LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Deeplearning

Chapter 7: Sequence Modeling: Recurrent and Recursive Nets

Bernd Bischl

Department of Statistics – LMU Munich

Winter term 2018

RNNS - WHAT FOR?

Suppose we would like to process sequential inputs, such as

- Text data
- Audio data

Can we do that with a classic dense net?

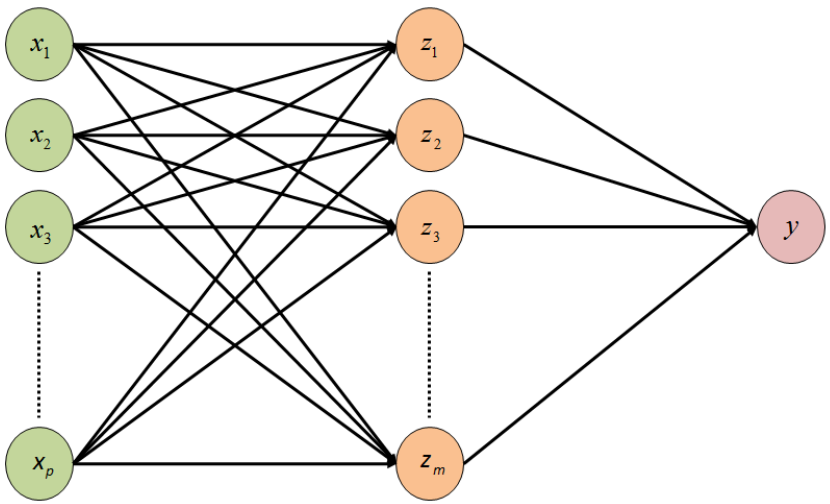


Figure: A dense architecture.

RNNS - WHAT FOR?

Suppose we would like to process sequential inputs, such as

- Text data
- Audio data

Can we do that with a classic dense net?

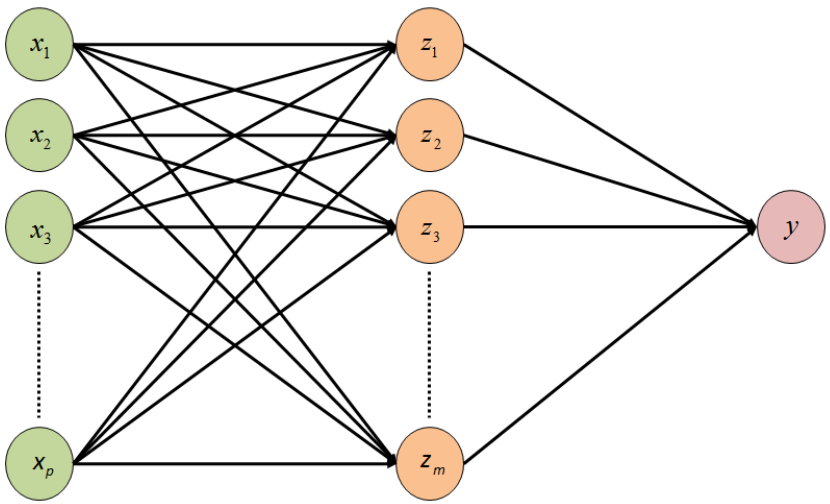


Figure: A dense architecture.

Hardly, the major drawbacks of these models are:

- **a fixed input length**.
  The length of sequential inputs can vary!
- **the assumption of independent training samples**.
  For sequential inputs, there are short and long term temporal dependencies!

RNNS - SENTIMENT ANALYSIS

- Suppose we would like to train a model to read a sentence and extract the year the narrator went to munich:
  - “I went to Munich in 2009”
  - “In 2009, I went to Munich”
- A standard dense network would have separate parameters for each input feature. Thus it would need to learn all of the rules of the language separately at each position in the sentence!
- To overcome this issue, we introduce **recurrent neural networks**!
- In order to go from a standard dense to such a recurrent net, we need to take advantage of an idea we have already learned in the CNN chapter: **parameter sharing**.

RNNS - SENTIMENT ANALYSIS

- Parameter sharing enables us to apply the model to examples of different forms (here: different lengths)!
- If we had separate parameters for each value of the input data, we could not generalize to sequence lengths not seen during training.
- Parameter sharing is specifically important, when a particular piece of information might occur at multiple positions within the input sequence.

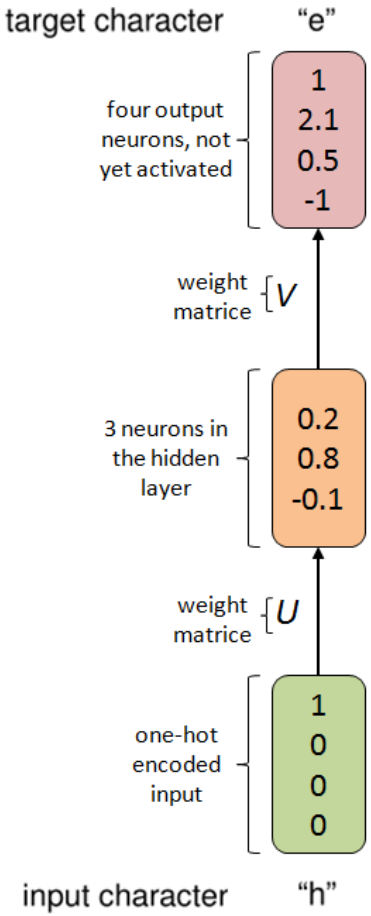
RNNS - GENERATE SEQUENCES

- Suppose we only had a vocabulary of four possible letters:

  “h”, “e”, “l” and “o”

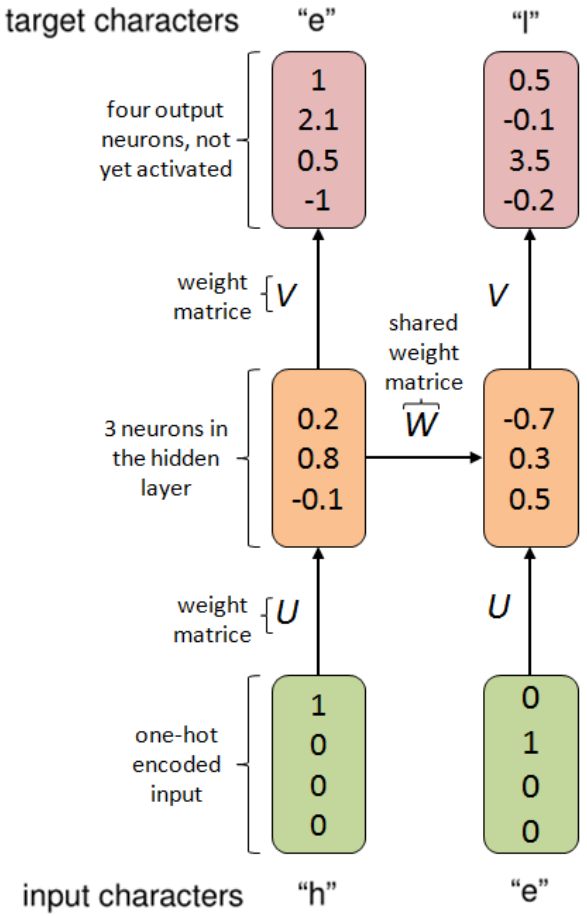
- Our goal is to train a neural net on the sequence “hello”.
- Our training sequence “hello” is in actually a source of 4 separate training examples:
  - The probability of “e” should be likely, given the context of “h”
  - “l” should be likely in the context of “he”
  - “l” should **also** be likely, given the context of “hel”
  - and finally “o” should be likely, given the context of “hell”
- So how to realize that?

RNNS - GENERATE SEQUENCES



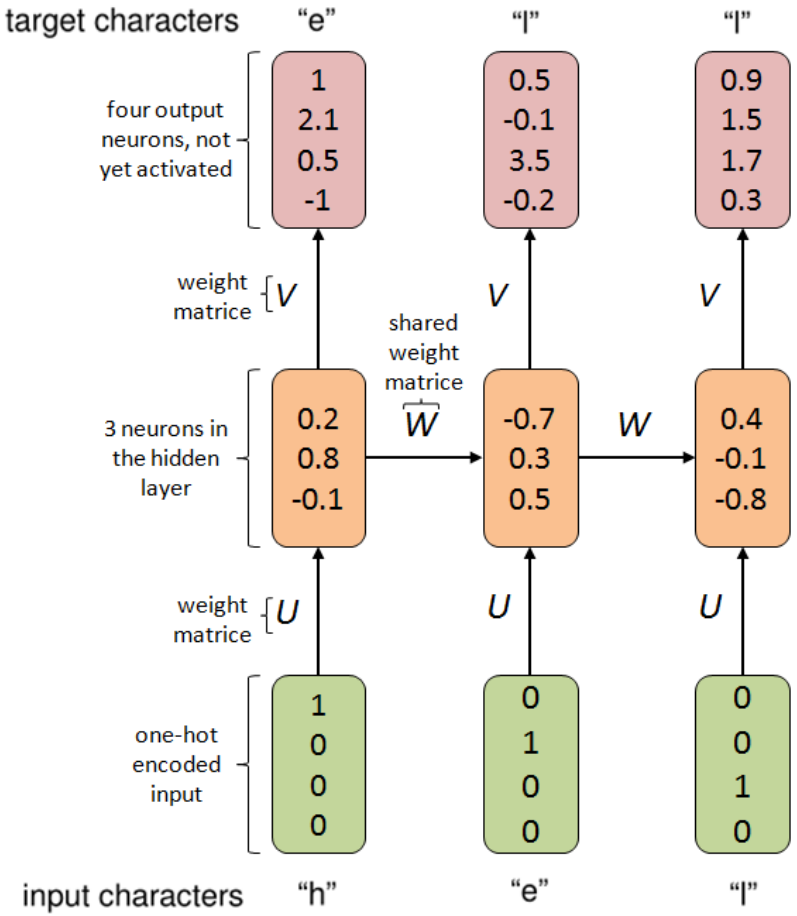
The probability of "e" should be likely, given the context of "h"

RNNS - GENERATE SEQUENCES



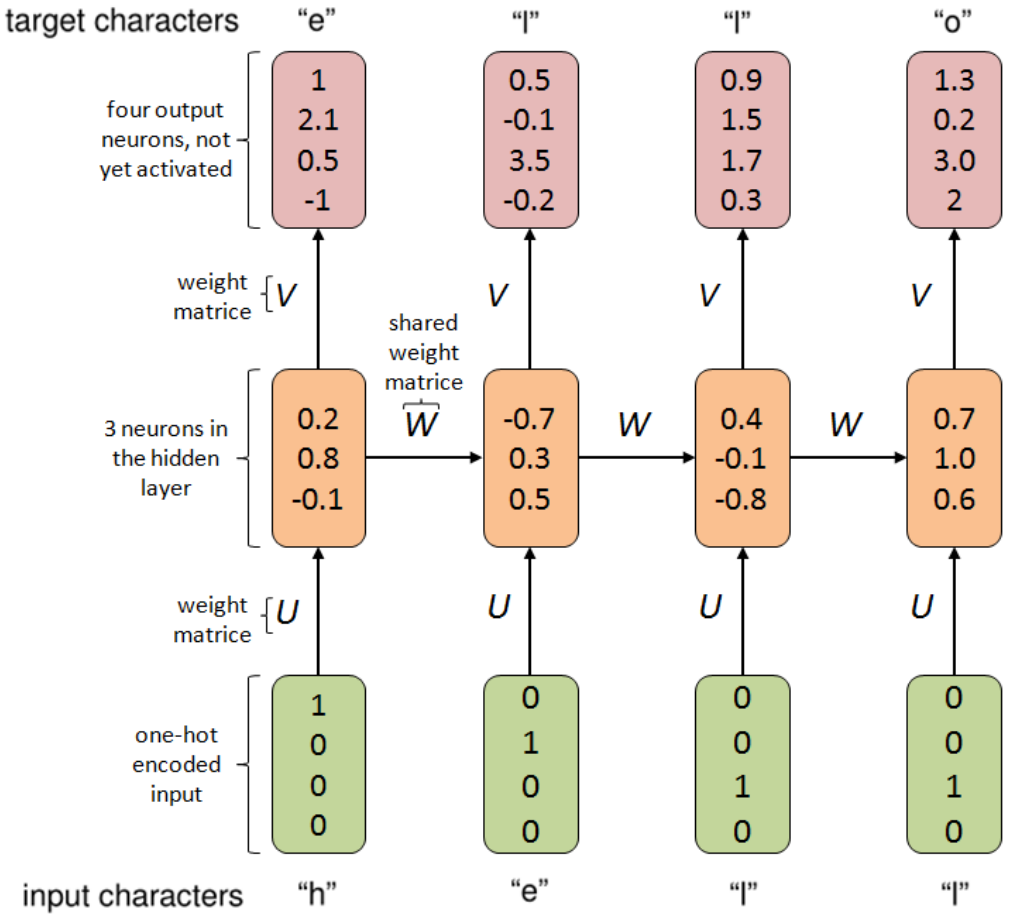
The probability of "l" should be likely, given in the context of "he"

RNNS - GENERATE SEQUENCES



The probability of "l" should **also** be likely, given in the context of "hel"

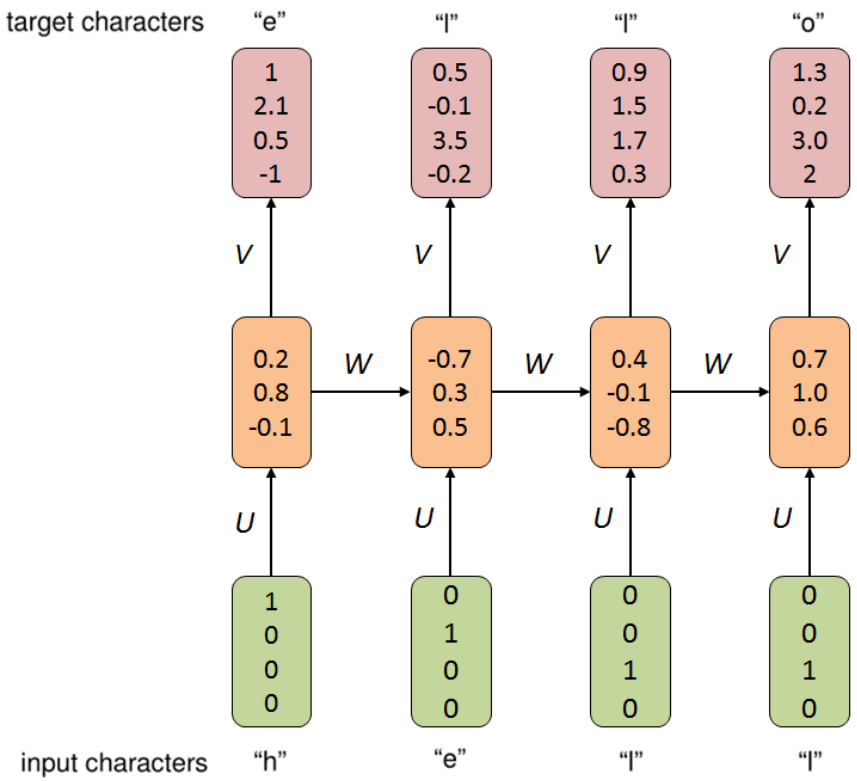
RNNS - GENERATE SEQUENCES



The probability of "o" should be likely, given the context of "hell"

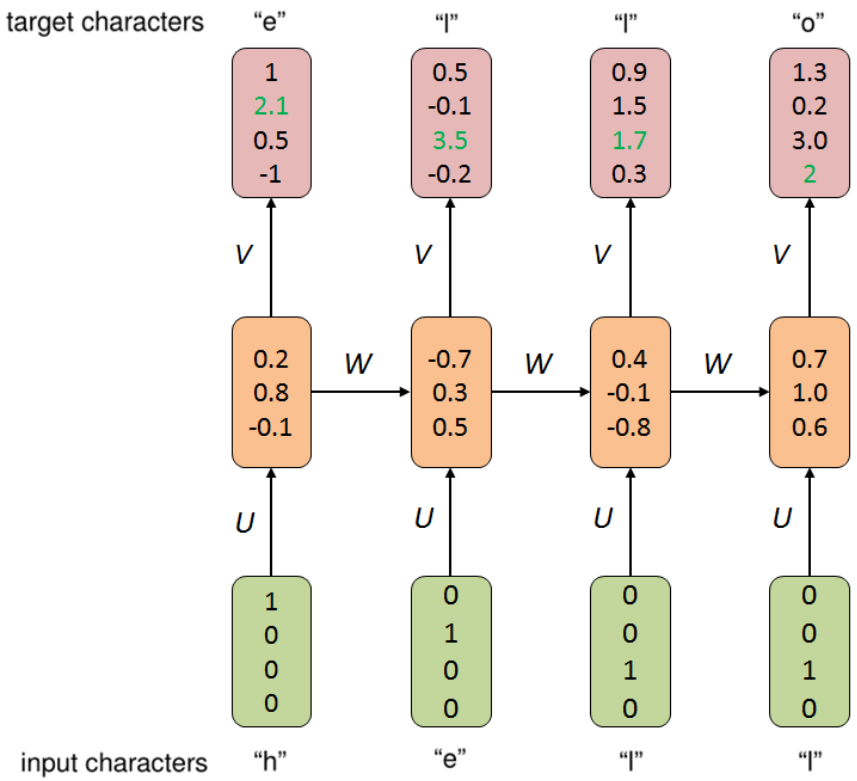
RNNS - GENERATE SEQUENCES

The RNN has a 4-dimensional input and output. The exemplary hidden layer consists of 3 neurons. This diagram shows the activations in the forward pass when the RNN is fed the characters “hell” as input. The output contains confidences the RNN assigns for the next character.



RNNS - GENERATE SEQUENCES

The RNN has a 4-dimensional input and output. The exemplary hidden layer consists of 3 neurons. This diagram shows the activations in the forward pass when the RNN is fed the characters “hell” as input. The output contains confidences the RNN assigns for the next character.



Our goal is to increase the confidence for the correct letters (green digits) and decrease the confidence of all others (we could also use a softmax activation to squash the digits to probabilities $\in [0, 1]$).

NATURAL LANGUAGE PROCESSING (NLP)

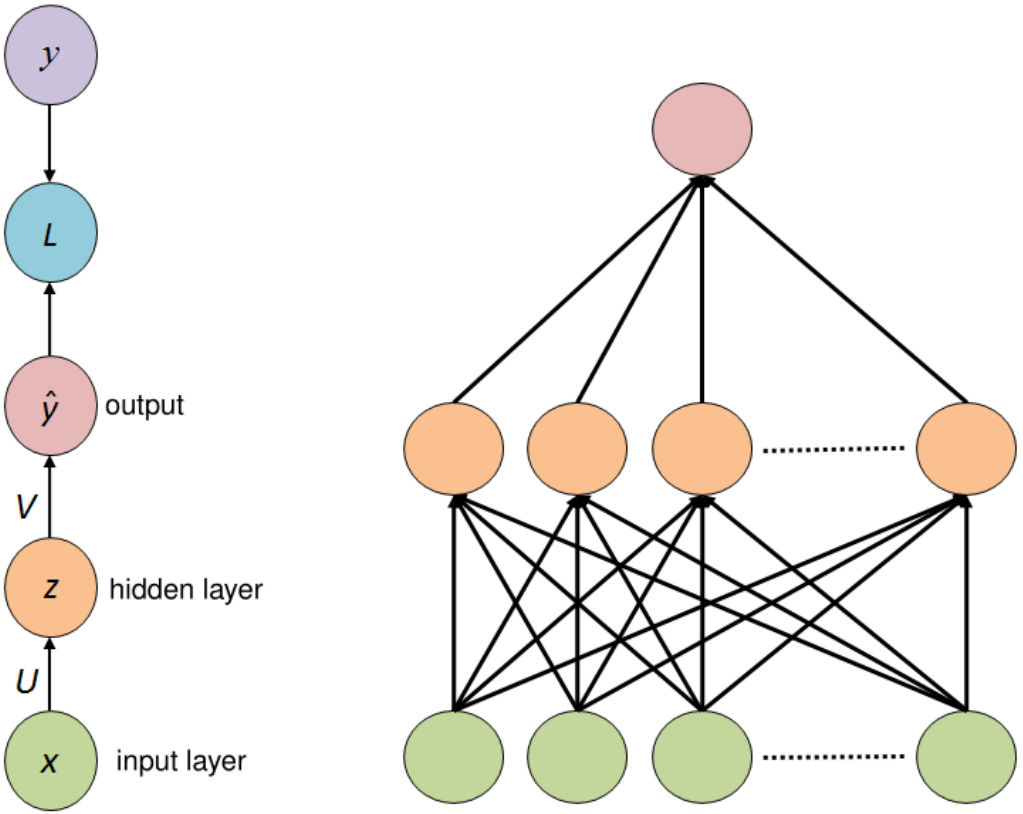
- What we just did is called natural language processing (NLP)
- Input: word/character, encoded as one-hot vector
- Output: probability distribution over words given previous words

$$P(y^{[1]}, \ldots, y^{[\tau]}) = \prod_{i=1}^{\tau} P(y^{[i]} | y^{[1]}, \ldots, y^{[i-1]})$$

$\Rightarrow$ given a sequence of previous characters, ask the RNN to model the probability distribution of the next character in the sequence!

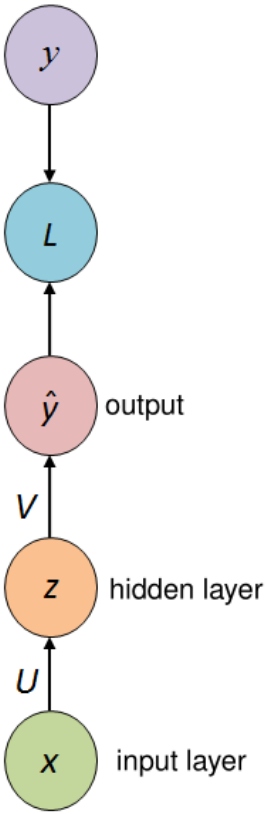
- Time to formalize RNNs...

RECURRENT NEURAL NETWORKS



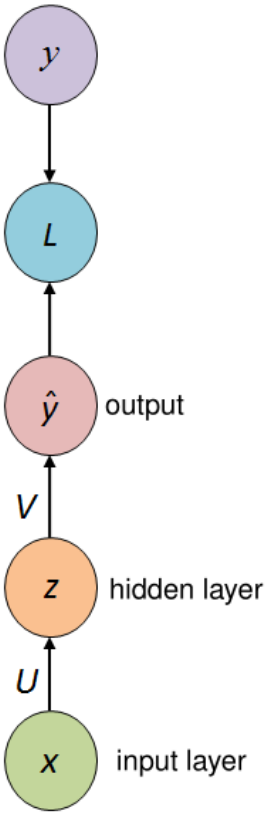
- On the left, the computational graph for a dense net. A loss function L measures how far each $\hat{y}$ is from the corresponding training target y. On the right, a potential unfolded network.

RECURRENT NEURAL NETWORKS



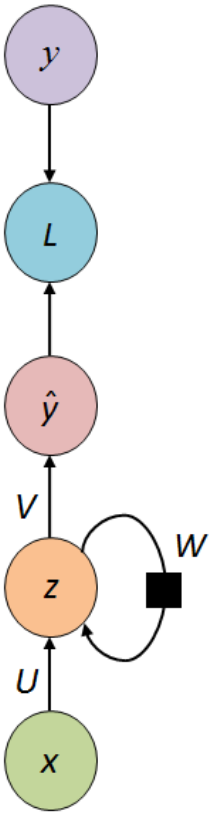
- In order to derive RNNs we have to extend our notation.

RECURRENT NEURAL NETWORKS



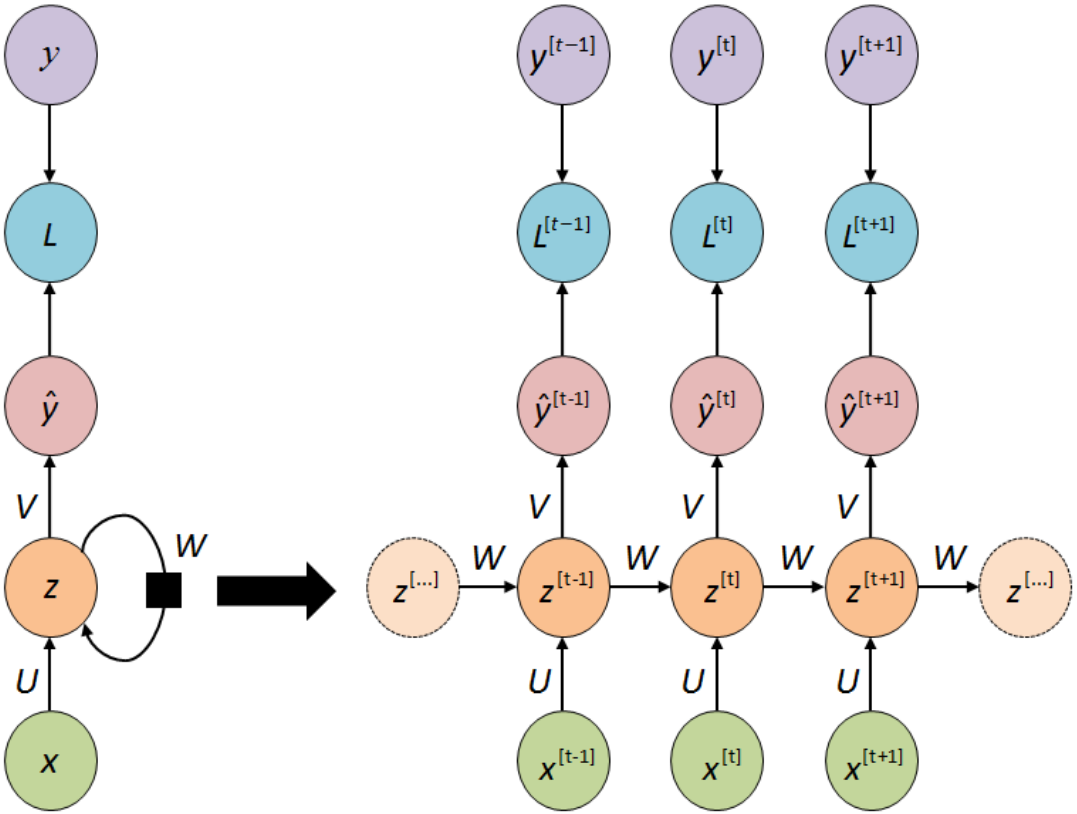
- In order to derive RNNs we have to extend our notation.
  - So far, we mapped some inputs x to outputs $\hat{y}$:
    $\hat{y} = \sigma(c + Vz) = \sigma(c + V\sigma(b + Ux))$
    ..with V and U being weight matrices.

RECURRENT NEURAL NETWORKS



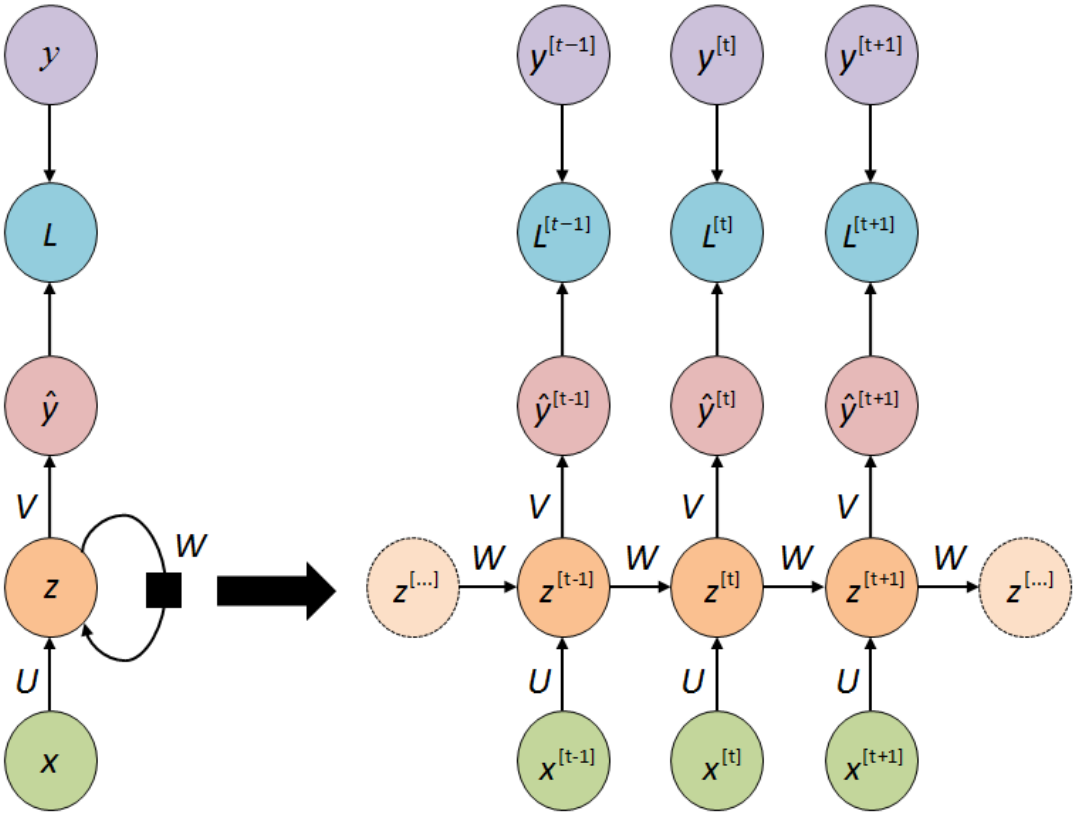
- A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.
- RNNs are networks with loops, allowing information to persist.

RECURRENT NEURAL NETWORKS



- Things might become more clear if we unfold the architecture.
- Instead of hidden layer, we call z the state of the system at time t.
- The state contains information about the whole past sequence.

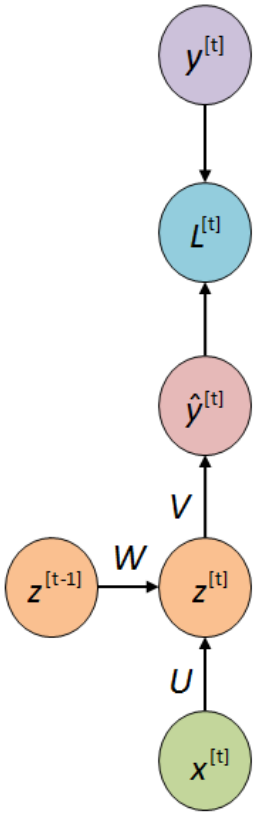
RECURRENT NEURAL NETWORKS



- We went from

$$\hat{y} = \sigma(c + V\sigma(b + Ux)) \text{ to}$$
$$\hat{y}^{[t]} = \sigma(c + V\sigma(b + Wz^{[t-1]} + Ux^{[t]}))$$

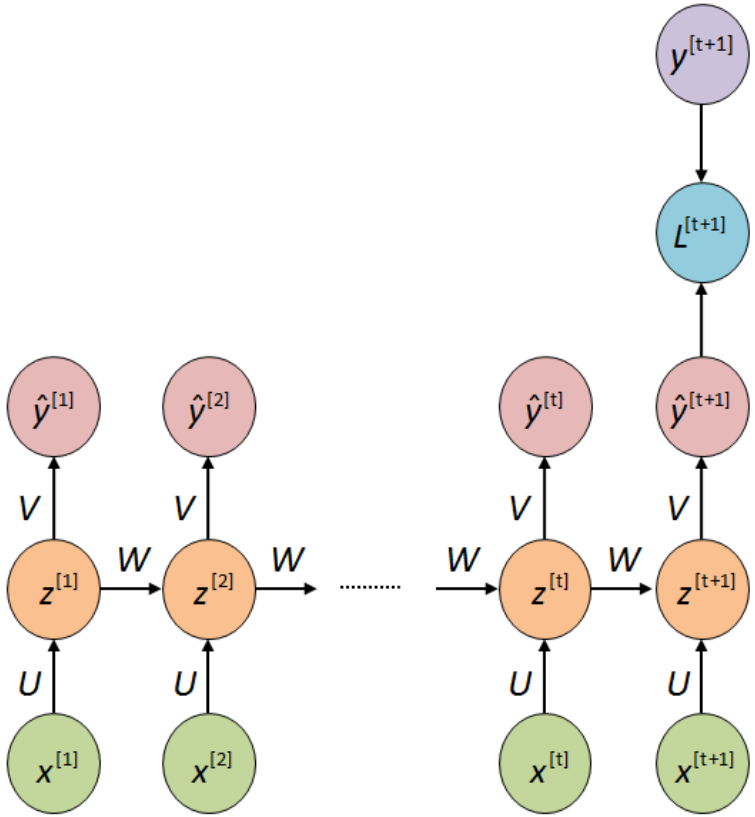
RECURRENT NEURAL NETWORKS



- Potential computational graph for time state t:

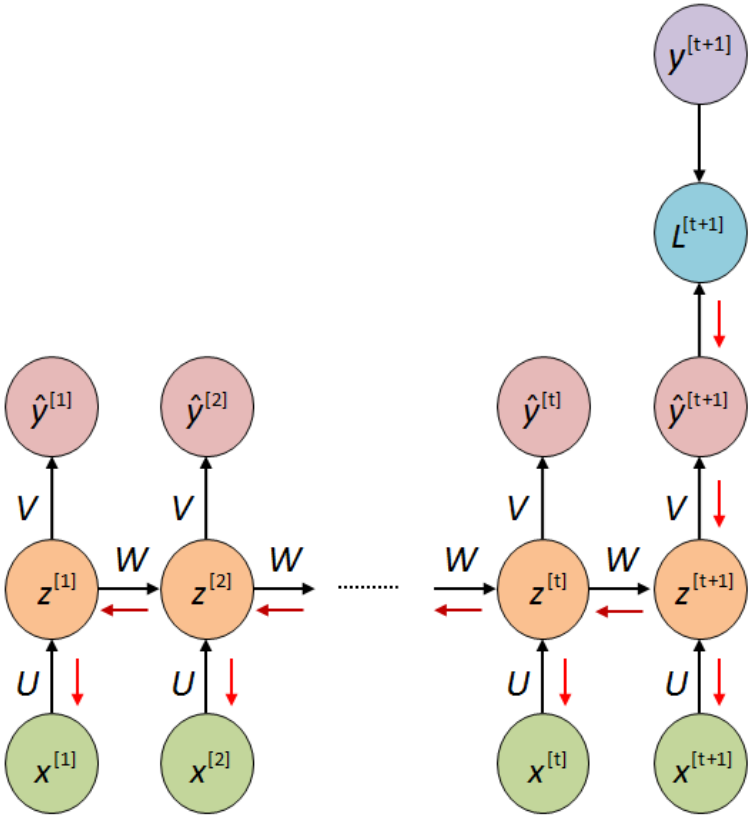
$$\hat{y}^{[t]} = \sigma(c + V\sigma(b + Wz^{[t-1]} + Ux^{[t]}))$$

BACKPROPAGATION THROUGH TIME



- To carry out backpropagation for an arbitrary RNN, we simply compute:

BACKPROPAGATION THROUGH TIME



- To carry out backpropagation for an arbitrary RNN, we simply compute:

$$\frac{\delta L}{\delta z^1} = \frac{\delta L}{\delta z^t} \frac{z^t}{\delta z^{t-1}} \cdots \frac{\delta z^2}{\delta z^1}$$

SOME MORE SOPHISTICATED APPLICATIONS

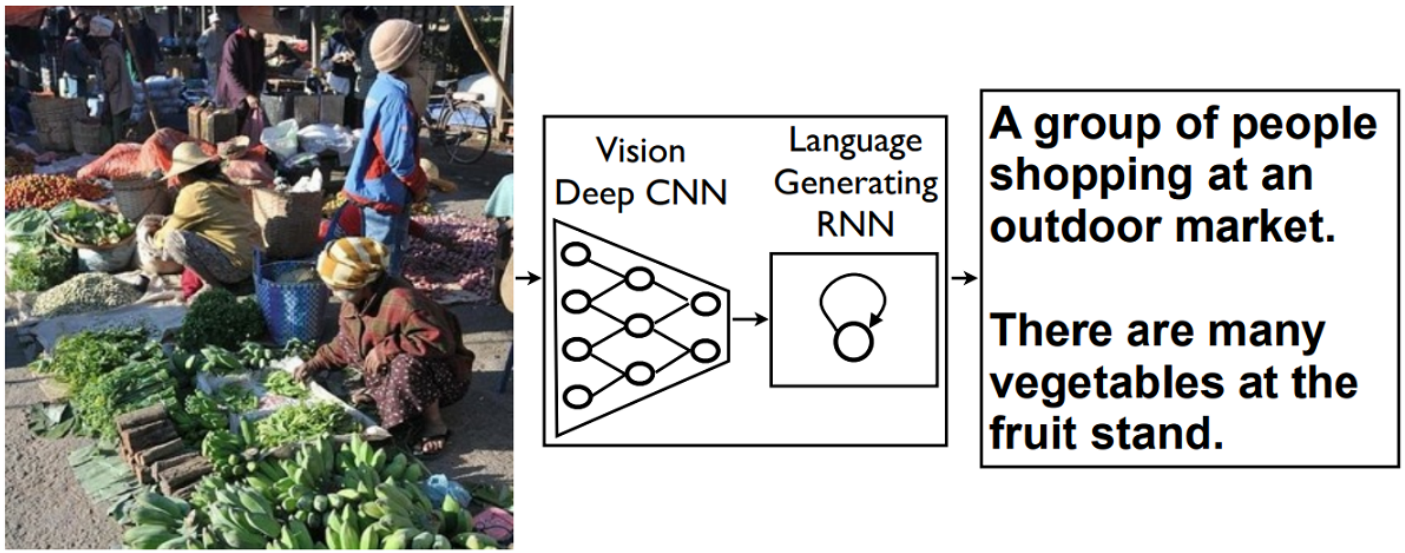


Figure: Show and Tell: A Neural Image Caption Generator (Oriol Vinyals et al. 2014). A language generating RNN tries to describe in brief the content of different images.

SOME MORE SOPHISTICATED APPLICATIONS

Figure: Show and Tell: A Neural Image Caption Generator (Oriol Vinyals et al. 2014). A language generating RNN tries to describe in brief the content of different images.

SOME MORE SOPHISTICATED APPLICATIONS

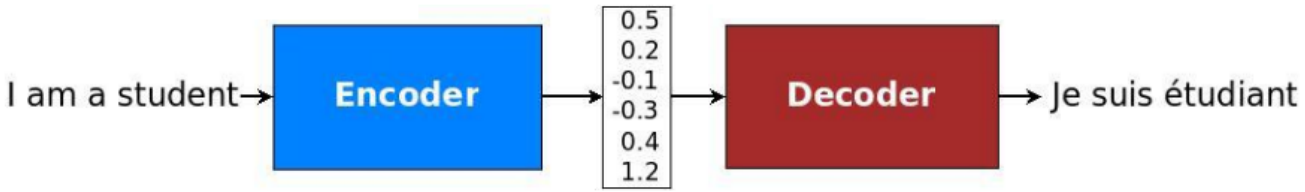


Figure: Neural Machine Translation (seq2seq): Sequence to Sequence Learning with Neural Networks (Ilya Sutskever et al. 2014). An encoder converts a source sentence into a “meaning” vector which is passed through a decoder to produce a translation.

SOME MORE SOPHISTICATED APPLICATIONS

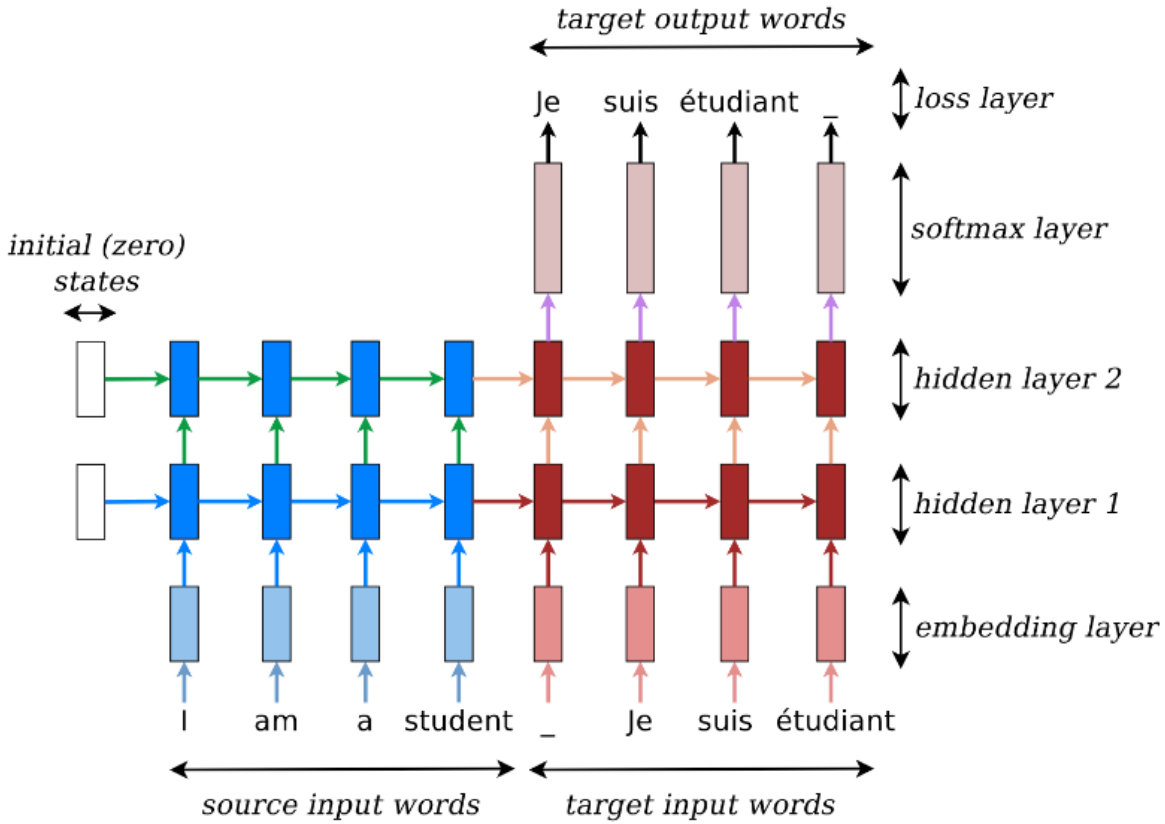


Figure: Neural Machine Translation (seq2seq): Sequence to Sequence Learning with Neural Networks (Ilya Sutskever et al. 2014). An encoder converts a source sentence into a “meaning” vector which is passed through a decoder to produce a translation.

SOME MORE SOPHISTICATED APPLICATIONS

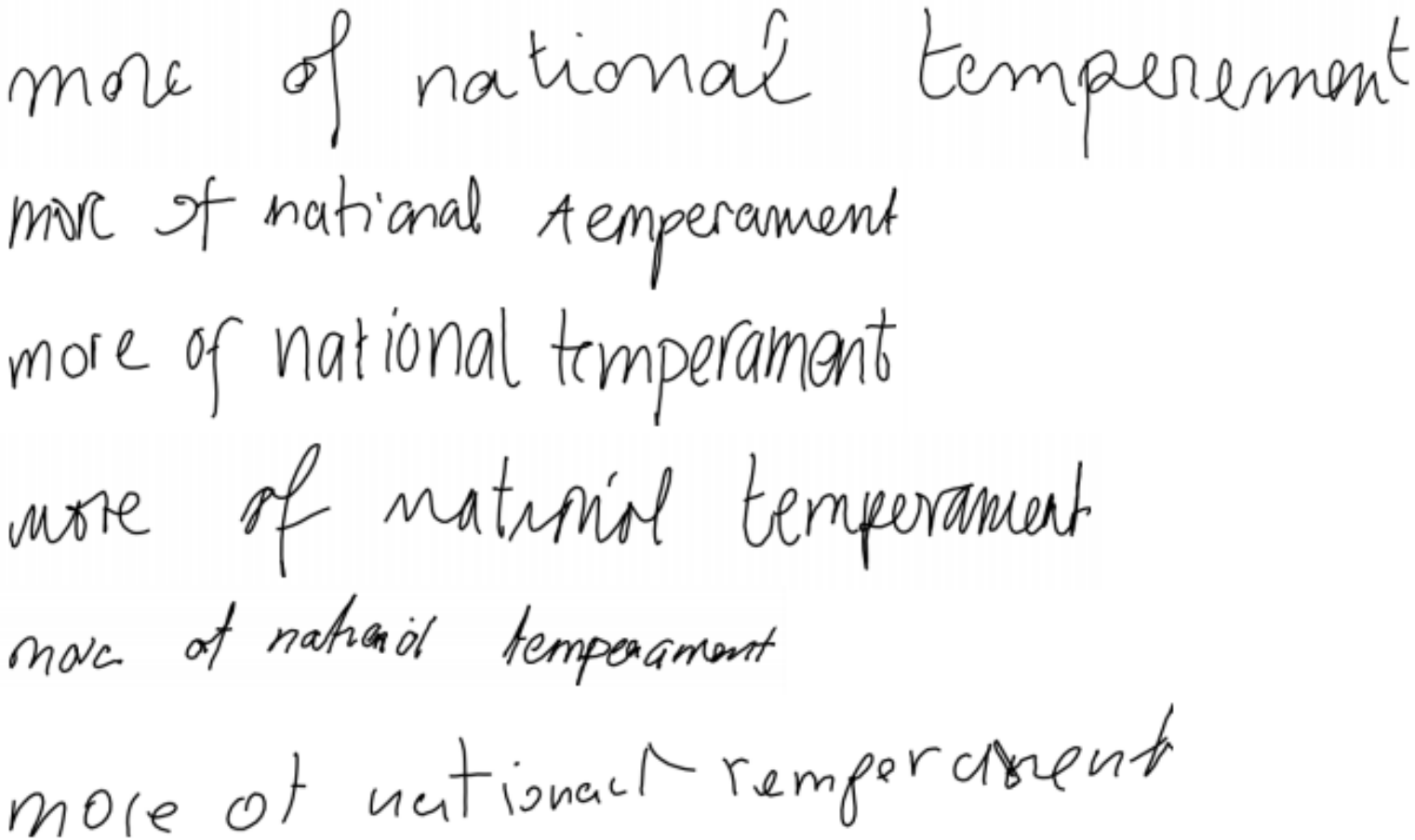


Figure: Generating Sequences With Recurrent Neural Networks (Alex Graves, 2013). Top row are real data, the rest are generated by various RNNs.

SOME MORE SOPHISTICATED APPLICATIONS

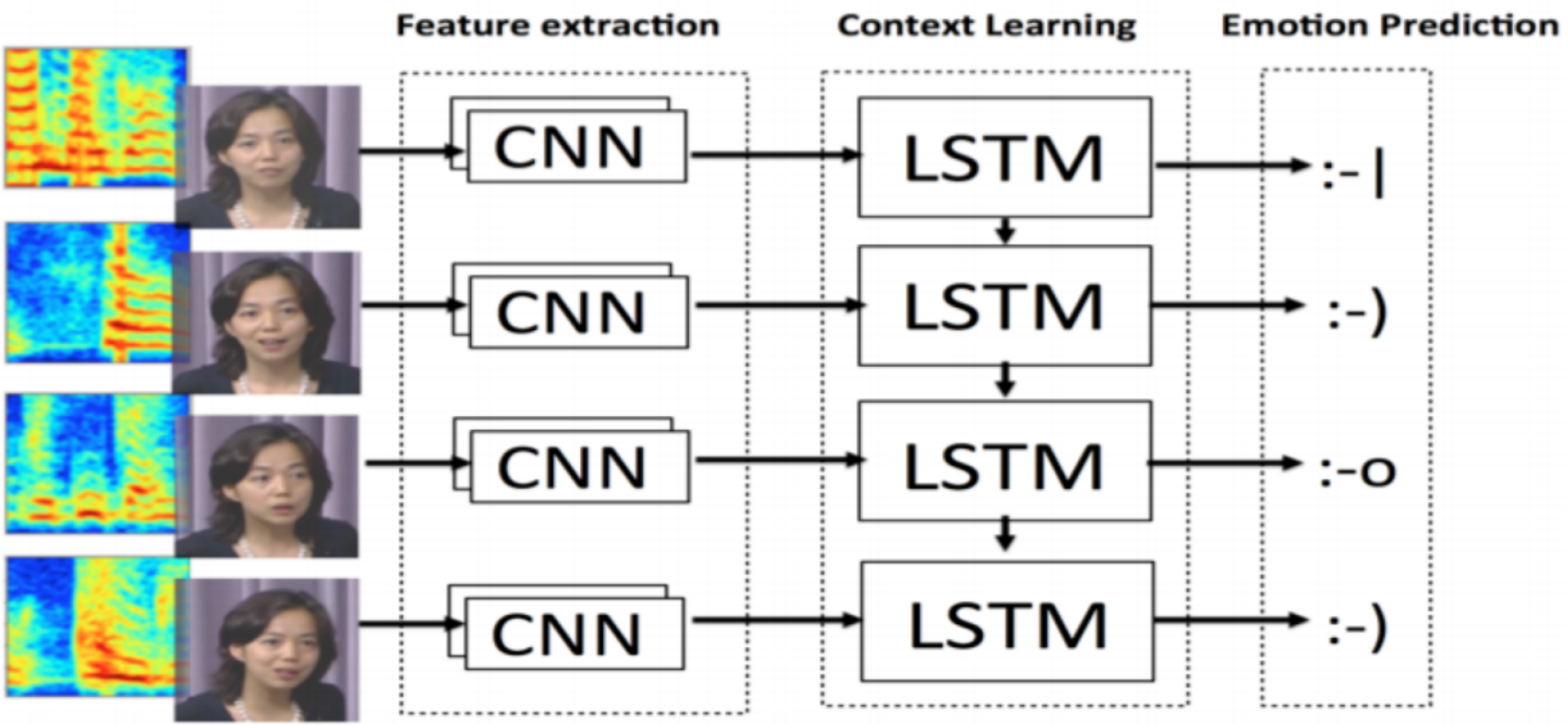


Figure: Convolutional and recurrent nets for detecting emotion from audio data (Namrata Anand & Prateek Verma, 2016). We already had this example in the CNN chapter!

SOME MORE SOPHISTICATED APPLICATIONS

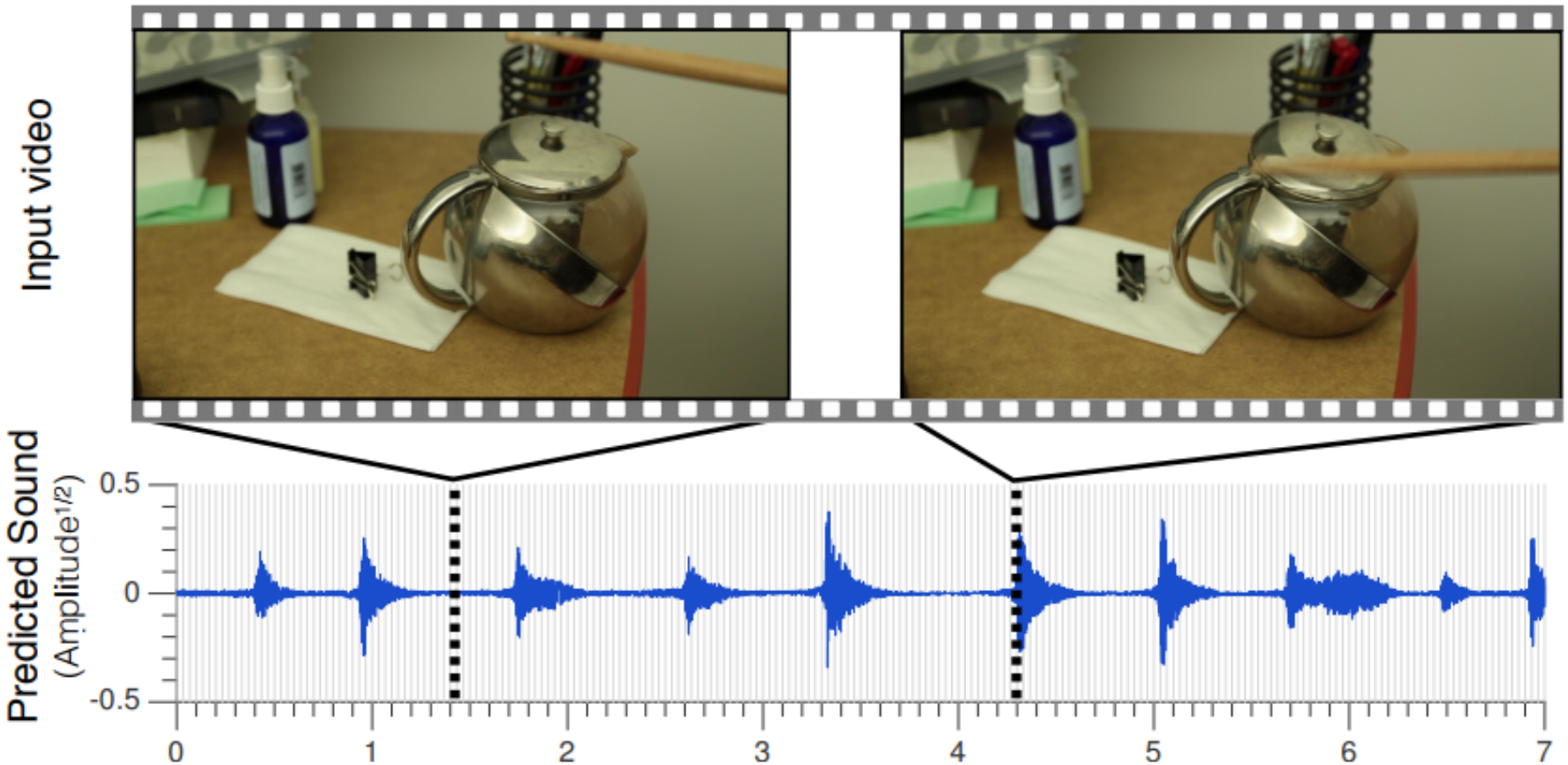


Figure: Visually Indicated Sounds (Andrew Owens et al. 2016). A model to synthesize plausible impact sounds from silent videos.

LONG-TERM DEPENDENCIES

- The challenge of learning long-term dependencies in RNNs often leads to gradients that 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.
- 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.

LONG-TERM DEPENDENCIES

- 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.
  - Even a huge change in the input will 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..).

LONG-TERM DEPENDENCIES

- 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.
- 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.

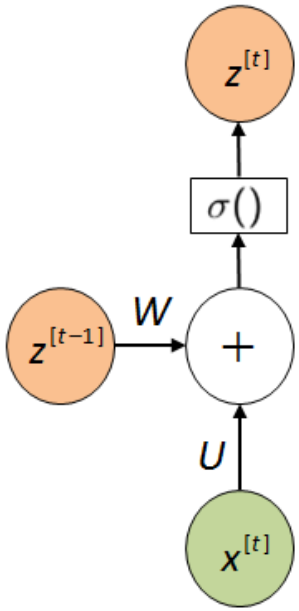
- To avoid exploding gradients, we simply clip the norm of the gradient at some threshold h (see chapter 3):

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

LONG SHORT-TERM MEMORY - LSTM

- The LSTM provides a different way of dealing with vanishing gradients.
- A cell state $s^{[t]}$ is introduced, which can be manipulated by different gates to forget old information, add new information and read information out of it.
- Each gate is a vector of the same size as the cell state and each element of the vector is a number between 0 and 1, with 0 meaning “let nothing pass” and 1 “let everything pass”.
- The gates are computed as a parametrized function of the previous hidden state $z^{[t-1]}$ and the input at the current time step $x^{[t]}$ multiplied by gate-specific weights and typically squashed through a sigmoid function into the range of $[0, 1]$.
- The cell state allows the recurrent neural network to keep information over long time ranges and therefore overcome the vanishing gradient problem.

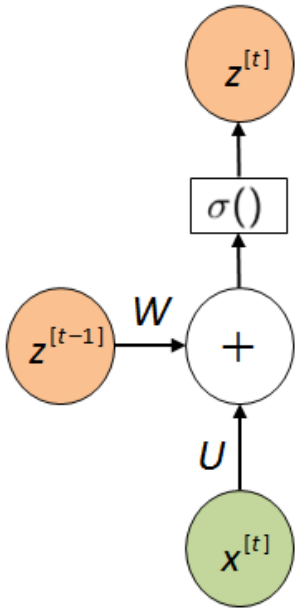
LONG SHORT-TERM MEMORY - LSTM

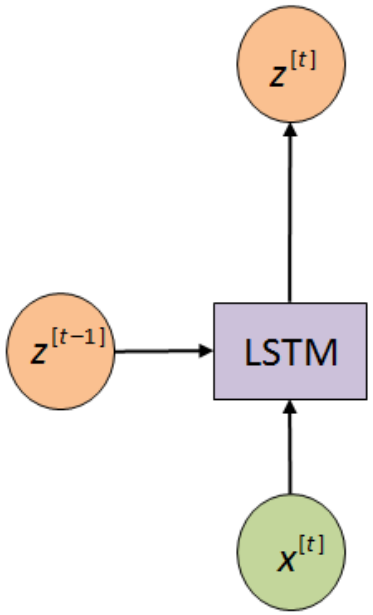


- Untill now, we simply computed

$$z^{[t]} = \sigma(b + Wz^{[t-1]} + Ux^{[t]})$$

LONG SHORT-TERM MEMORY - LSTM



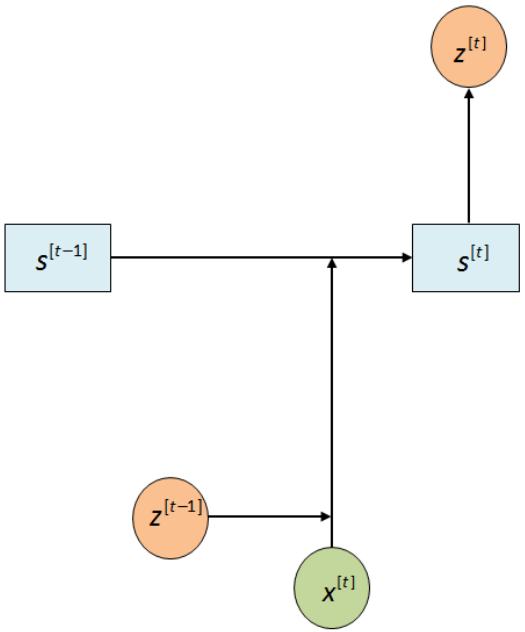


- Untill now, we simply computed

$$z^{[t]} = \sigma(b + Wz^{[t-1]} + Ux^{[t]})$$

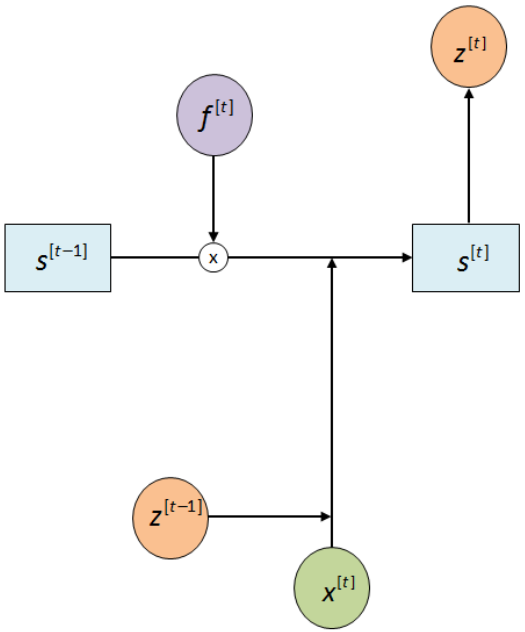
- Now we introduce the LSTM cell (this is where the fun begins)

LONG SHORT-TERM MEMORY - LSTM



- The key to LSTMs is the cell state $s^{[t]}$
- The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates

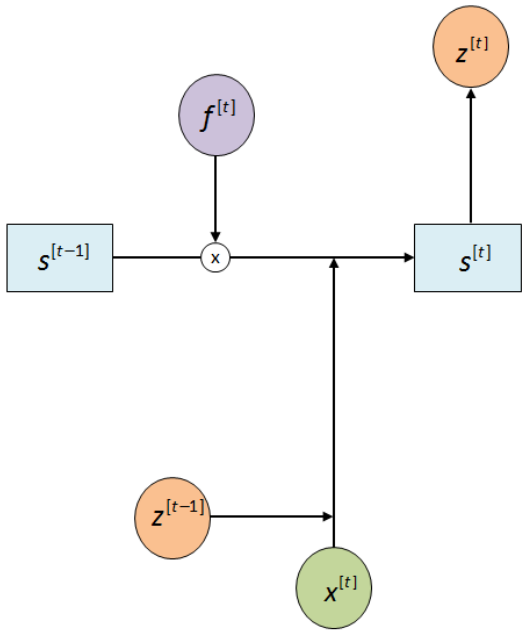
LONG SHORT-TERM MEMORY - LSTM



- The cell state $s^{[t]}$ is computed as a function of the previous cell state $s^{[t-1]}$, multiplied by the forget gate $f^{[t]}$.

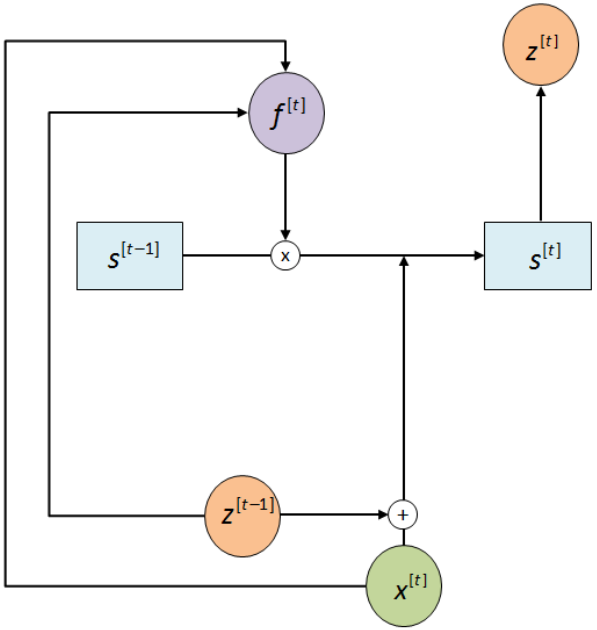
$$s^{[t]} = f^{[t]} s^{[t-1]}, \text{ with } f^{[t]} \in [0, 1]$$

LONG SHORT-TERM MEMORY - LSTM



- Think of a model trying to predict the next word based on all the previous ones. The cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old one.

LONG SHORT-TERM MEMORY - LSTM

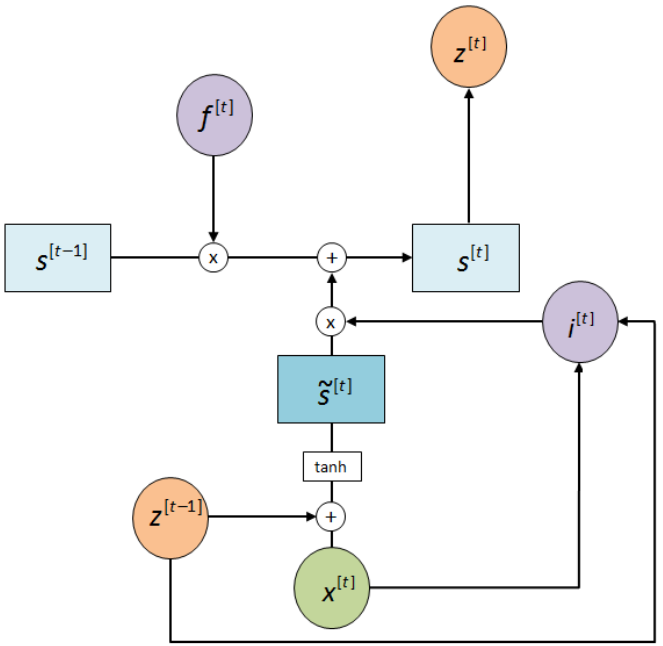


- We obtain the forget gate by computing

$$f^{[t]} = \sigma(b^f + W^f z^{[t-1]} + U^f x^{[t]})$$

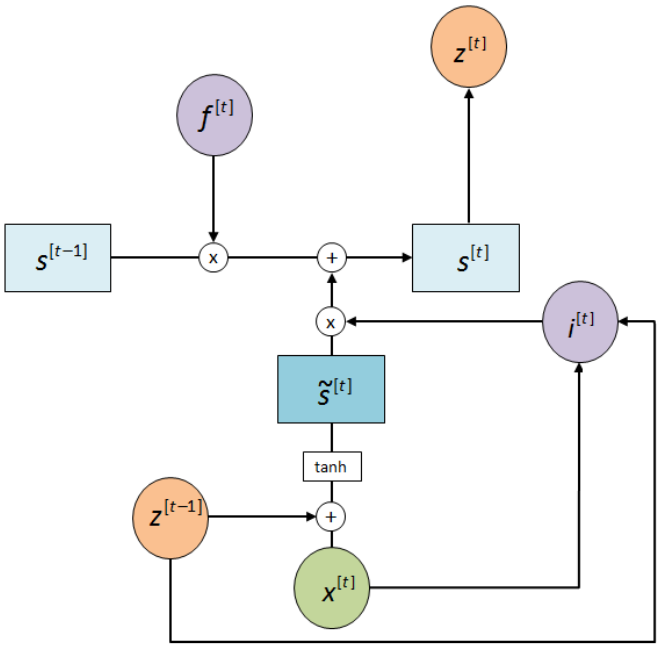
- $\sigma()$ is obviously a sigmoid, to squash the values to $[0, 1]$

LONG SHORT-TERM MEMORY - LSTM



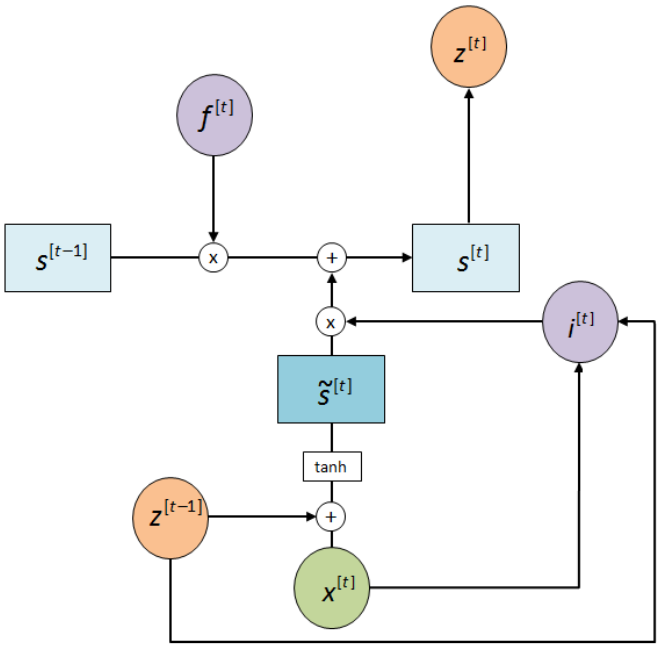
- Now it is time to add new information $i^{[t]}$ into the cell state.
- We also incorporate information of the previous hidden state $z^{[t-1]}$
- In the case of the language model, this is where we add the new information about the gender of our subject.

LONG SHORT-TERM MEMORY - LSTM



- The cell recurrent connection needs a function whose derivatives sustain for a long span to address the vanishing gradient problem
- $\tilde{s}^{[t]} = tanh(b + Wz^{[t-1]} + Ux^{[t]}) \in [-1, 1]$
- $i^{[t]} = \sigma(b^i + W^i z^{[t-1]} + U^i x^{[t]}) \in [0, 1]$

LONG SHORT-TERM MEMORY - LSTM

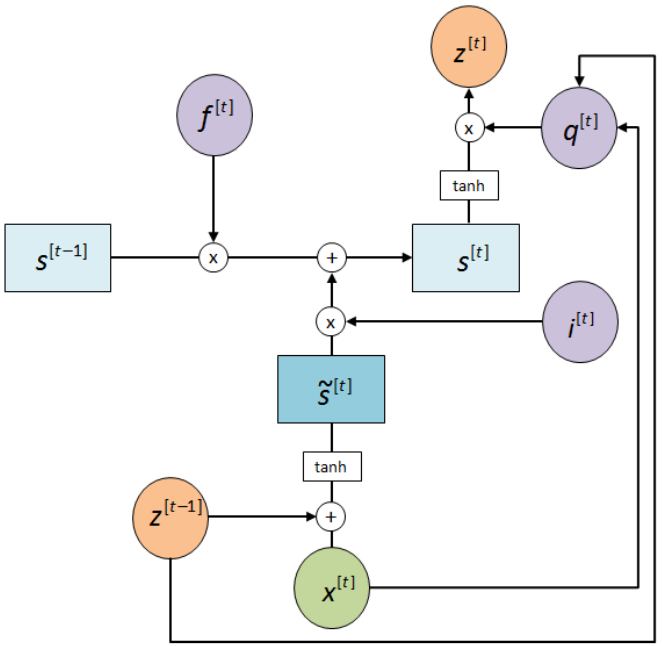


- Now we can finally update the cell state $s^{[t]}$:

$$s^{[t]} = f^{[t]} s^{[t-1]} + i^{[t]} \tilde{s}^{[t]}$$

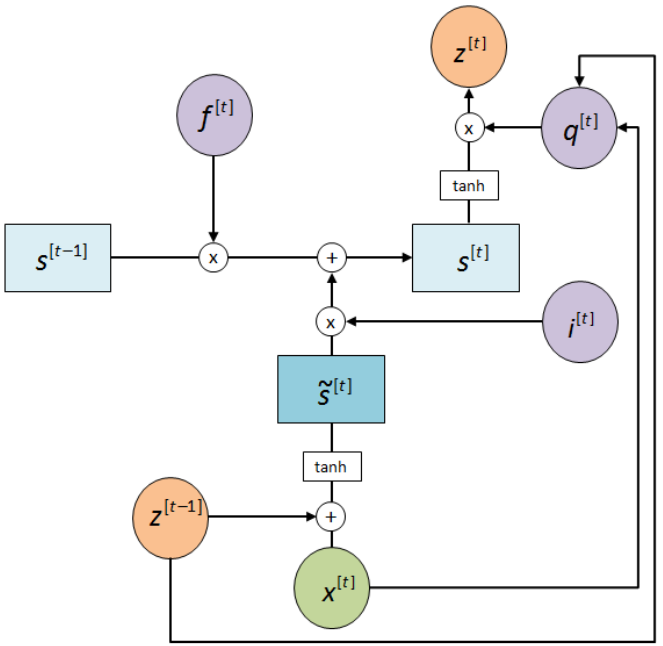
By the way: this does not mean that our lstm is complete!

LONG SHORT-TERM MEMORY - LSTM



- In order to complete the lstm cell, one final ingredient is missing
- The output will be a filtered version of our cell state
- First, we run a sigmoid layer which decides what parts of the cell state we're going to output: $q^{[t]} = \sigma(b^q + W^q z^{[t-1]} + U^q x^{[t]}$

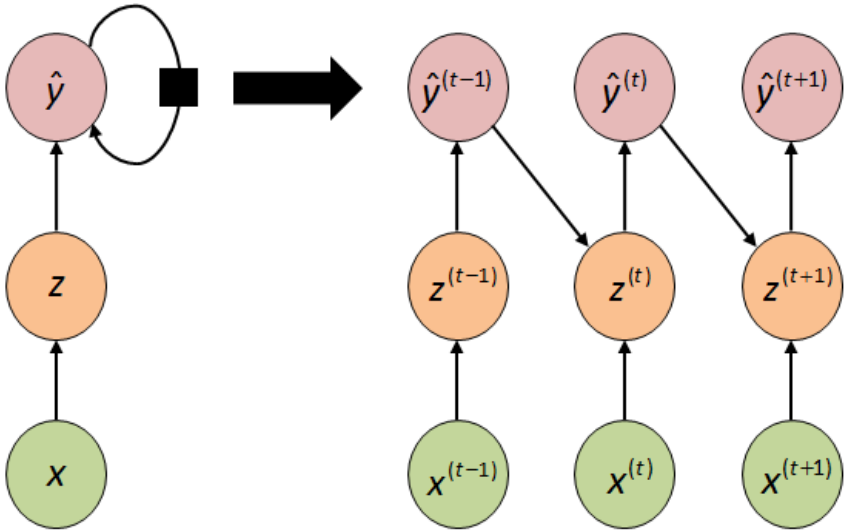
LONG SHORT-TERM MEMORY - LSTM



- Finally, the new state $z^{[t]}$ is then a function of the LSTM cell state, multiplied by the output gate:

$$z^{[t]} = q^{[t]} \cdot tanh(s^{[t]})$$

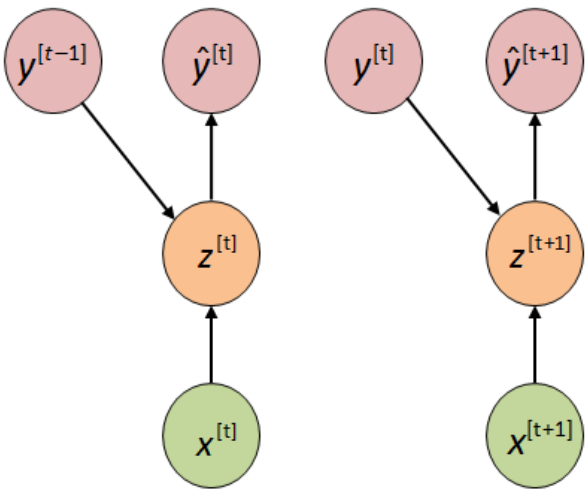
RNN WITH OUTPUT RECURRENCE



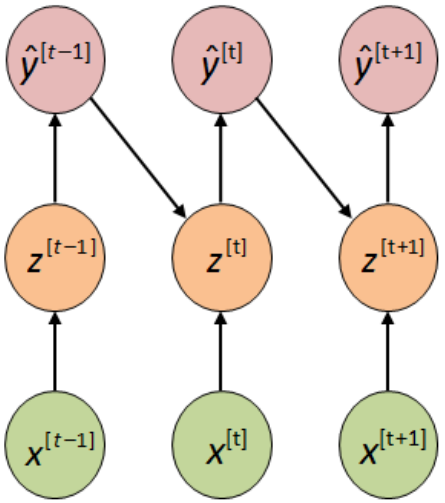
- Such an RNN is less powerful (can express a smaller set of functions).
- However, it may be easier to train because each time step it can be trained in isolation from the others, allowing greater parallelization during training.

TEACHER FORCING

- Teacher forcing is a procedure that emerges from the maximum likelihood criterion, in which during training the model receives the ground truth output $y^{[t]}$ as input at time $[t+1]$.



- At training time



- At testing time

ONE-OUTPUT RNN

- Recurrent Neural Networks do not need to have an output at each time step, instead they can only have outputs at a few time steps.
- A common variant is an RNN with only one output at the end of the sequence.
- Information from the whole input sequence is incorporated into the final hidden state, which is then used to create an output, e.g. a sentiment (“positive”, “neutral” or “negative”) for a movie review.
- Other applications of such an architecture are sequence labeling, e.g. classify an article into different categories (“sports”, “politics” etc.)

ONE-OUTPUT RNN

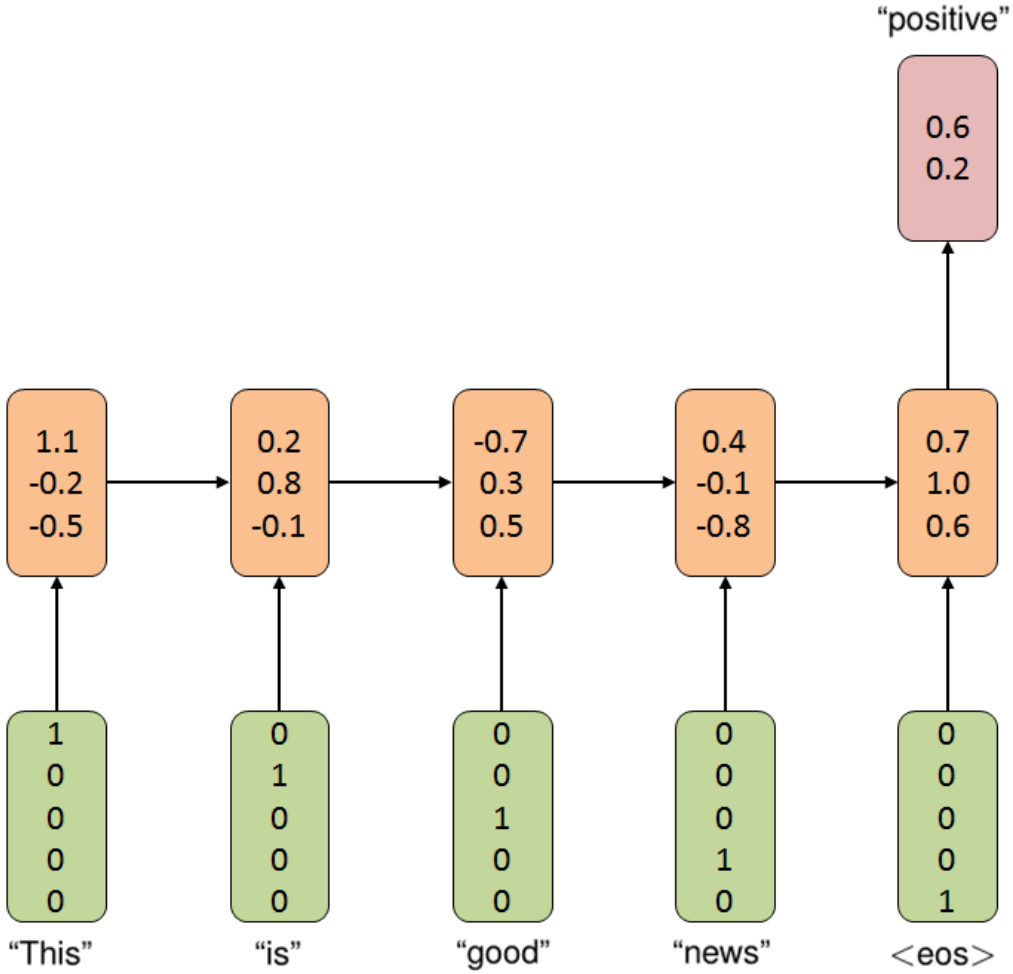


Figure: A Recurrent Neural Network with one output at the end of the sequence. Such a model can be used for sentiment analysis (<eos> = "end of sequence").

DEEP RNNS

- Another generalization of the simple RNN are bidirectional RNNs.
- These allow to process sequential data depending on both past and future inputs, e.g. an application predicting missing words, which probably depend on both preceding and following words.
- One RNN processes inputs in the forward direction from $x^{[1]}$ to $x^{[T]}$ computing a sequence of hidden states $(z^{[1]}, \ldots, z^{(T)})$, another RNN in the backward direction from $x^{[T]}$ to $x^{[1]}$ computing hidden states $(g^{[T]}, \ldots, g^{[1]})$
- Predictions are then based on both hidden states, which could be concatenated.
- With connections going back in time the whole input sequence must be known in advance to train and infer from the model.
- Bidirectional RNNs are often used for the encoding of a sequence in machine translation.

DEEP RNNS

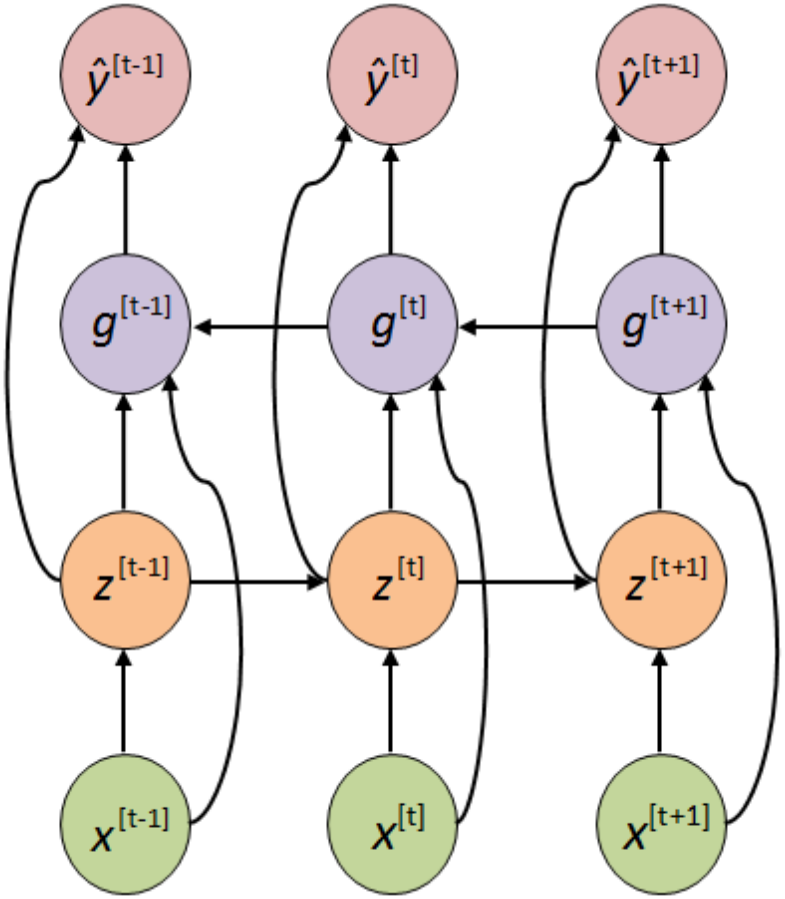


Figure: A bidirectional RNN consists of a forward RNN processing inputs from left to right and a backward RNN processing inputs backwards in time.

RECURSIVE NEURAL NETWORKS

- Recursive Neural Networks are a generalization of Recurrent Neural Networks.
- A tree structure instead of a chain structure is used for the computations of the RNN.
- A fixed set of weights is repeatedly applied to the nodes of the tree.
- Recursive neural networks have been successfully applied to sentiment analysis!

RECURSIVE NEURAL NETWORKS

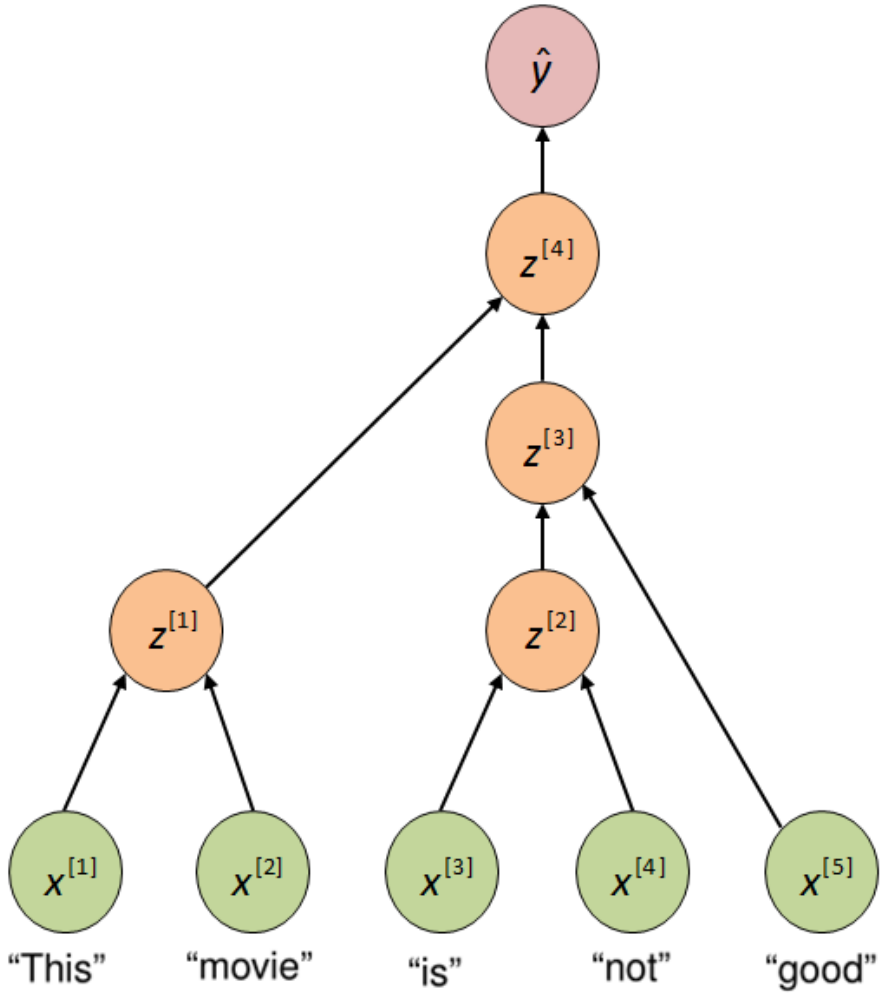


Figure: A recursive neural network

ENCODER-DECODER NETWORK

- Standard RNNs operate on input and output sequences of the same length.
- But for many interesting tasks such as question answering or machine translation the network needs to map an input sequence to an output sequence of different length.
- This is what an encoder-decoder (also called sequence-to-sequence architecture) enables us to do!
- An input/encoder-RNN processes the input sequence of length n_x and omits a fixed-length context vector C, usually the final hidden state or a simple function thereof.
- One time step after the other information from the input sequence is processed, added to the hidden state and passed forward in time through the recurrent connections between hidden states in the encoder.

ENCODER-DECODER NETWORK

- The context summarizes important information from the input sequence, e.g. the intent of a question in an question answering task or the meaning of a text in the case of machine translation.
- The decoder RNN uses this information to predict the output, a sequence of length n_y, which could vary from n_x.
- In natural language processing the decoder is a language model with recurrent connections between the output at one time step and the hidden state at the next time step as well as recurrent connections between the hidden states:

$$p(y^{[1]}, \ldots, y^{[y_n]} | x^{[1]}, \ldots, x^{[x_n]}) = \prod_{t=1}^{n_y} p(y^{[t]} | C; y^{[1]}, \ldots, y^{[t-1]})$$

  with C being the context-vector.
- This architecture is now jointly trained to minimize the translation error given a source sentence.

ENCODER-DECODER NETWORK

- Each conditional probability is then

$$p(y^{[t]}|y^{[1]},\ldots,y^{[t-1]};C) = f(y^{[t-1]},g^{[t]},C)$$

  where f is a non-linear function, e.g. the tanh and $g^{[t]}$ is the hidden state of the decoder network.

- Encoder-decoder architectures are often used for machine translation, where they excel phrasebased translation models.

ENCODER-DECODER NETWORK

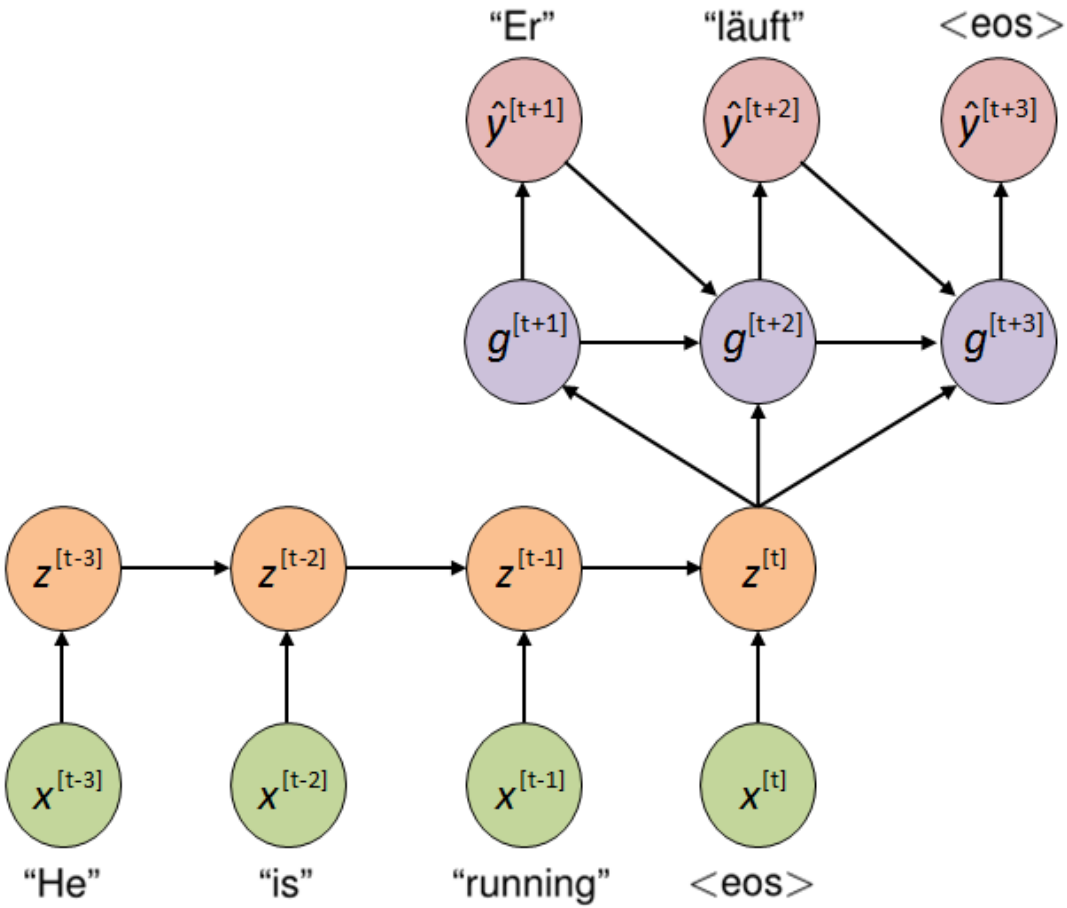


Figure: Encoder-decoder allows an RNN to process different length input and output sequences. In the first part of the network information from the input is encoded in the context, here the final hidden state, which is then passed on to every hidden state of the decoder, which produces the target sequence.

ATTENTION

- In a classical decoder-encoder RNN all information about the input sequence must be incorporated into the final hidden state, which is then passed as an input to the decoder network.
- With a long input sequence this fixed-sized context vector is unlikely to capture all relevant information about the past.
- It allows the decoder network to focus on different parts of the input sequence by adding connections from all hidden states of the encoder to each hidden state of the decoder
- At each point in time a set of weights is computed, which determine, how to combine the hidden states of the encoder into a context vector c_i, which holds the necessary information to predict the correct output.

ATTENTION

- Each hidden state contains mostly information from recent inputs, in the case of a bidirectional RNN to encode the input sequence, a hidden state contains information from recent preceding and following inputs.

ATTENTION

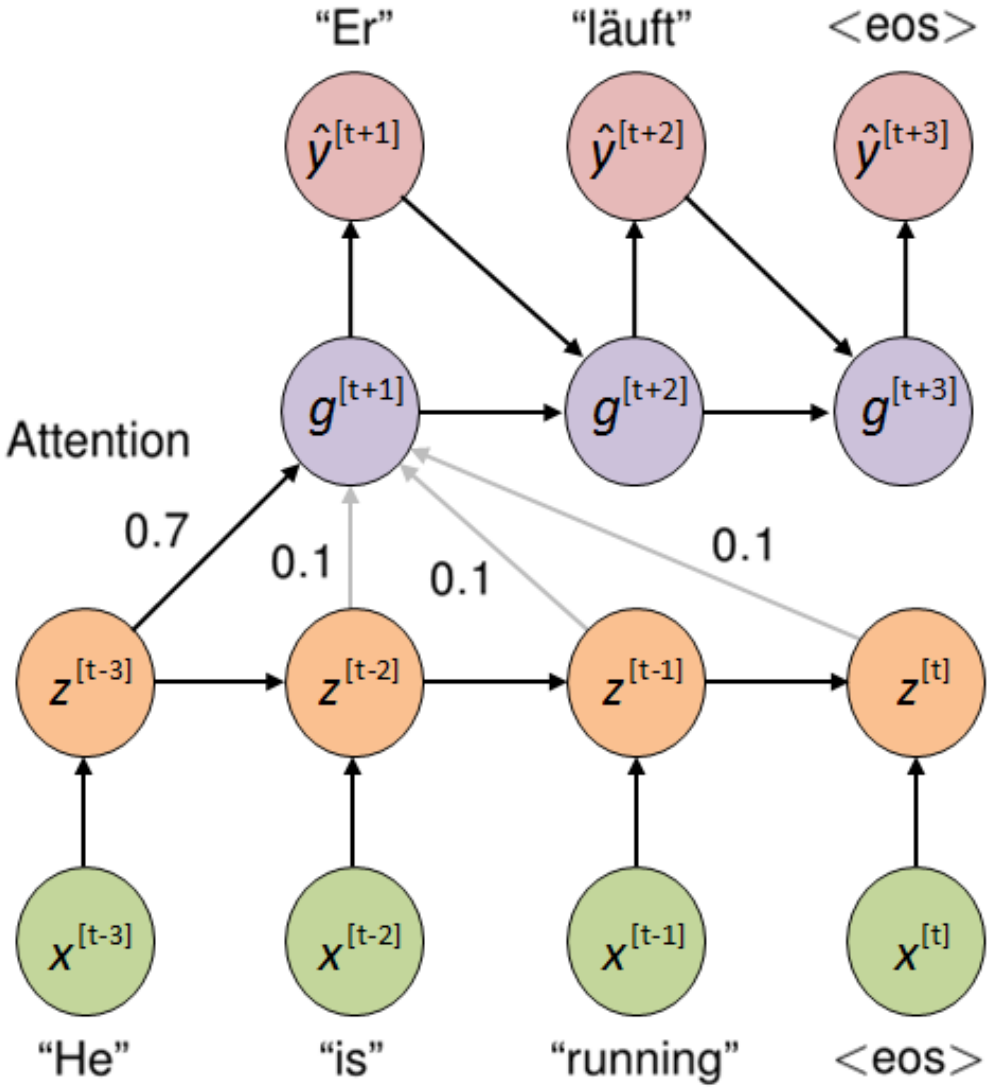


Figure: Attention at $t = t + 1$

ATTENTION

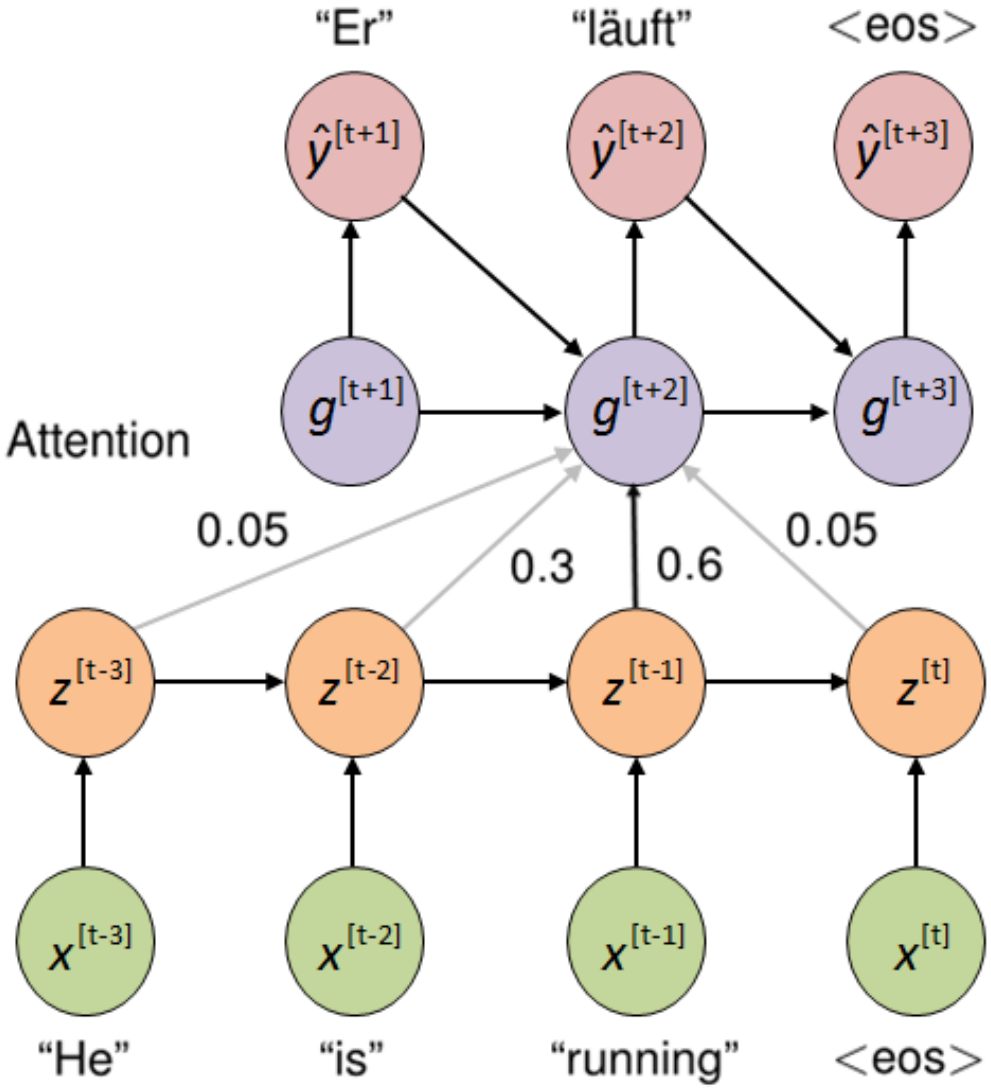


Figure: Attention at $t = t + 2$

REFERENCES

Ian Goodfellow, Yoshua Bengio and Aaron Courville (2016)

Deep Learning

http://www.deeplearningbook.org/

Oriol Vinyals, Alexander Toshev, Samy Bengio and Dumitru Erhan (2014)

Show and Tell: A Neural Image Caption Generator

https://arxiv.org/abs/1411.4555

Alex Graves (2013)

Generating Sequences With Recurrent Neural Networks

https://arxiv.org/abs/1308.0850

Namrata Anand and Prateek Verma (2016)

Convolutional and recurrent nets for detecting emotion from audio data

http://cs231n.stanford.edu/reports/2015/pdfs/Cs_231n_paper.pdf

REFERENCES

Andrew Owens, Phillip Isola, Josh H. McDermott, Antonio Torralba, Edward H. Adelson and William T. Freeman (2015)

Visually Indicated Sounds

https://arxiv.org/abs/1512.08512