LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Deep Learning

Chapter 8: Recurrent Neural Networks (RNNs)

Mina Rezaei

Department of Statistics – LMU Munich

Winter Semester 2020

LECTURE OUTLINE

RNNs – The basic idea

RNNs - Computational Graph

Computing the Loss Gradient in RNNs

MOTIVATION FOR RNNS

- The two major types of neural network architectures that we've seen so far are fully-connected networks and Convolutional Neural Networks (CNNs).
- In both cases, the input layers have a fixed size and, therefore, these networks can (typically) only handle fixed-length inputs.
- The primary reason for this is that it is not possible to vary the size of the input layer without also varying the number of learnable parameters/weights in the network.
- In many cases, we would like to feed **variable length inputs** to the network.
- Common examples of this are **sequence data** such as time-series, audio and text.
- Therefore, we need a new class of neural network architectures that are able to handle such variable length inputs: **Recurrent Neural Networks (RNNs)**.

RNNs – The basic idea

RNNS - INTRODUCTION

- Suppose we have some text data and our task is to analyse the *sentiment* in the text.
- For example, given an input sentence, such as "This is good news.", the network has to classify it as either 'positive' or 'negative'.
- We would like to train a simple neural network (such as the one below) to perform the task.

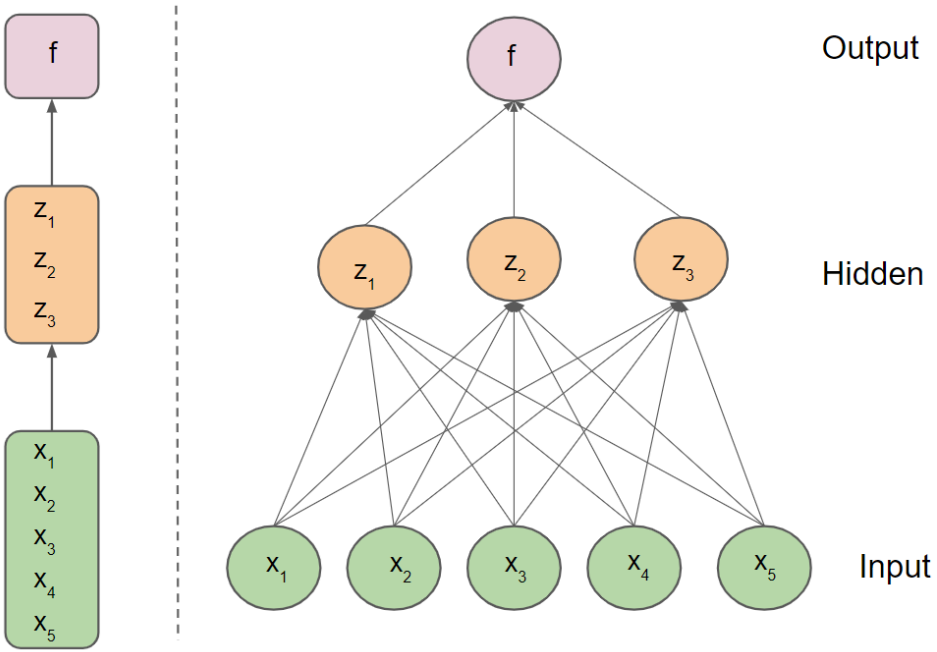


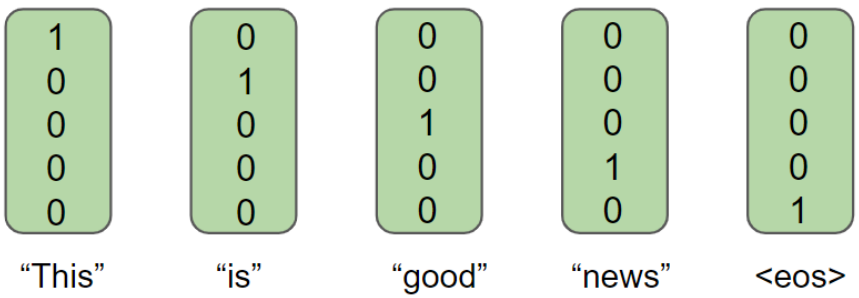
Figure: Two equivalent visualizations of a dense net with a single hidden layer.

RNNS - INTRODUCTION

- Because sentences can be of varying lengths, we need to modify the dense net architecture to handle such a scenario.
- One approach is to draw inspiration from the way a human reads a sentence; that is, one word at a time.
- An important cognitive mechanism that makes this possible is "**short-term memory**".
- As we read a sentence from beginning to end, we retain some information about the words that we have already read and use this information to understand the meaning of the entire sentence.
- Therefore, in order to feed the words in a sentence sequentially to a neural network, we need to give it the ability to retain some information about past inputs.

RNNS - INTRODUCTION

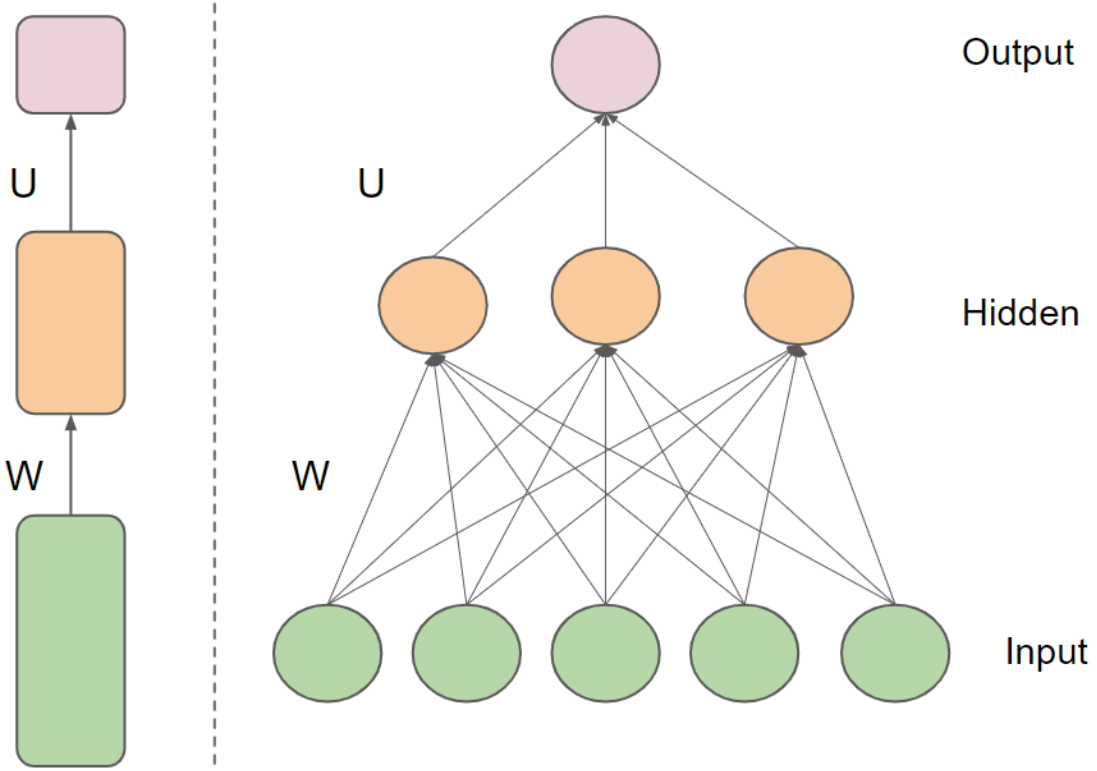
- When words in a sentence are fed to the network one at a time, the inputs are no longer independent. For example, it is much more likely that the word "good" is followed by "morning" rather than "plastic". **We need to model this (long-term) dependence.**
- Each word must still be encoded as a fixed-length vector because the size of the input layer will remain fixed.
- Here, for the sake of the visualization, each word is represented as a 'one-hot coded' vector of length 5. (<eos> = 'end of sequence')



(The standard approach is to use word embeddings (more on this later)).

RNNS - INTRODUCTION

- Our goal is to feed the words to the network sequentially in discrete time-steps.
- A regular dense neural network with a single hidden layer only has two sets of weights: 'input-to-hidden' weights ***W*** and 'hidden-to-output' weights ***U***.



RNNS - INTRODUCTION

- In order to enable the network to retain information about past inputs, we introduce an **additional set of weights** $\boldsymbol{V}$, from the hidden neurons at time-step t to the hidden neurons at time-step $t + 1$.
- Having this additional set of weights makes the activations of the hidden layer depend on **both** the current input and the activations for the *previous* input.

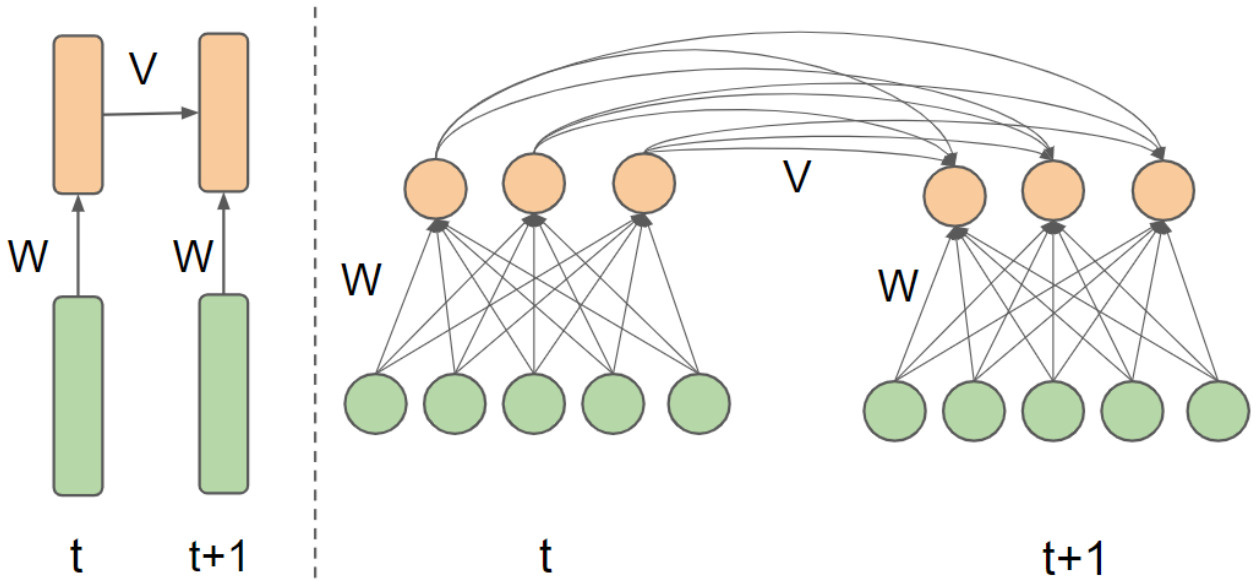


Figure: Input-to-hidden weights $\boldsymbol{W}$ and **hidden-to-hidden** weights $\boldsymbol{V}$. The hidden-to-output weights $\boldsymbol{U}$ are not shown in the figure.

RNNS - INTRODUCTION

- With this additional set of hidden-to-hidden weights $\boldsymbol{V}$, the network is now a Recurrent Neural Network (RNN).
- In a regular feed-forward network, the activations of the hidden layer are only computed using the input-hidden weights $\boldsymbol{W}$ (and bias $\boldsymbol{b}$).

$$\boldsymbol{z} = \sigma(\boldsymbol{W}^\top \mathbf{x} + \boldsymbol{b})$$

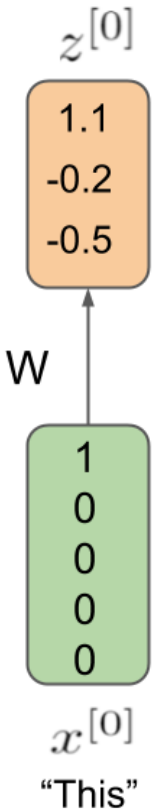
- In an RNN, the activations of the hidden layer (at time-step t) are computed using *both* the input-to-hidden weights W and the hidden-to-hidden weights $\boldsymbol{V}$.

$$\boldsymbol{z}^{[t]} = \sigma(\boldsymbol{V}^\top \boldsymbol{z}^{[t-1]} + \boldsymbol{W}^\top \boldsymbol{x}^{[t]} + \boldsymbol{b})$$

- The vector $\boldsymbol{z}^{[t]}$ represents the short-term memory of the RNN because it is a function of the current input $\boldsymbol{x}^{[t]}$ and the activations $\boldsymbol{z}^{[t-1]}$ of the previous time-step.
- Therefore, by recurrence, it contains a "summary" of *all* previous inputs.

APPLICATION EXAMPLE - SENTIMENT ANALYSIS

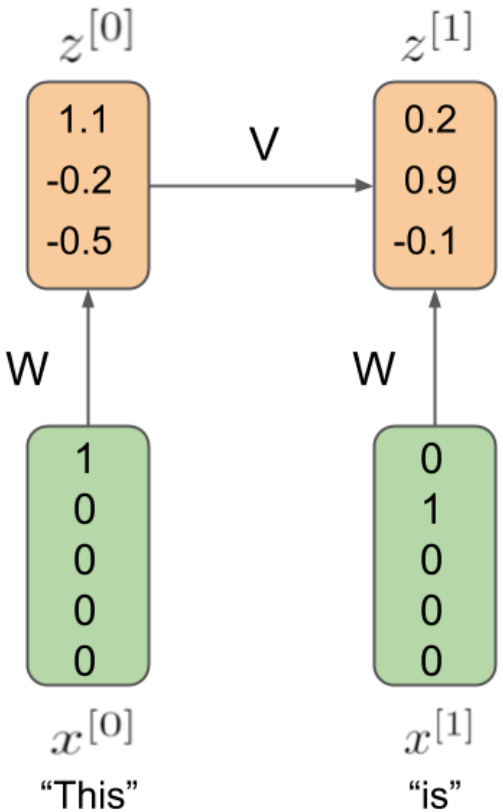
- At $t = 0$, we feed the word "This" to the network and obtain $\boldsymbol{z}^{[0]}$.
- $\boldsymbol{z}^{[0]} = \sigma(\boldsymbol{W}^\top \boldsymbol{x}^{[0]} + \boldsymbol{b})$



Because this is the very first input, there is no past state (or, equivalently, the state is initialized to 0).

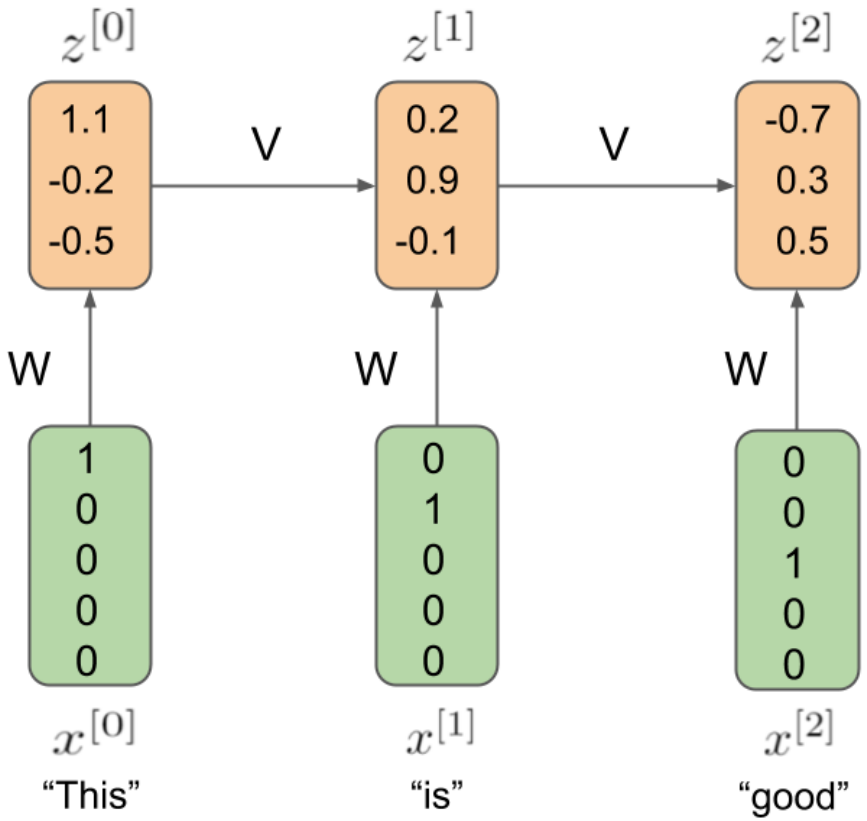
APPLICATION EXAMPLE - SENTIMENT ANALYSIS

- At $t = 1$, we feed the second word to the network to obtain $\boldsymbol{z}^{[1]}$.
- $\boldsymbol{z}^{[1]} = \sigma(\boldsymbol{V}^\top \boldsymbol{z}^{[0]} + \boldsymbol{W}^\top \mathbf{x}^{[1]} + \boldsymbol{b})$



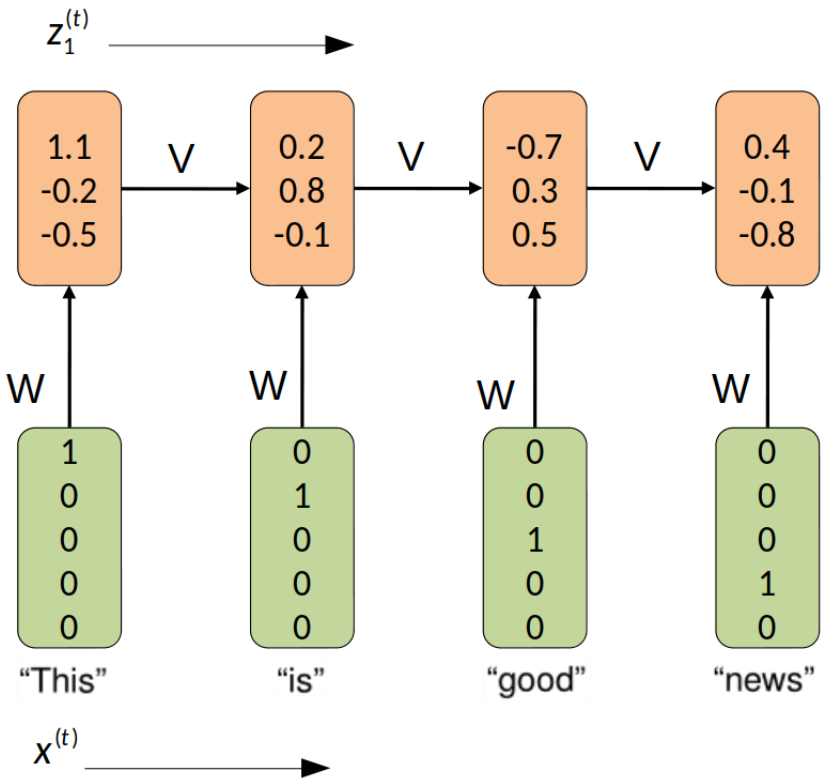
APPLICATION EXAMPLE - SENTIMENT ANALYSIS

- At $t = 2$, we feed the next word in the sentence.
- $\boldsymbol{z}^{[2]} = \sigma(\boldsymbol{V}^\top \boldsymbol{z}^{[1]} + \boldsymbol{W}^\top \mathbf{x}^{[2]} + \boldsymbol{b})$



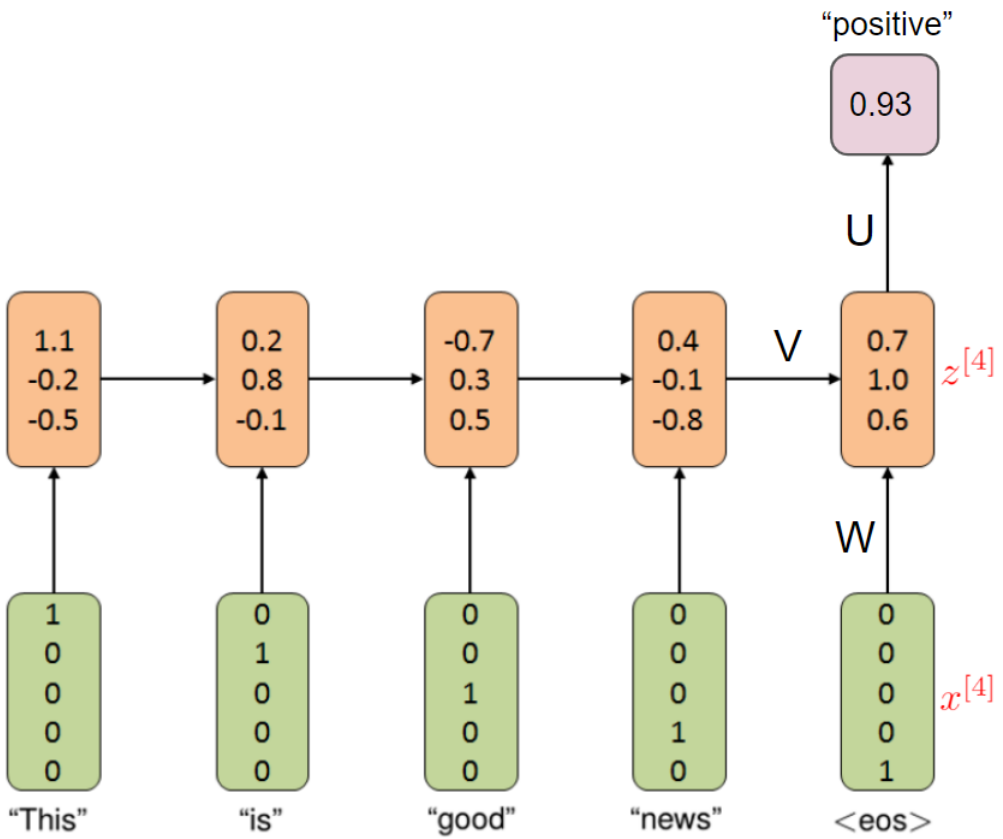
APPLICATION EXAMPLE - SENTIMENT ANALYSIS

- At $t = 3$, we feed the next word ("news") in the sentence.
- $\boldsymbol{z}^{[3]} = \sigma(\boldsymbol{V}^\top \boldsymbol{z}^{[2]} + \boldsymbol{W}^\top \mathbf{x}^{[3]} + \boldsymbol{b})$



APPLICATION EXAMPLE - SENTIMENT ANALYSIS

- Once the entire input sequence has been processed, the prediction of the network can be generated by feeding the activations of the final time-step to the output neuron(s).
- $f = \sigma(\boldsymbol{U}^\top \boldsymbol{z}^{[4]} + c)$, where c is the bias of the output neuron.



PARAMETER SHARING

- This way, the network can process the sentence one word at a time and the length of the network can vary based on the length of the sequence.
- It is important to note that no matter how long the input sequence is, the matrices $\boldsymbol{W}$ and $\boldsymbol{V}$ are the same in every time-step. This is another example of **parameter sharing**.
- Therefore, the number of weights in the network is independent of the length of the input sequence.

SEQUENCE MODELING: – DESIGN CRITERIA

To model sequence we need to:

- Handle variable-length sequence,
- Track long-term dependencies,
- Maintain information about order,
- Share parameter across the sequence.

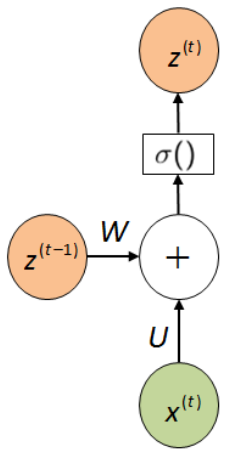
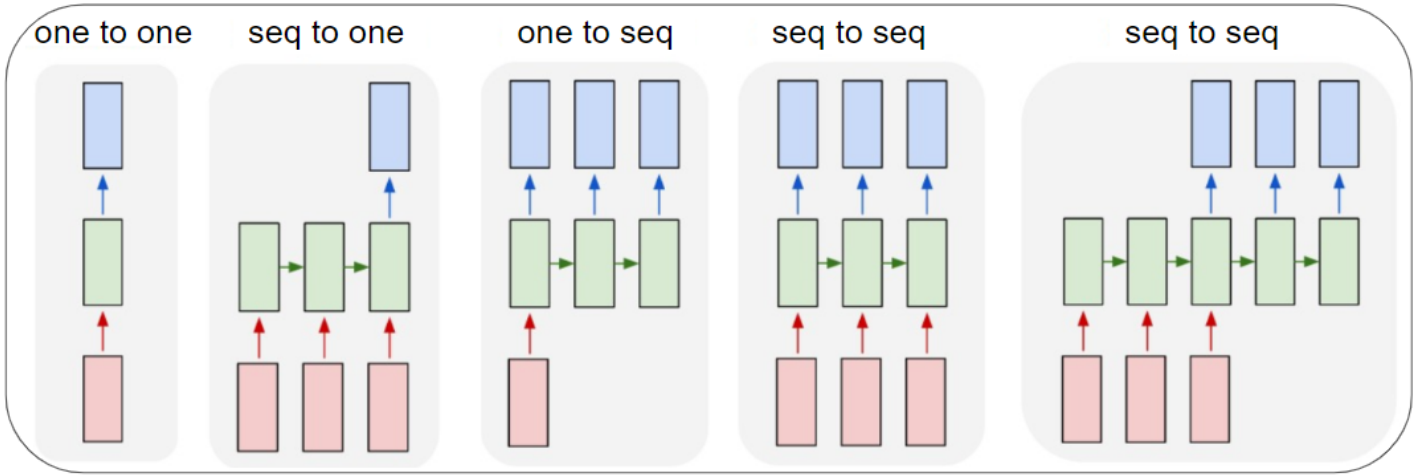


Figure: Vanilla RNN

RNNS - USE CASE SPECIFIC ARCHITECTURES

RNNs are very versatile. They can be applied to a wide range of tasks.



Credit: Andrej Karpathy

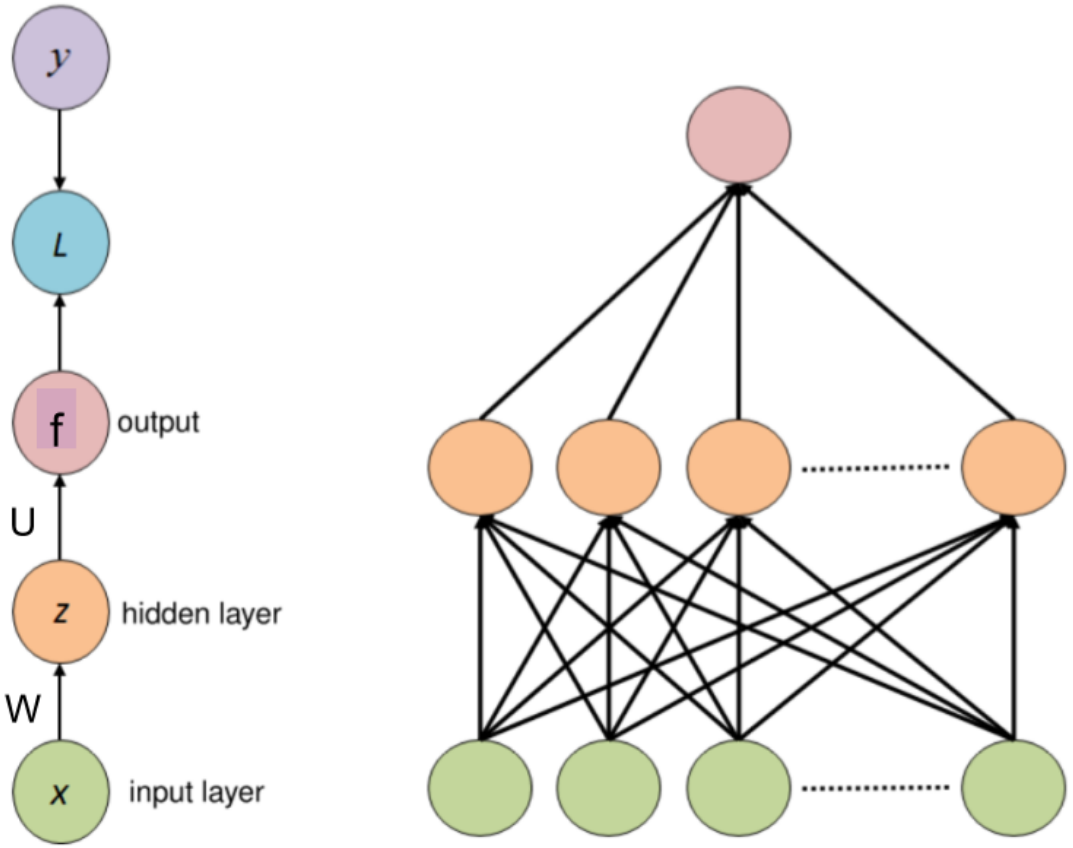
Figure: RNNs can be used in tasks that involve multiple inputs and/or multiple outputs.

Examples:

- Sequence-to-One: Sentiment analysis, document classification.
- One-to-Sequence: Image captioning.
- Sequence-to-Sequence: Language modelling, machine translation, time-series prediction.

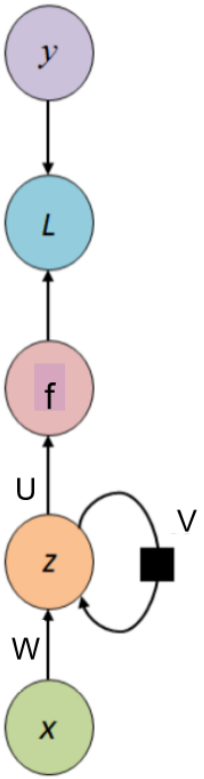
RNNs - Computational Graph

RNNS - COMPUTATIONAL GRAPH



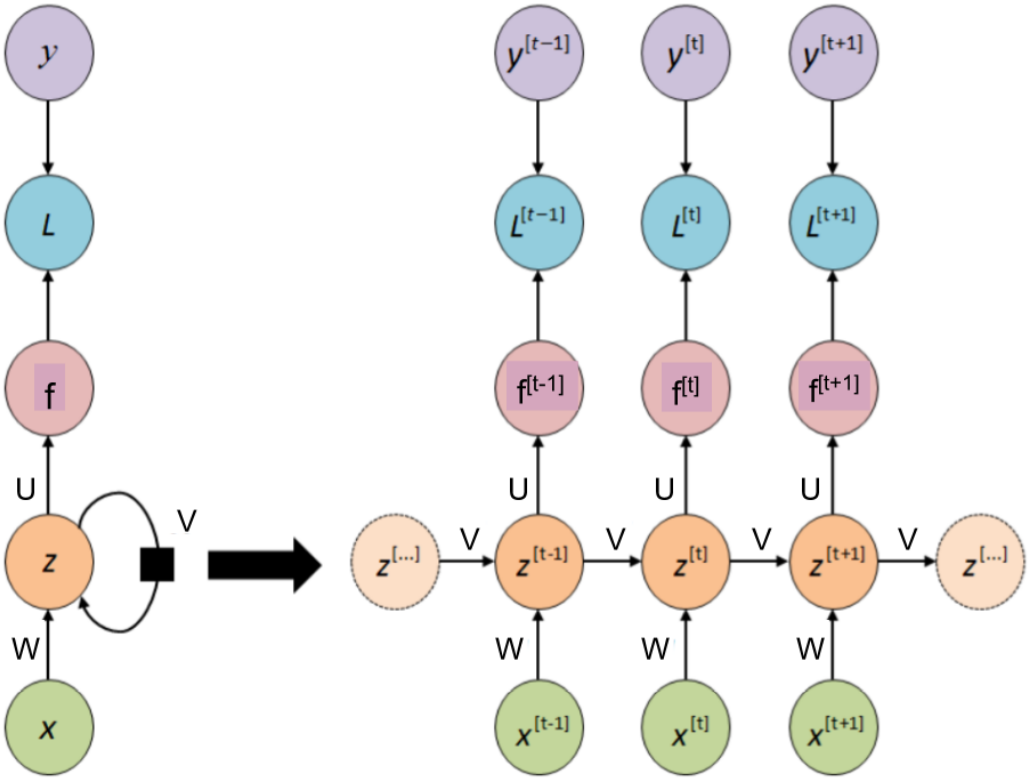
- On the left is the computational graph for the dense net on the right. A loss function L measures how far each output f is from the corresponding training target y.

RNNS - COMPUTATIONAL GRAPH



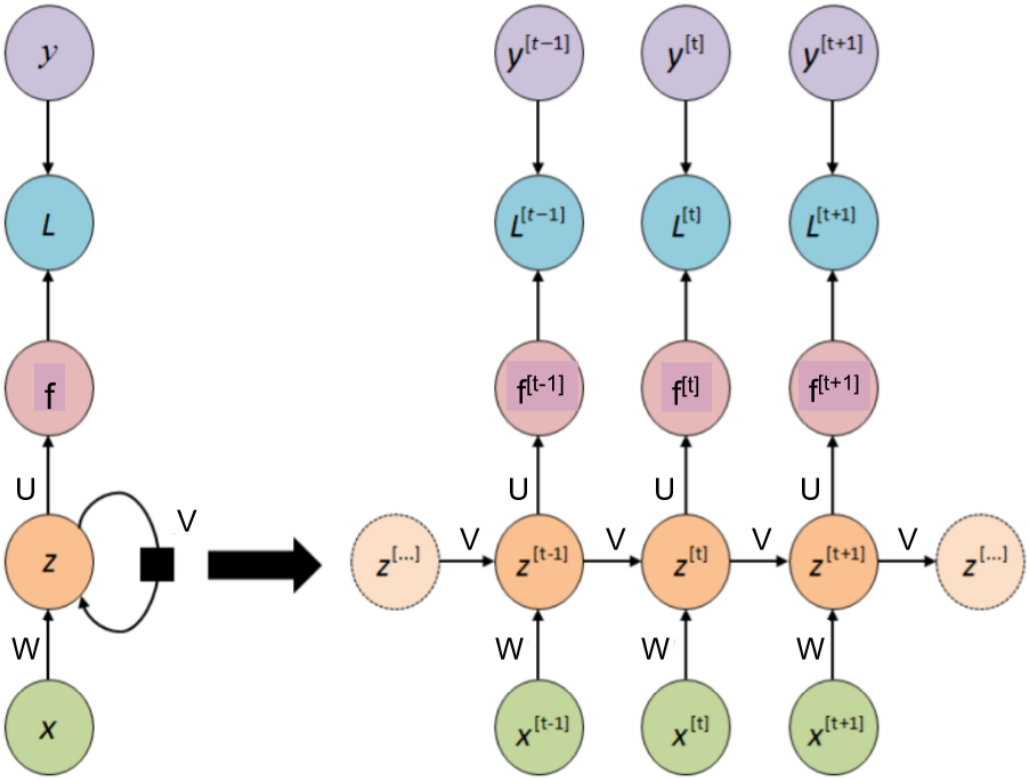
- A helpful way to think of an RNN is as multiple copies of the same network, each passing a message to a successor.
- RNNs are networks with loops, allowing information to persist.

RNNS - COMPUTATIONAL GRAPH



- Things might become more clear if we unfold the architecture.
- We call $\boldsymbol{z}^{[t]}$ the *state* of the system at time t.
- Recall, the state contains information about the whole past sequence.

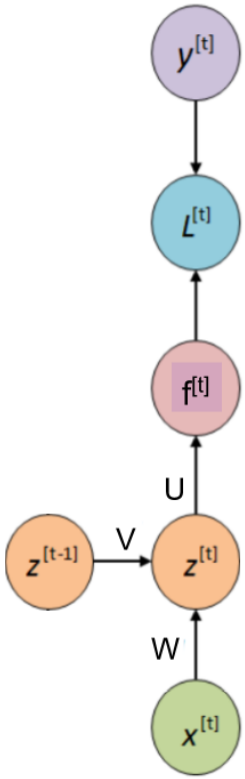
RNNS - COMPUTATIONAL GRAPH



- We went from

$$f = \tau(c + \boldsymbol{U}^\top \sigma(\boldsymbol{b} + \boldsymbol{W}^\top \boldsymbol{x})) \text{ for the dense net, to}$$
$$f^{[t]} = \tau(c + \boldsymbol{U}^\top \sigma(\boldsymbol{b} + \boldsymbol{V}^\top \boldsymbol{z}^{[t-1]} + \boldsymbol{W}^\top \mathbf{x}^{[t]})) \text{ for the RNN.}$$

RNNS - COMPUTATIONAL GRAPH

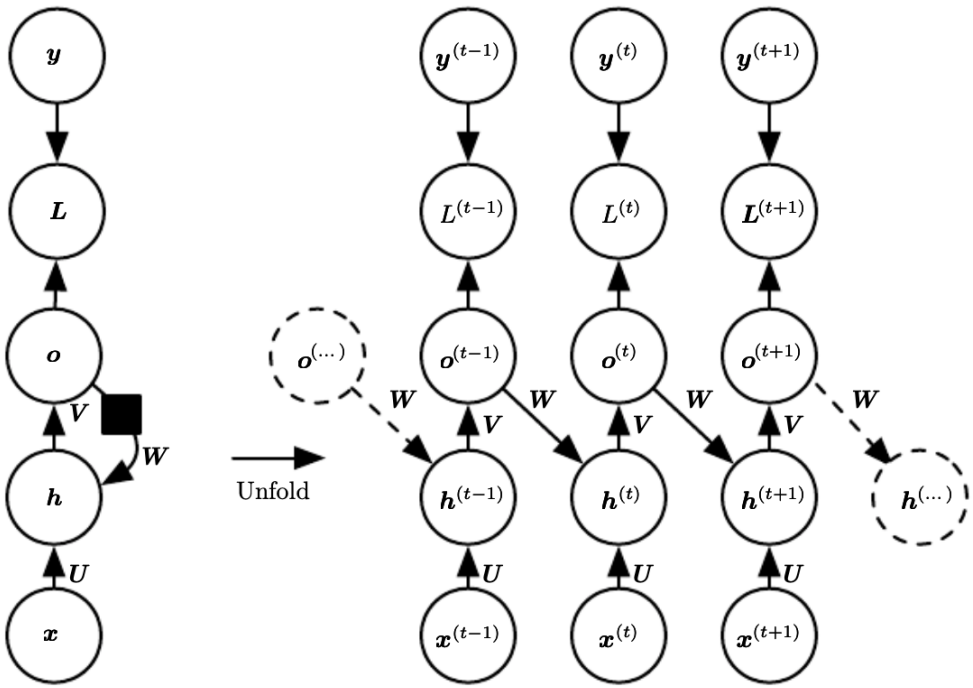


- Potential computational graph for time-step t:

$$f^{[t]} = \tau(c + \boldsymbol{U}^\top \sigma(\boldsymbol{b} + \boldsymbol{V}^\top \boldsymbol{z}^{[t-1]} + \boldsymbol{W}^\top \mathbf{x}^{[t]}))$$

RNNS - COMPUTATIONAL GRAPH WITH RECURRENT OUTPUT-HIDDEN CONNECTIONS

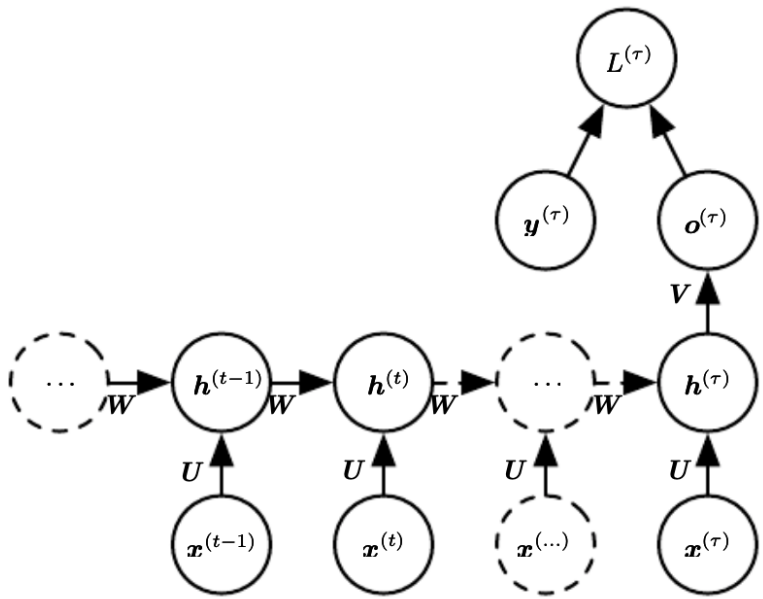
Recurrent connections do not need to map from hidden to hidden neurons!



An RNN whose only recurrence is the feedback connection from the output to the hidden layer. At each time stept, the input is x_t, the hidden layer activations are $h^{(t)}$, the outputs are $o^{(t)}$, the targets are $y^{(t)}$, and the loss is $L^{(t)}$. (Left) Circuitdiagram. (Right) Unfolded computational graph. The RNN in this figure is trained to put a specific output value into o, and o is the only information it is allowed to send to the future. There are no direct connections from h going forward. The previous h is connected to the present only indirectly, via the predictions it was used to produce.

RNNS - COMPUTATIONAL GRAPH FOR SEQ TO ONE MAPPING

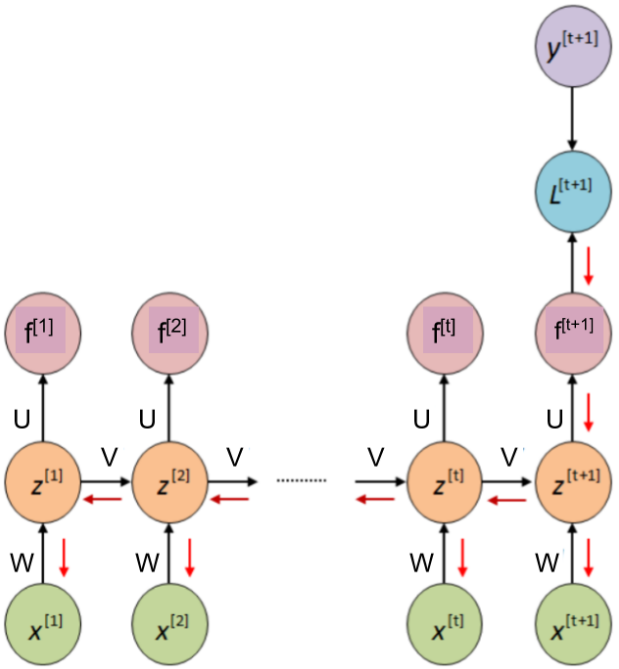
RNNs do not need to produce an output at each time step. Often only one output is produced after processing the whole sequence.



Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed size representation used as input for further processing. There might be a target right at the end (as depicted here), or the gradient on the output $o^{(t)}$ can be obtained by backpropagating from further downstream modules.

Computing the Loss Gradient in RNNs

BACKPROPAGATION THROUGH TIME



- For training the RNN we need to compute $\frac{dL}{du_{i,j}}$, $\frac{dL}{dv_{i,j}}$, and $\frac{dL}{dw_{i,j}}$.
- To do so, during backpropagation at time step t we may need to compute

$$\frac{dL}{d\boldsymbol{z}^{[1]}} = \frac{dL}{d\boldsymbol{z}^{[t]}} \frac{d\boldsymbol{z}^{[t]}}{d\boldsymbol{z}^{[t-1]}} \cdots \frac{d\boldsymbol{z}^{[2]}}{d\boldsymbol{z}^{[1]}}$$

LONG-TERM DEPENDENCIES

- Here, $\boldsymbol{z}^{[t]} = \sigma(\boldsymbol{V}^\top \boldsymbol{z}^{[t-1]} + \boldsymbol{W}^\top \mathbf{x}^{[t]} + \boldsymbol{b})$
- It follows that:

$$\frac{d\boldsymbol{z}^{[t]}}{d\boldsymbol{z}^{[t-1]}} = \text{diag}(\sigma'(\boldsymbol{V}^\top \boldsymbol{z}^{[t-1]} + \boldsymbol{W}^\top \mathbf{x}^{[t]} + \boldsymbol{b}))\boldsymbol{V}^\top = \boldsymbol{D}^{[t-1]}\boldsymbol{V}^\top$$

$$\frac{d\boldsymbol{z}^{[t-1]}}{d\boldsymbol{z}^{[t-2]}} = \text{diag}(\sigma'(\boldsymbol{V}^\top \boldsymbol{z}^{[t-2]} + \boldsymbol{W}^\top \mathbf{x}^{[t-1]} + \boldsymbol{b}))\boldsymbol{V}^\top = \boldsymbol{D}^{[t-2]}\boldsymbol{V}^\top$$

$$\vdots$$

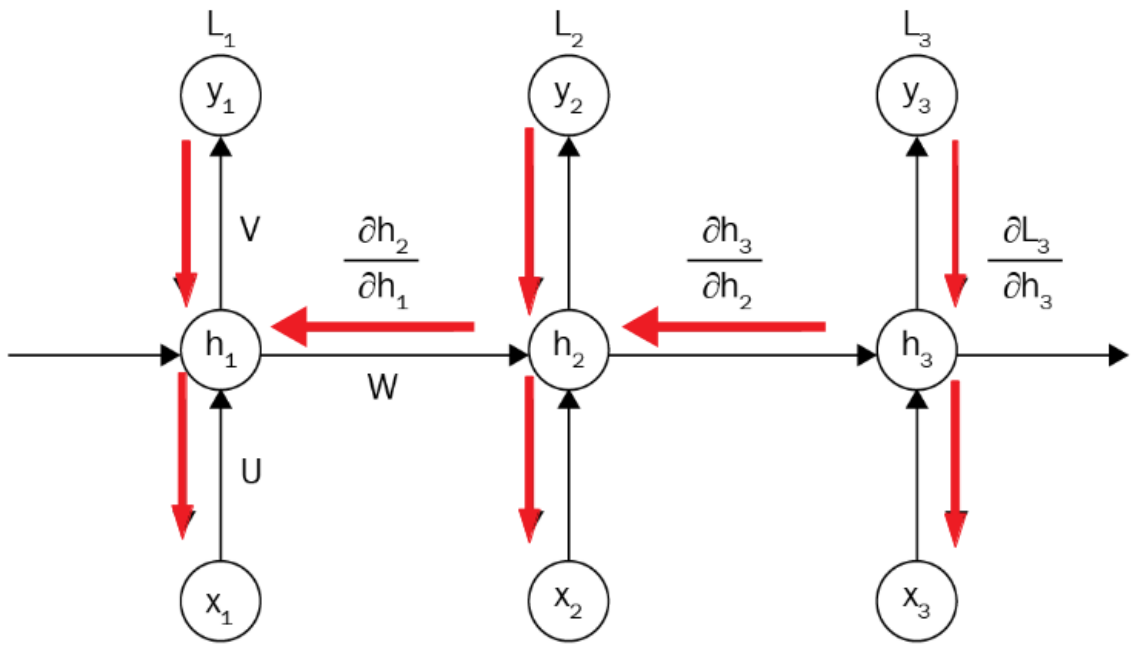
$$\frac{d\boldsymbol{z}^{[2]}}{d\boldsymbol{z}^{[1]}} = \text{diag}(\sigma'(\boldsymbol{V}^\top \boldsymbol{z}^{[1]} + \boldsymbol{W}^\top \mathbf{x}^{[2]} + \boldsymbol{b}))\boldsymbol{V}^\top = \boldsymbol{D}^{[1]}\boldsymbol{V}^\top$$

$$\frac{dL}{d\boldsymbol{z}^{[1]}} = \frac{dL}{d\boldsymbol{z}^{[t]}}\frac{d\boldsymbol{z}^{[t]}}{d\boldsymbol{z}^{[t-1]}}\cdots\frac{d\boldsymbol{z}^{[2]}}{d\boldsymbol{z}^{[1]}} = \boldsymbol{D}^{[t-1]}\boldsymbol{D}^{[t-2]}\ldots\boldsymbol{D}^{[1]}(\boldsymbol{V}^\top)^{[t-1]}$$

LONG-TERM DEPENDENCIES

- Therefore, for an arbitrary time-step i in the past, $\frac{d\boldsymbol{z}^{[t]}}{d\boldsymbol{z}^{[i]}}$ will contain the term $(\boldsymbol{V}^\top)^{t-i}$ within it (this follows from the chain rule).
- Based on the largest eigenvalue of $\boldsymbol{V}^\top$, the presence of the term $(\boldsymbol{V}^\top)^{t-i}$ can either result in vanishing or exploding gradients.
- This problem is quite severe for RNNs (as compared to feedforward networks) because the **same** matrix $\boldsymbol{V}^\top$ is multiplied several times. ▸ Click here
- As the gap between t and i increases, the instability worsens.
- It is quite challenging for RNNs to learn long-term dependencies. The gradients either **vanish** (most of the time) or **explode** (rarely, but with much damage to the optimization).
- That happens simply because we propagate errors over very many stages backwards.

LONG-TERM DEPENDENCIES



Many values > 1:
exploding gradient

Gradient clipping to scale big gradient

Figure: Exploding of gradients

LONG-TERM DEPENDENCIES

- Recall, that we can counteract exploding gradients by implementing gradient clipping.
- To avoid exploding gradients, we simply clip the norm of the gradient at some threshold h (see chapter 4):

$$\text{if } ||\nabla W|| > \text{h} : \nabla W \leftarrow \frac{h}{||\nabla W||} \nabla W$$

LONG-TERM DEPENDENCIES

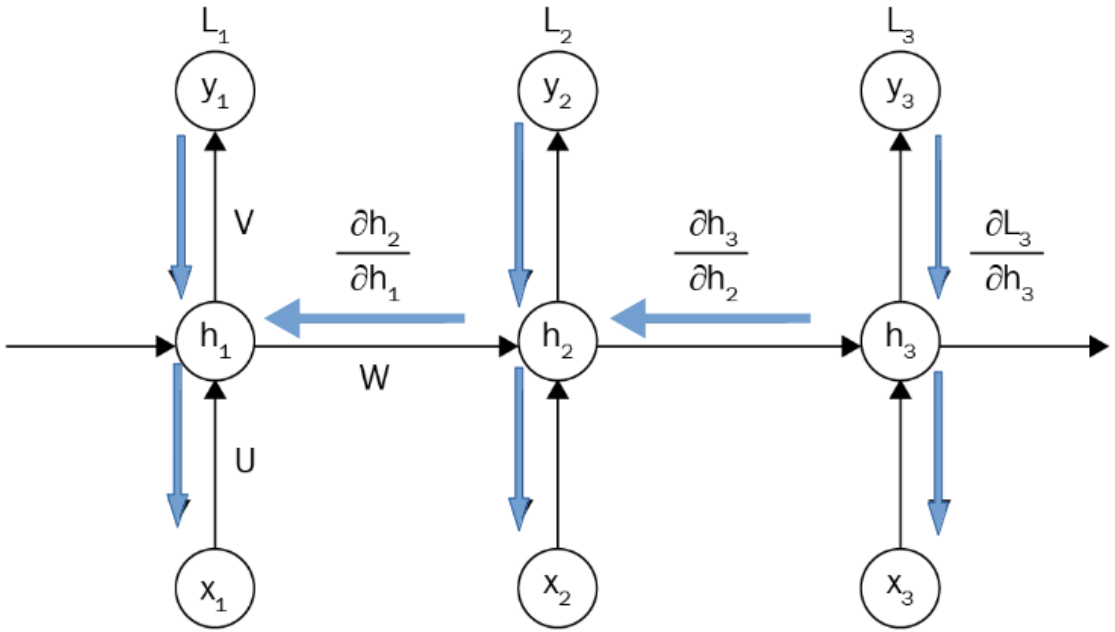


Figure: Vanishing gradients

LONG-TERM DEPENDENCIES

- Even if we assume that the parameters are such that the recurrent network is stable (can store memories, with gradients not exploding), the difficulty with long-term dependencies arises from the exponentially smaller weights given to long-term interactions (involving the multiplication of many Jacobians) compared to short-term ones.
- A more sophisticated solution is needed for the vanishing gradient problem in RNNs.
- The **vanishing gradient problem** is heavily dependent on the parameter initialization method, but in particular on the choice of the activation functions.
  - For example, the sigmoid maps a real number into a “small” range (i.e. $[0, 1]$).
  - As a result, large regions of the input space are mapped into a very small range.

LONG-TERM DEPENDENCIES

- Even a huge change in the input will only produce a small change in the output. Hence, the gradient will be small.
- This becomes even worse when we stack multiple layers of such non-linearities on top of each other (For instance, the first layer maps a large input to a smaller output region, which will be mapped to an even smaller region by the second layer, which will be mapped to an even smaller region by the third layer and so on..).
- We can avoid this problem by using activation functions which do not have the property of "squashing" the input.
- The most popular choice is obviously the Rectified Linear Unit (ReLU) which maps x to $max(0, x)$.
- The really nice thing about ReLU is that the gradient is either 0 or 1, which means it never saturates. Thus, gradients can't vanish.

LONG-TERM DEPENDENCIES

- The downside of this is that we can obtain a “dead” ReLU. It will always return 0 and consequently never learn because the gradient is not passed through.

REFERENCES

Ian Goodfellow, Yoshua Bengio and Aaron Courville (2016)

Deep Learning

http://www.deeplearningbook.org/

Andrej Karpathy (2015)

The Unreasonable Effectiveness of Recurrent Neural Networks

http://karpathy.github.io/2015/05/21/rnn-effectiveness/