \cdots \times \frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_{T-1}} \times \frac{\partial J_T}{\partial \mathbf{h}_T} \\ &= \left[\prod_{t=i+1}^{T} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}\right] \frac{\partial J_T}{\partial \mathbf{h}_T} \\ &= \left[\prod_{t=i+1}^{T} \mathbf{W}_{hh}^\top\right] \frac{\partial J_T}{\partial \mathbf{h}_T} \\ &= \left(\mathbf{W}_{hh}^\top\right)^{T-i} \frac{\partial J_T}{\partial \mathbf{h}_T}\end{aligned}$$

You can already start to see the problem: We have the power of a matrix $\mathbf{W}_{hh}^\top$, which will either die away or explode as the length $T - i$ gets bigger. Whether it dies or explodes will depend on the characteristics of the matrix.

One way to characterise the matrix is through its eigenvectors and eigenvalues. We can decompose $\mathbf{W}_{hh}^{\top}$ as

$$\mathbf{W}_{hh}^{\top} = \mathbf{P}\mathbf{D}\mathbf{P}^{-1}$$

where $\mathbf{P}$ is the matrix of eigenvectors as columns and $\mathbf{D}$ is a matrix with the eigenvalues on its diagonal. With this decomposition, you can get an expression for the power:

$$\left(\mathbf{W}_{hh}^{\top}\right)^{n} = \mathbf{P}\mathbf{D}^{n}\mathbf{P}^{-1}$$

as explained by slcmath@pc.

This means that the gradient above can be written as

$$\frac{\partial J_T}{\partial \mathbf{h}_i} = \mathbf{P}\mathbf{D}^{T-i}\,\mathbf{P}^{-1}\,\frac{\partial J_T}{\partial \mathbf{h}_T}$$

Now we can see that if the eigenvalues in $\mathbf{D}$ is $< 1$, then $\mathbf{D}^{T-i}$ will approach $\mathbf{0}$ as $T - i$ grows. I.e. the gradients will vanish as our sequences get longer.

On the other hand, if the eigenvalues are large, then the gradient will explode.

A similar proof exists for the non-linear case: Vanishing occurs when the eigenvalues are $< c$, with constant $c$ depending on the dimensionalities and the specific nonlinearity used.

*The derivation on this page will not be examined.*

## Solutions to exploding gradients

Exploding gradients could lead to overflow while training: `inf` or `NaN`

It could also cause irresponsibly large gradient steps:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\partial J}{\partial \boldsymbol{\theta}}$$

Solution: Gradient clipping

- If the norm of the gradient is larger than some threshold, scale the gradient down before taking a step.
- You are still taking a gradient step in the same direction, but just a smaller step.

## Solutions to vanishing gradients

Vanishing gradients mean your model can't learn long-term dependencies:

```
When she tried to print her tickets, she found
that the printer was out of toner. She went to
the stationery store to buy more toner. It was
very overpriced. After installing the toner
into the printer, she finally printed her ...
```

The solution is to use more advanced classes of RNN networks:

- Long short-term memory (LSTM)
- Gated recurrent units (GRUs)

Let's learn about these types of RNNs: colha's blog

Done. You are now an expert in LSTMs and GRUs.

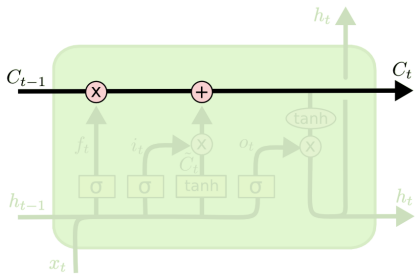
**Why do these networks fix the vanishing gradient problem?**

In the vanilla RNN we have:

$$\mathbf{h}_t = g\left(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{ih}\mathbf{x}_t + \mathbf{b}_h\right)$$

If $\mathbf{h}_1$ needs to affect the output at time step $T$, it has to go through a ton of $\mathbf{W}_{hh}$ multiplications ($T-1$ multiplications to be exact).

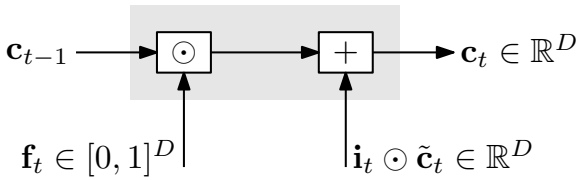
The crucial part in LSTMs and GRUs is the line running at the top of the cell state:[3]




This gives a direct connection between time step $t-1$ and $t$, without any matrix multiplication in between. The gate can *decide* to ignore what happened at $t-1$ (based on what is happening at $t$), but it can also decide to let past hidden information flow through unaltered.

[3]Figure from colha's blog.

To make this concrete, let's compare the gradients for the cell state to that of the hidden layer from the linear RNN that we looked at before.




For the cell line we have

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$

giving the derivative

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \mathrm{diag}(\mathbf{f}_t)$$

This derivative is different at different time steps $t$, since the forget gate output $\mathbf{f}_t$ is different. So we will repeatedly apply $\mathbf{f}_t \in [0,1]^D$, but sometimes it will let past information through (with values close to $1$) and sometimes it will block past information (values close to $0$).

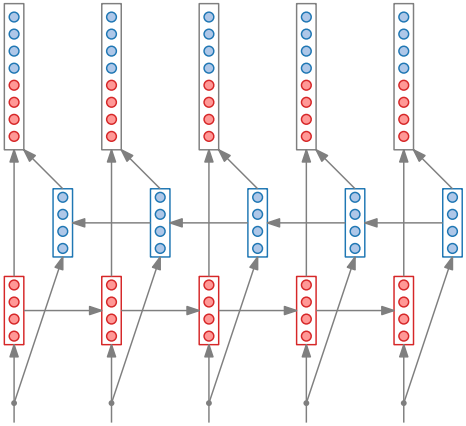
Contrast this with the derivative for the linear RNN:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \mathbf{W}_{hh}^\top$$

Here you have the same term being applied over and over again over the time steps.

# Extensions of RNNs

## Bidirectional RNNs

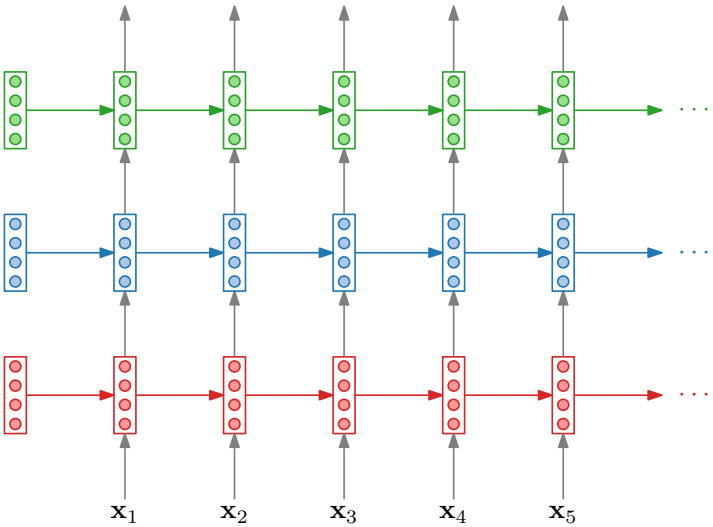


$$\mathbf{h_t} = \left[\overrightarrow{\mathbf{h}_t}; \overleftarrow{\mathbf{h}_t}\right]$$

$$\overleftarrow{\mathbf{h}_t} = \mathrm{RNN}_{\leftarrow}(\mathbf{h}_{t+1}, \mathbf{x}_t)$$

$$\overrightarrow{\mathbf{h}_t} = \mathrm{RNN}_{\rightarrow}(\mathbf{h}_{t-1}, \mathbf{x}_t)$$

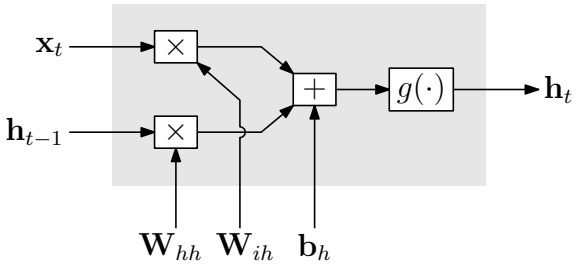
## Multilayer RNNs



# Common misconceptions

## Combing operations in a node in the computational graph

Not realising that nodes in a computational graph can contain internal operations itself. As long as you can get a node's outputs from its inputs (forward pass) and the derivatives of its outputs w.r.t. its inputs (backward), several operations can be combined as a single node. E.g. in the RNN computational graphs in this note I use grey blocks to indicate the recurrent operation:




$$\mathbf{h}_t = g\left(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{ih}\mathbf{x}_t + \mathbf{b}_h\right)$$

## Videos covered in this note

- From feedforward to recurrent neural networks (15 min)
- RNN language model loss function (9 min)
- RNN definition and computational graph (3 min)
- Backpropagation through time (25 min)
- Vanishing and exploding gradients in RNNs (13 min)
- Solutions to exploding and vanishing gradients (in RNNs) (10 min)
- Extensions of RNNs (8 min)

## Further reading

For vanilla RNNs, and specifically looking at different input-output configurations, make sure to read Andrej Karpathy's blog post: *The unreasonable effectiveness of recurrent neural networks*.

## Acknowledgements

This note relied very very heavily on content from:

- Chris Manning's CS224N course at Stanford University
- The D2L textbook from Zhang et al. (2021)

## References

C. Manning, "CS224N: Language models and recurrent neural networks," *Stanford University*, 2022.

C. Manning, "CS224N: Simple and LSTM recurrent neural networks," *Stanford University*, 2022.

A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*, 2021.