LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Deep Learning

Chapter 8: Recurrent Neural Networks (RNNs)

Bernd Bischl

Department of Statistics – LMU Munich

Winter term 2020

MOTIVATION FOR RNNS

- The two major types of neural network architectures that we have 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.
- However, 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.

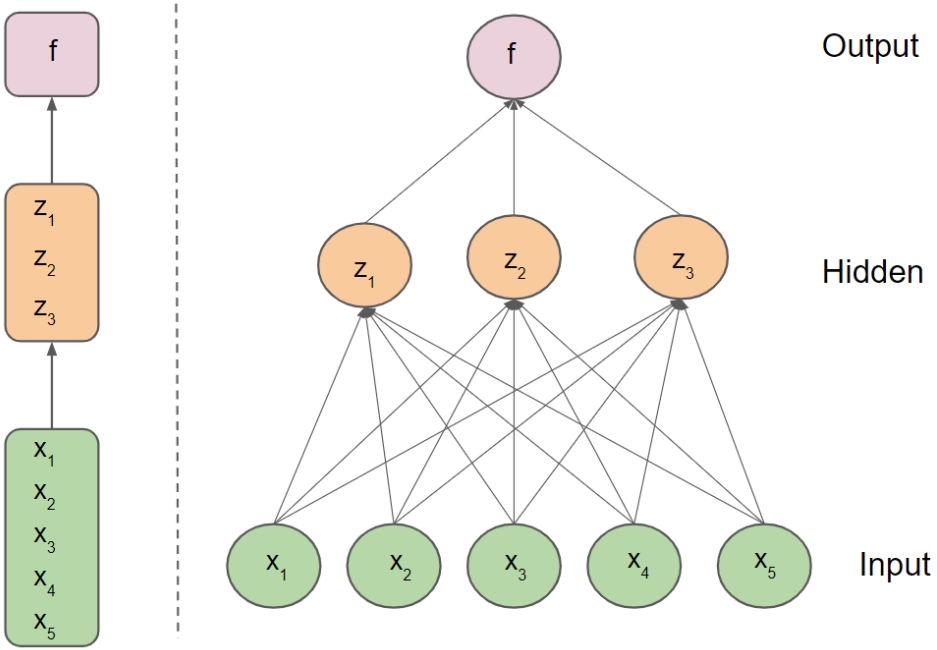


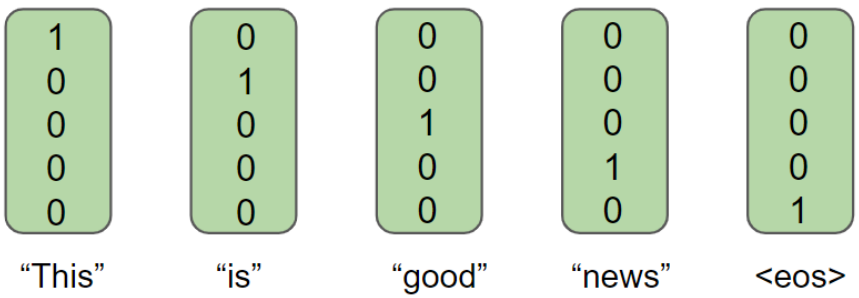
Figure: Two equivalent visualizations of a dense net with a single hidden layer.

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've 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

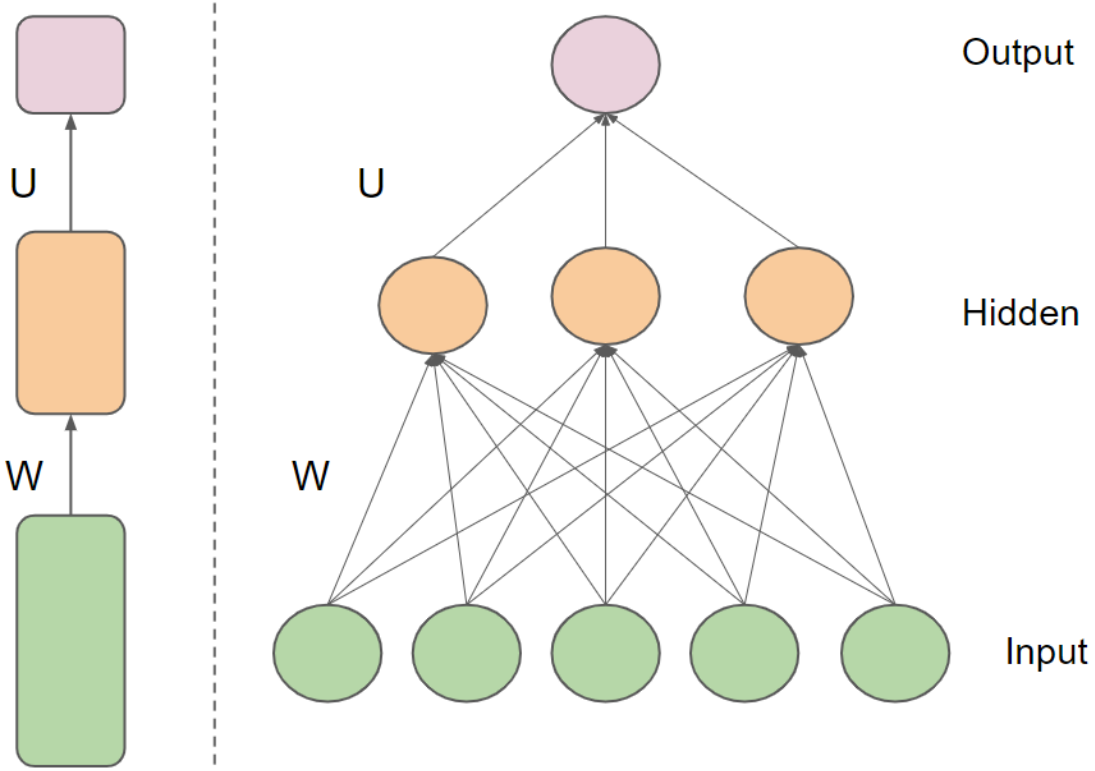
- When words in a sentence are fed to the network one at a time, the inputs are no longer independent. For example, it is much more likely that the word "good" is followed by "morning" rather than "plastic". We need to model this dependence.
- Each word must still be encoded as a fixed-length vector 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** 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.

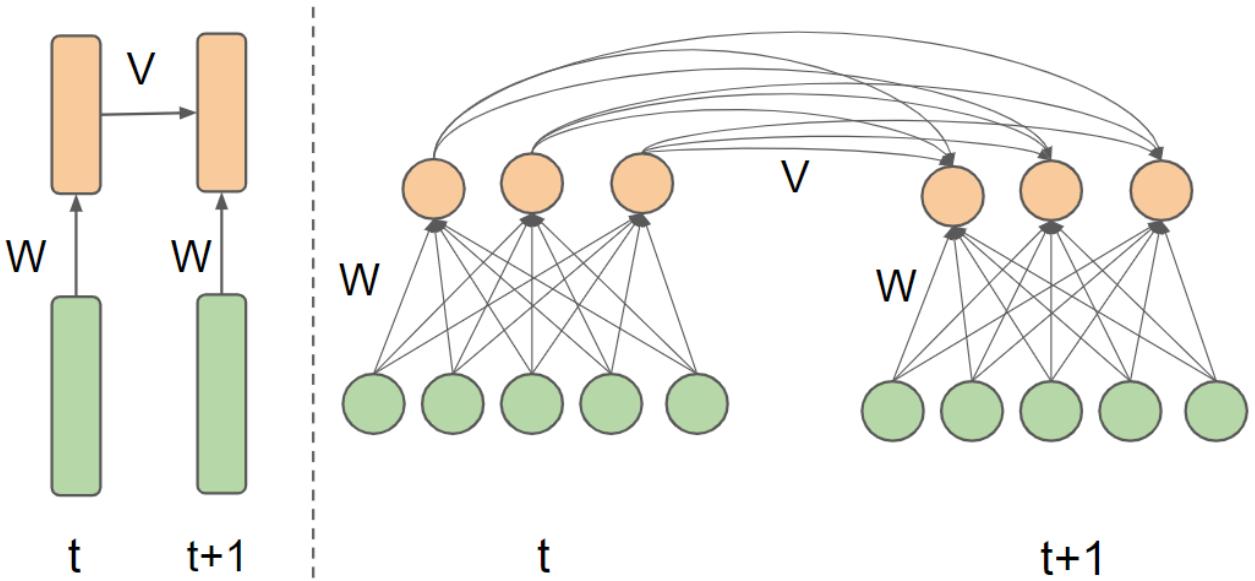


Figure: Input-to-hidden weights W and **hidden-to-hidden** weights V. The hidden-to-output weights U are not shown in the figure.

RNNS - INTRODUCTION

- With this additional set of hidden-to-hidden weights 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 W (and bias b).

$$z = \sigma(W^T x + b)$$

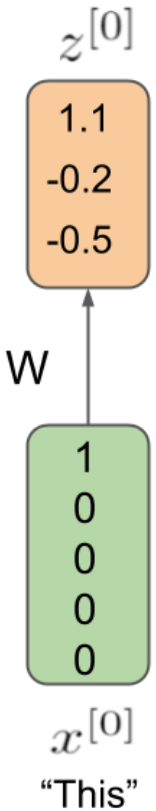
- In an RNN, the activations of the hidden layer (at time-step t) are computed using *both* the input-to-hidden weights W and the hidden-to-hidden weights V.

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

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

APPLICATION EXAMPLE - SENTIMENT ANALYSIS

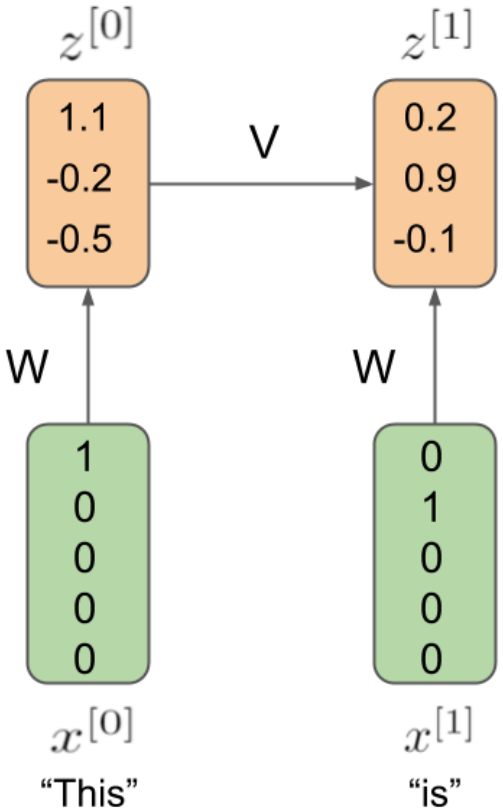
- At $t = 0$, we feed the word "This" to the network and obtain $z^{[0]}$.
- $z^{[0]} = \sigma(W^T x^{[0]} + b)$



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

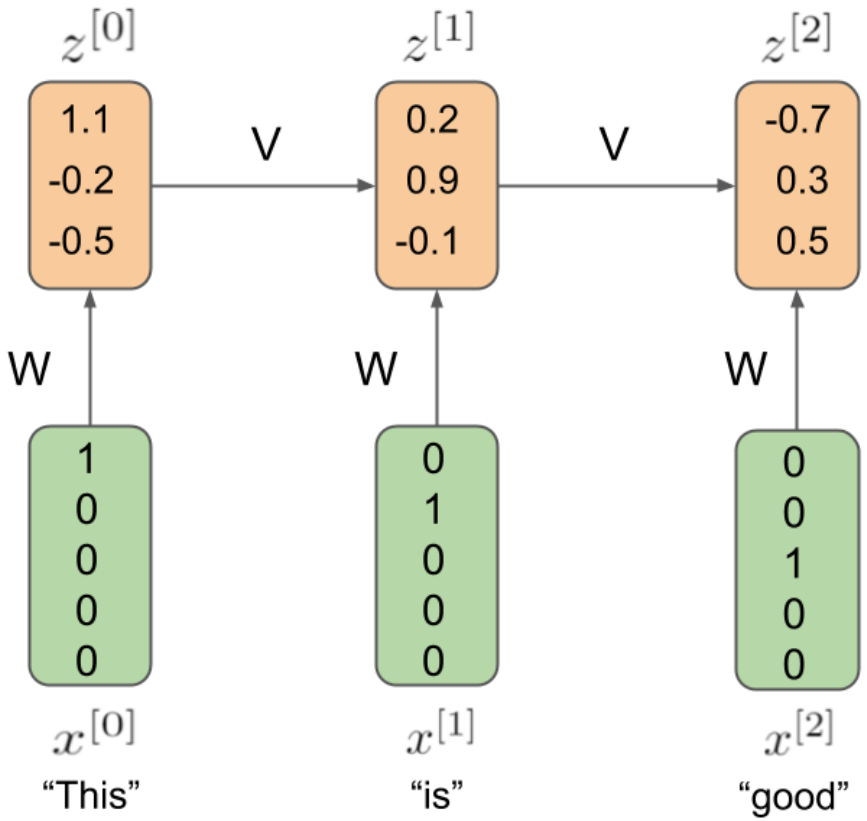
APPLICATION EXAMPLE - SENTIMENT ANALYSIS

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



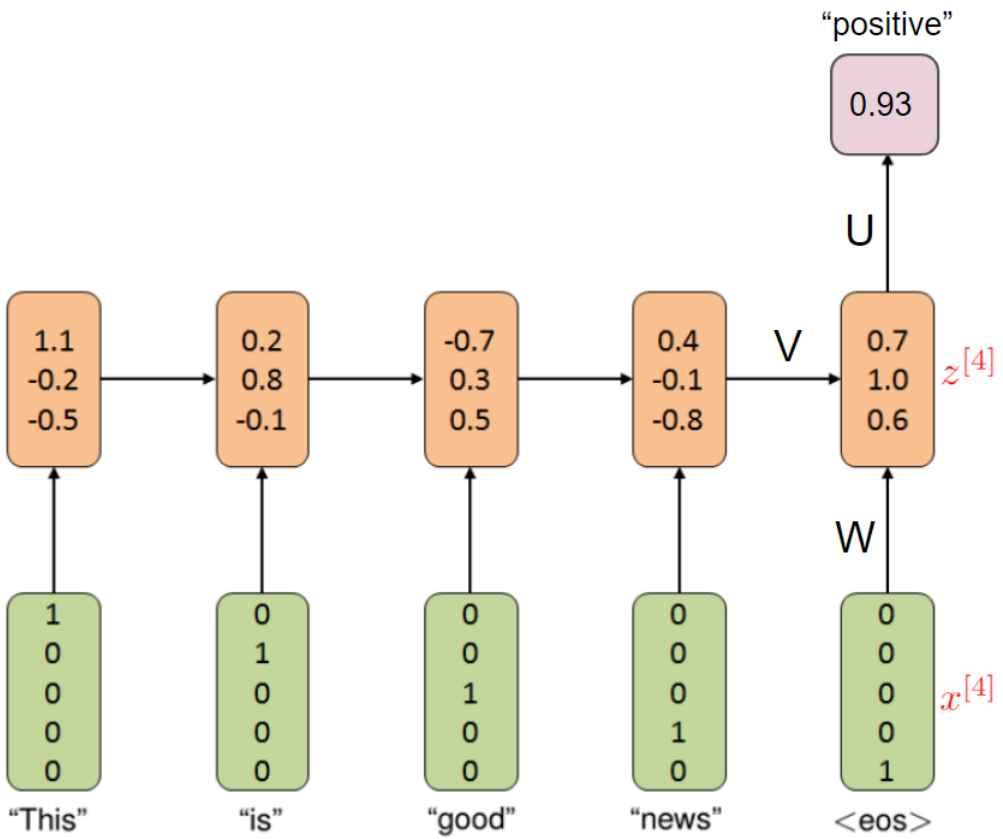
APPLICATION EXAMPLE - SENTIMENT ANALYSIS

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



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).
- $f = \sigma(U^T z^{[4]} + c)$, where c is the bias of the output neuron.

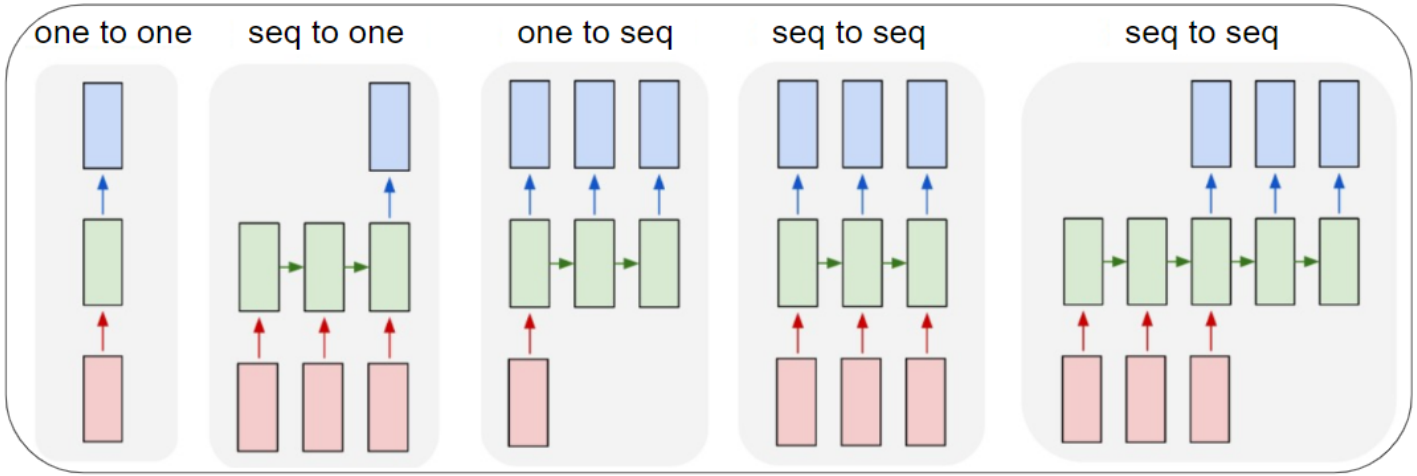


PARAMETER SHARING

- 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's important to note that no matter how long the input sequence is, the matrices W and 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.

RNNS - USE CASE SPECIFIC ARCHITECTURES

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



credit: Andrej Karpathy

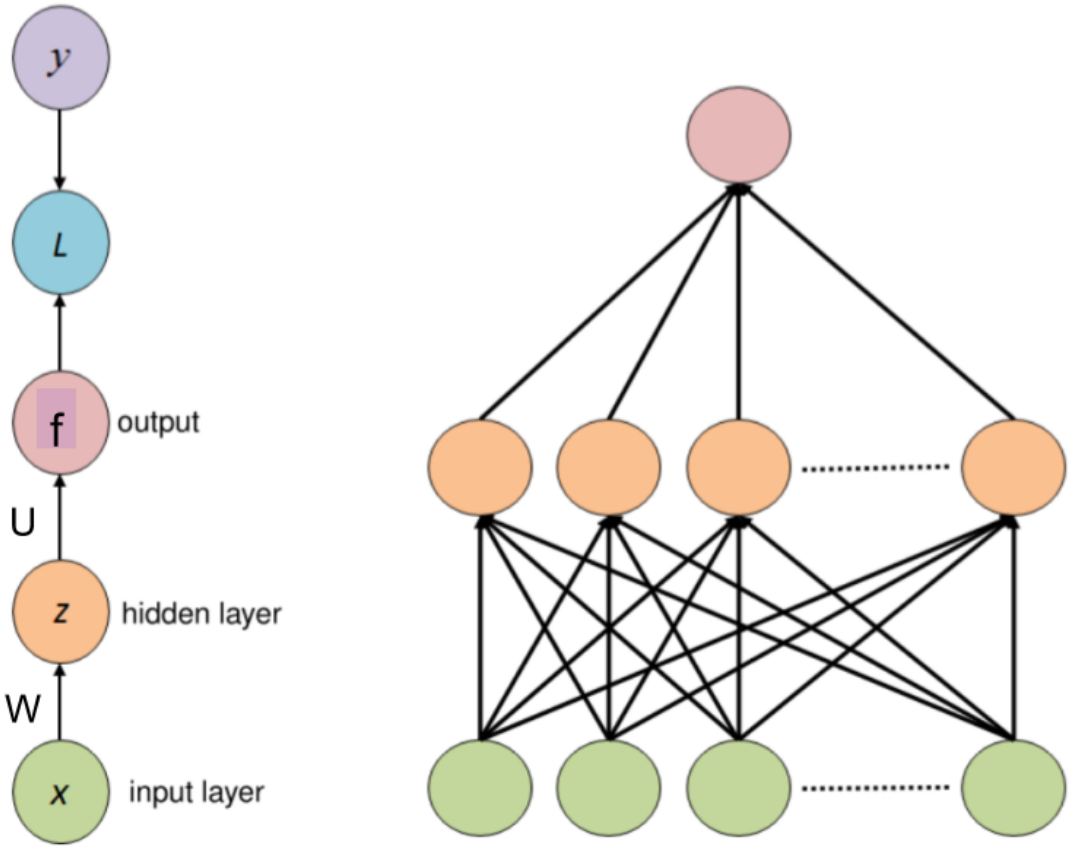
Figure: RNNs can be used in tasks that involve multiple inputs and/or multiple outputs.

Examples:

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

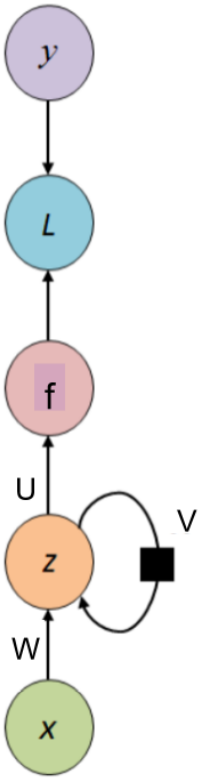
RNNs - Computational Graph

RNNS - COMPUTATIONAL GRAPH



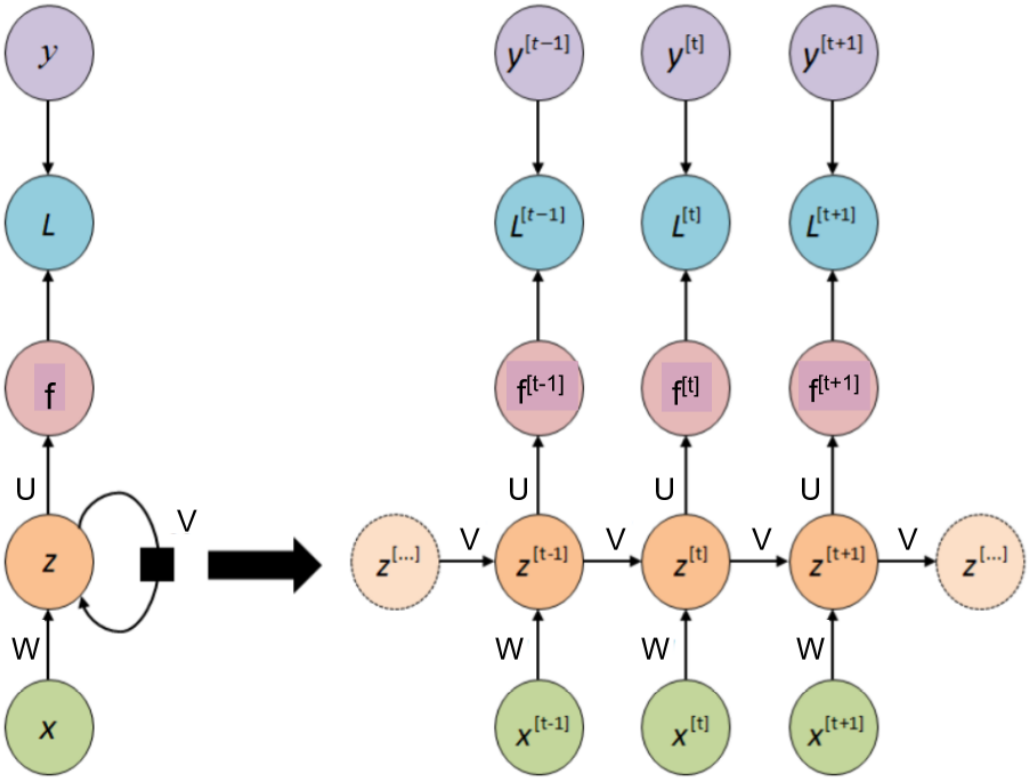
- On the left is the computational graph for the dense net on the right. A loss function L measures how far each output f is from the corresponding training target y.

RNNS - COMPUTATIONAL GRAPH



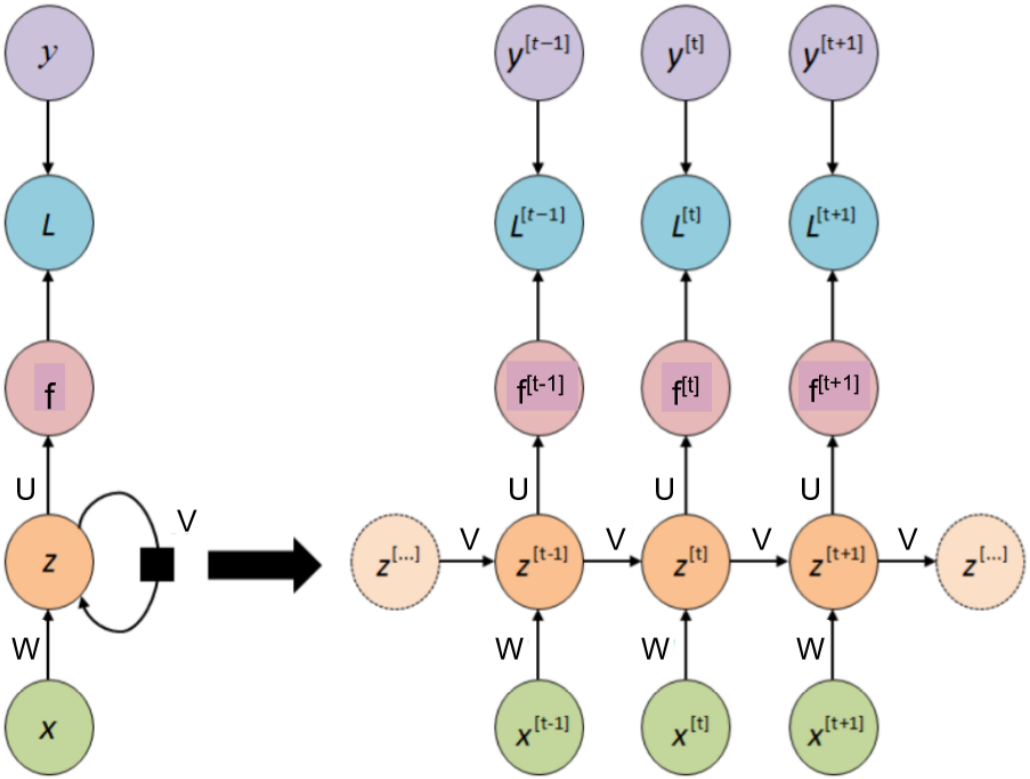
- A helpful way to think of an RNN is as multiple copies of the same network, each passing a message to a successor.
- RNNs are networks with loops, allowing information to persist.

RNNS - COMPUTATIONAL GRAPH



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

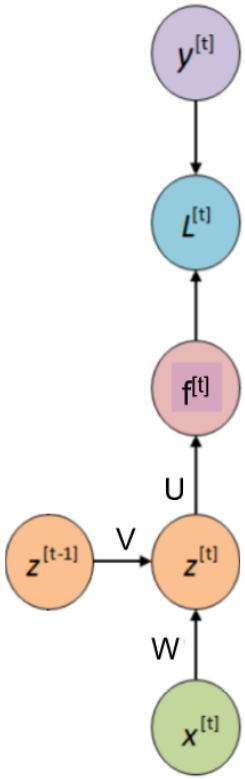
RNNS - COMPUTATIONAL GRAPH



- We went from

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

RNNS - COMPUTATIONAL GRAPH

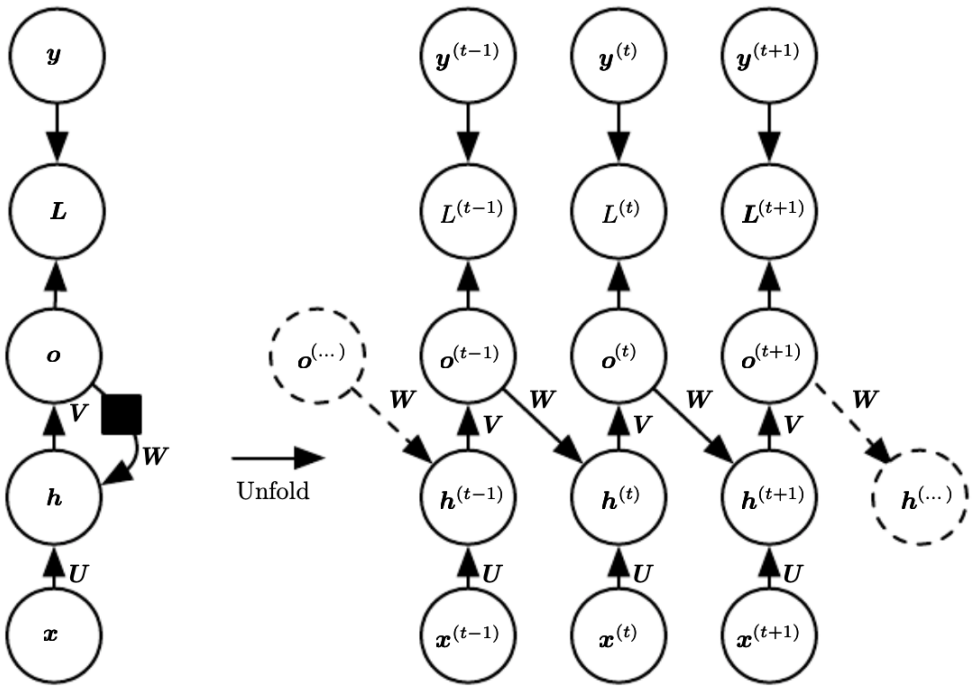


- Potential computational graph for time-step t:

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

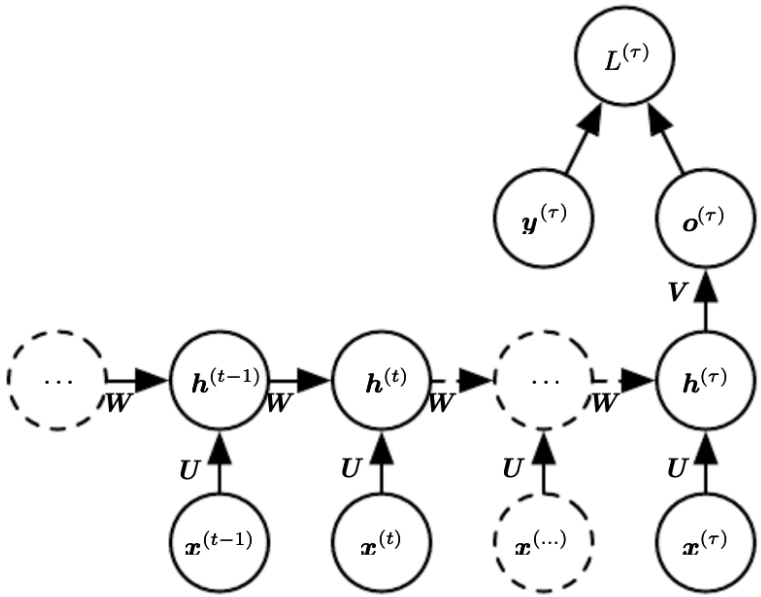
RNNS - COMPUTATIONAL GRAPH WITH RECURRENT OUTPUT-HIDDEN CONNECTIONS

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



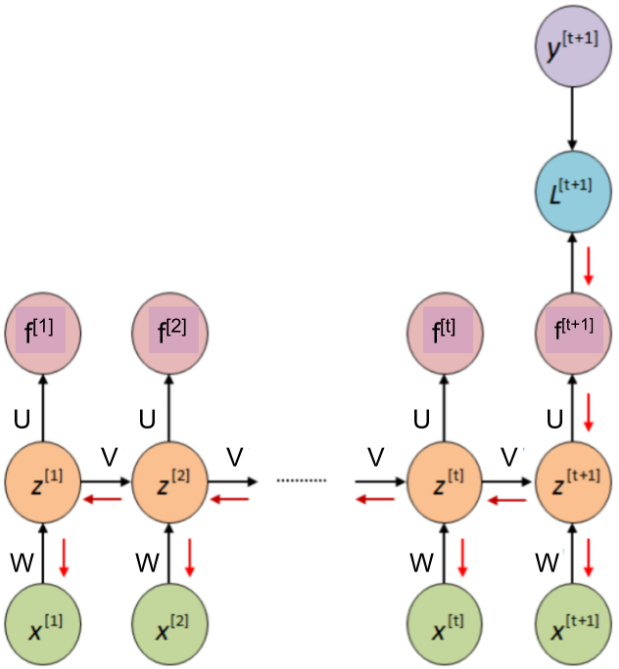
RNNS - COMPUTATIONAL GRAPH FOR SEQ TO ONE MAPPING

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



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

$$\frac{dL}{dz^{[1]}} = \frac{dL}{dz^{[t]}} \frac{dz^{[t]}}{dz^{[t-1]}} \cdots \frac{dz^{[2]}}{dz^{[1]}}$$

LONG-TERM DEPENDENCIES

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

$$\frac{dz^{[t]}}{dz^{[t-1]}} = diag(\sigma'(V^T z^{[t-1]} + W^T x^{[t]} + b))V^T$$

- Therefore, for an arbitrary time-step i in the past, $\frac{dz^{[t]}}{dz^{[i]}}$ will contain the term $(V^T)^{t-i}$ within it. (this follows from the chain rule)
- Based on the largest eigenvalue of V^T, the presence of the term $(V^T)^{t-i}$ can either result in vanishing or exploding gradients.
- This problem is quite severe for RNNs (as compared to feedforward networks) because the **same** matrix V^T is multiplied several times. (▸ Click here)
- As the gap between t and i increases, the instability worsens.

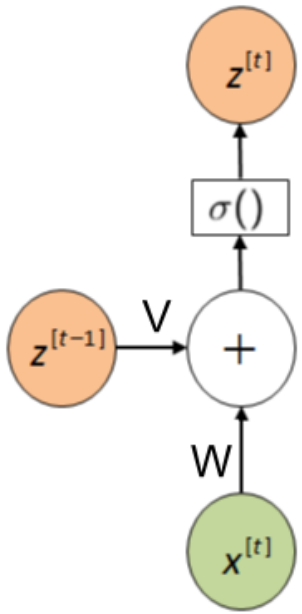
LONG-TERM DEPENDENCIES

- 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.
- Recall, that we can counteract exploding gradients by implementing gradient clipping.
- 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.

Gated units: LSTMs and GRUs

LONG SHORT-TERM MEMORY (LSTM)

The LSTM provides a way of dealing with vanishing gradients and modelling long-term dependencies.

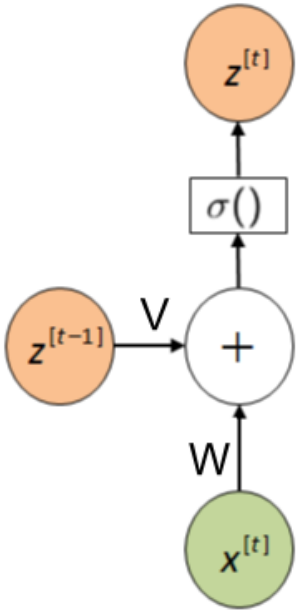


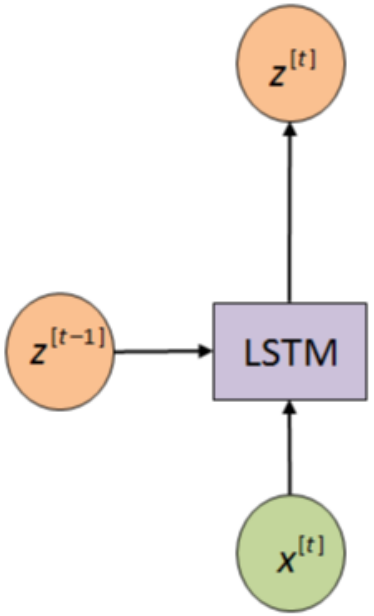
- Until now, we simply computed

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

LONG SHORT-TERM MEMORY (LSTM)

The LSTM provides a way of dealing with vanishing gradients and modelling long-term dependencies.



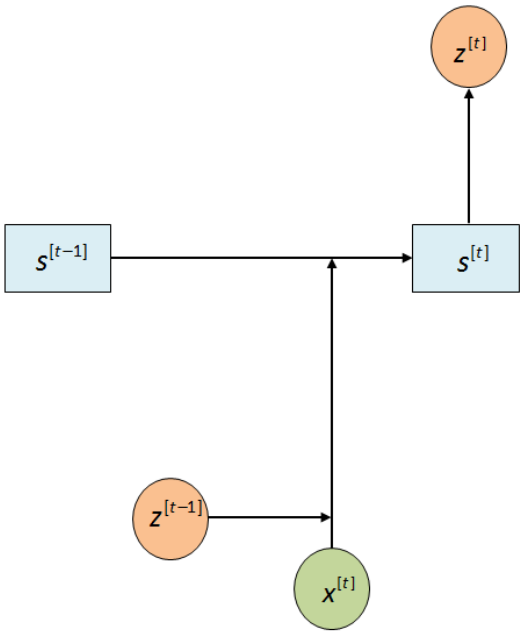


- Until now, we simply computed

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

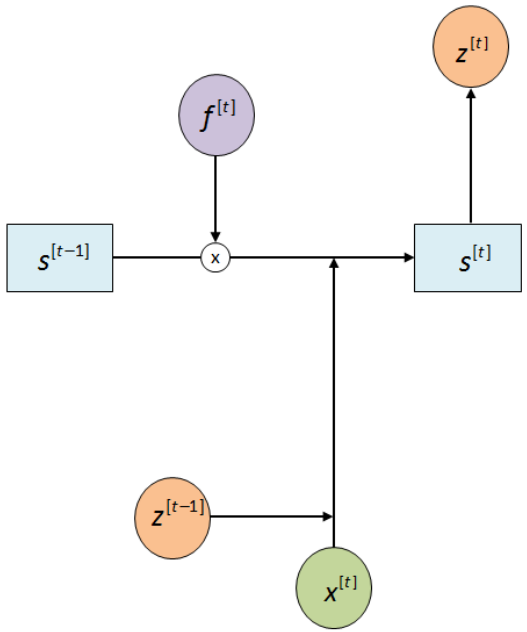
- Now we introduce the LSTM cell (which is a small network on its own)

LONG SHORT-TERM MEMORY (LSTM)



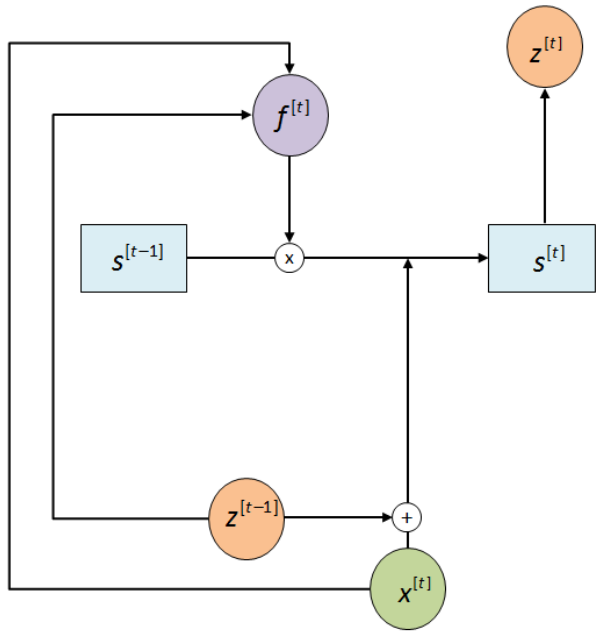
- The key to LSTMs is the **cell state** $s^{[t]}$.
- $s^{[t]}$ can be manipulated by different **gates** to forget old information, add new information, and read information out of it.
- Each gate is a vector of the same size as the $s^{[t]}$ with elements between 0 ("let nothing pass") and 1 ("let everything pass").

LONG SHORT-TERM MEMORY (LSTM)



- **Forget gate** $f^{[t]}$: indicates which information of the old cell state we should forget.
- Intuition: 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 now see a new subject, we want to forget the gender of the old one.

LONG SHORT-TERM MEMORY (LSTM)

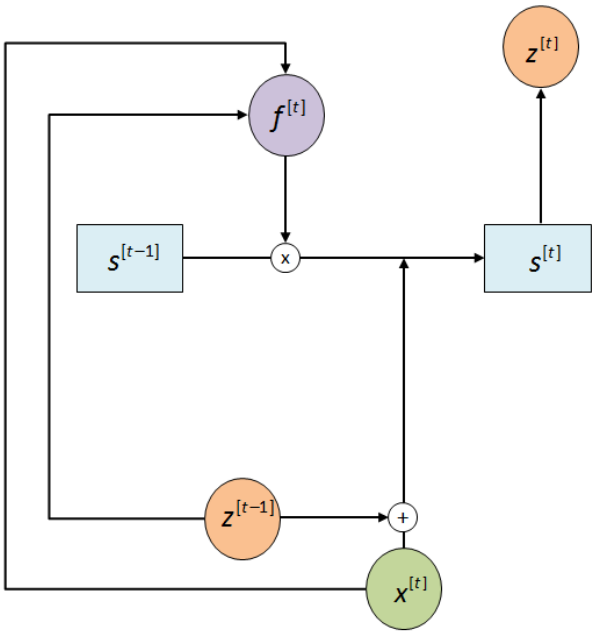


- We obtain the forget gate by computing

$$f^{[t]} = \sigma(b_f + V_f^T z^{[t-1]} + W_f^T x^{[t]})$$

- $\sigma()$ is a sigmoid, squashing the values to $[0, 1]$, and V_f, $W - f$ are foget gate specific weights.

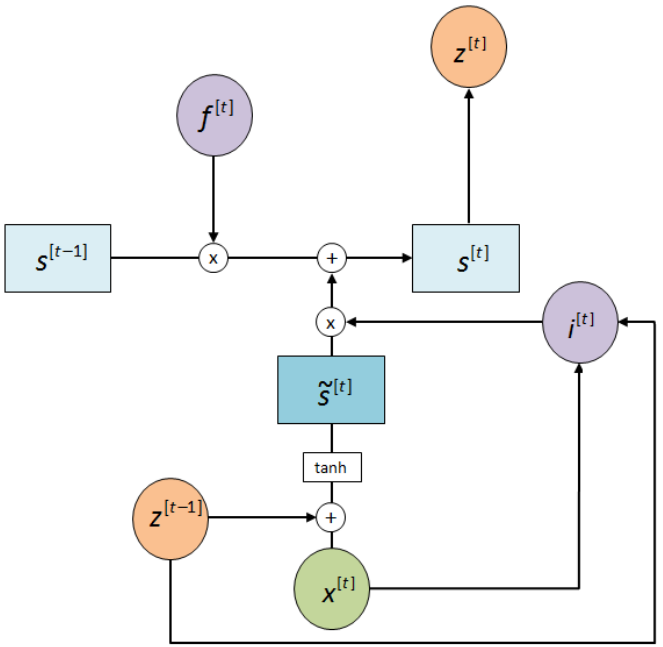
LONG SHORT-TERM MEMORY (LSTM)



- To compute the cell state $s^{[t]}$, the first step is to multiply (element-wise) the previous cell state $s^{[t-1]}$ by the forget gate $f^{[t]}$.

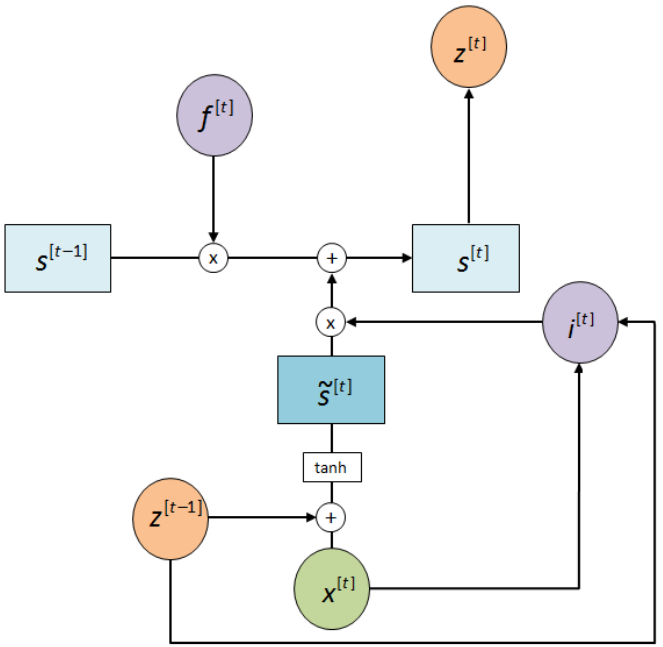
$$f^{[t]} \odot s^{[t-1]}, \text{ with } f^{[t]} \in [0, 1]$$

LONG SHORT-TERM MEMORY (LSTM)



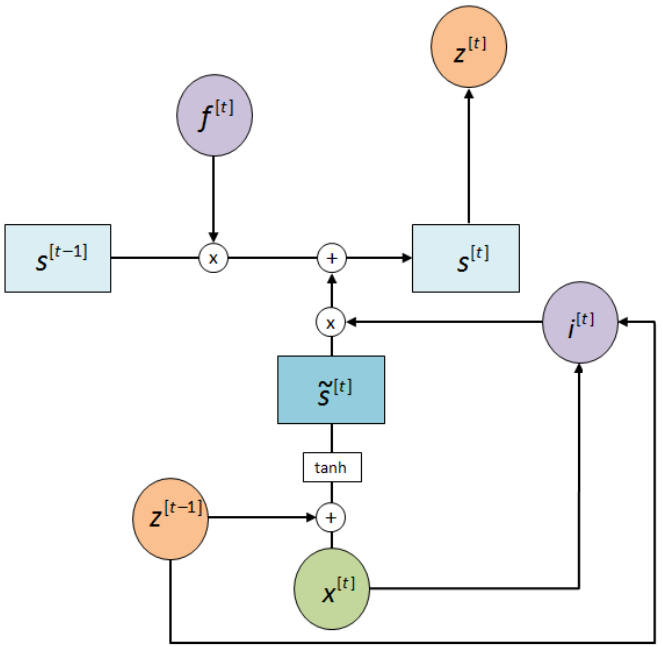
- **Input gate** $i^{[t]}$: indicates which new information should be added to $s^{[t]}$.
- Intuition: In our example, this is where we add the new information about the gender of the new subject.

LONG SHORT-TERM MEMORY (LSTM)



- The new information is given by $\tilde{s}^{[t]} = tanh(b + V^T z^{[t-1]} + W^T x^{[t]}) \in [-1, 1]$.
- The input gate is given by $i^{[t]} = \sigma(b_i + V_i^T z^{[t-1]} + W_i^T x^{[t]}) \in [0, 1]$.
- W and V are weights of the new information, W_i and V_i the weights of the input gate.

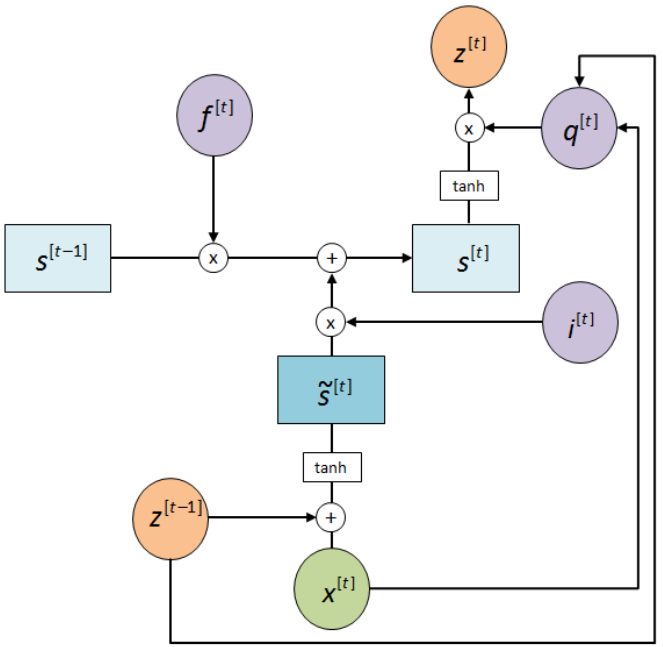
LONG SHORT-TERM MEMORY (LSTM)



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

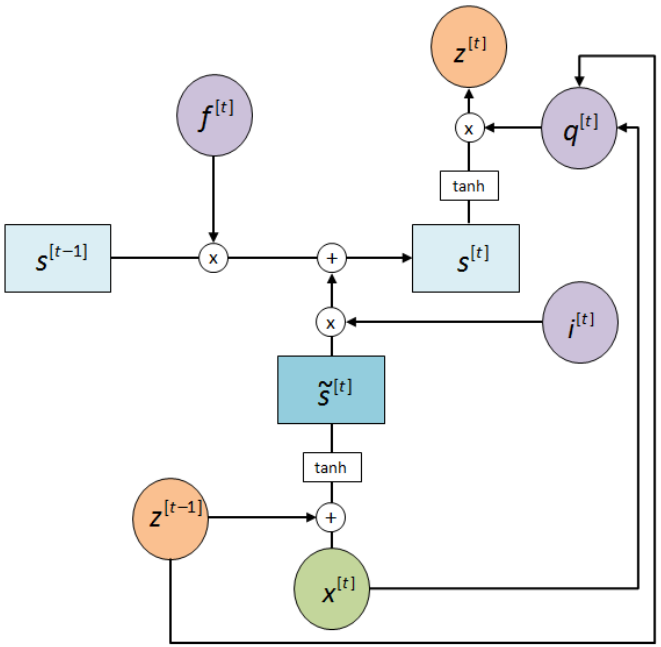
$$s^{[t]} = f^{[t]} \odot s^{[t-1]} + i^{[t]} \odot \tilde{s}^{[t]}$$

LONG SHORT-TERM MEMORY (LSTM)



- **Output gate** $q^{[t]}$: Indicates which information form the cell state is filtered.
- It is given by $q^{[t]} = \sigma(b_q + V_q^T z^{[t-1]} + W_q^T x^{[t]})$, with specific weights W_q, V_q.

LONG SHORT-TERM MEMORY (LSTM)

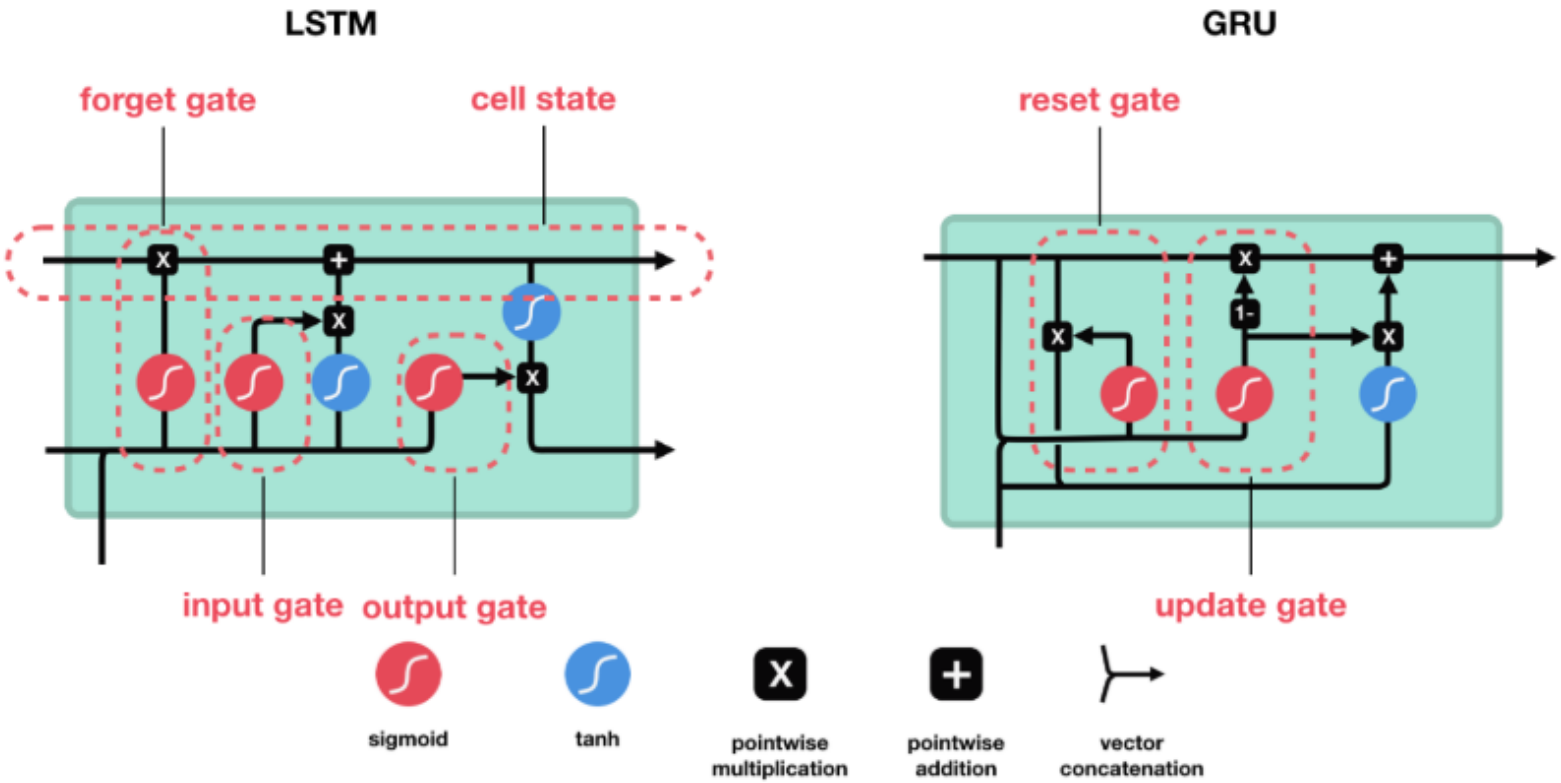


- Finally, the new state $z^{[t]}$ of the LSTM is a function of the cell state, multiplied by the output gate:

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

GATED RECURRENT UNIT (GRU)

A popular alternative to LSTM is the Gated Recurrent Unit (GRU). Its performance can match and sometimes exceed that of an LSTM.

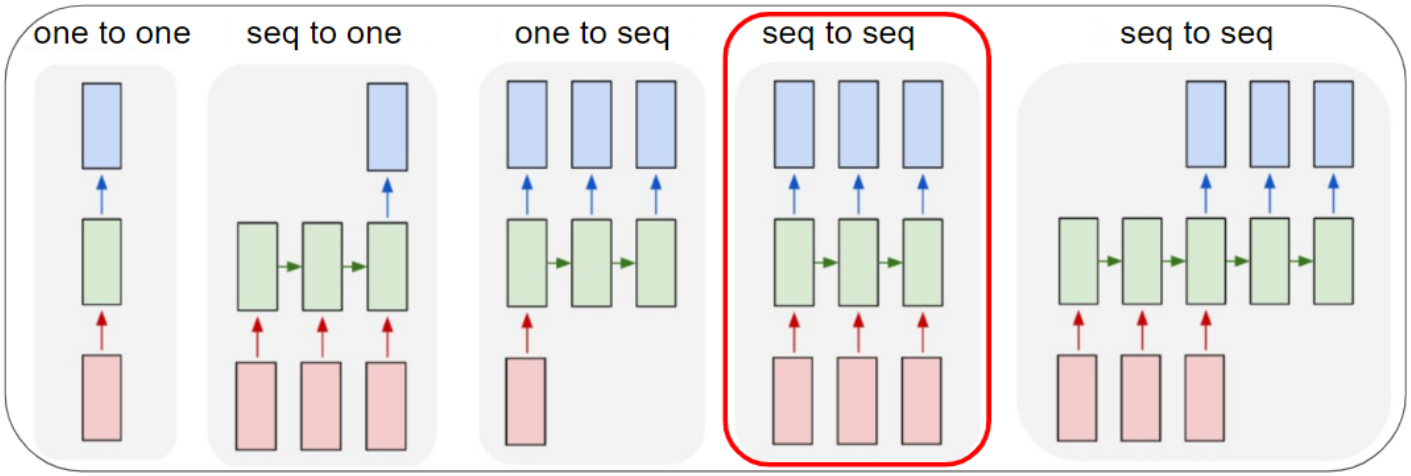


credit: Michael Nguyen

Figure: A GRU has fewer gates and is faster is to train.

Application: Language modelling

Seq-to-Seq (Type I)



RNNS - LANGUAGE MODELLING

- In an earlier example, we built a 'sequence-to-one' RNN model to perform 'sentiment analysis'.
- Another common task in Natural Language Processing (NLP) is **'language modelling'**.
- Input: word/character, encoded as a one-hot vector.
- Output: probability distribution over words/characters given previous words

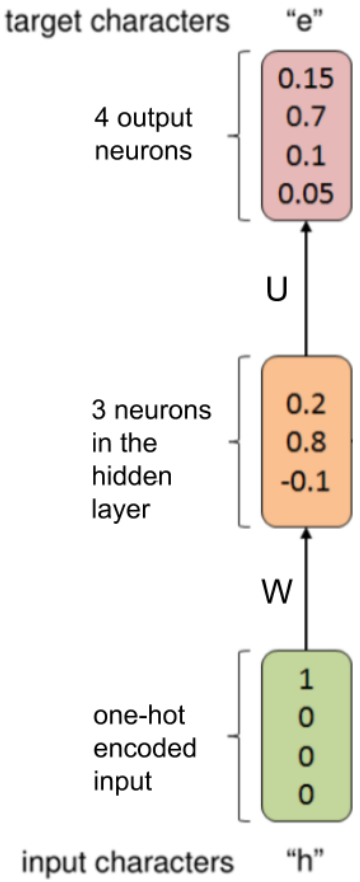
$$P(y^{[1]}, \ldots, y^{[\tau]}) = \prod_{i=1}^{\tau} P(y^{[i]} | y^{[1]}, \ldots, y^{[i-1]})$$

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

RNNS - LANGUAGE MODELLING

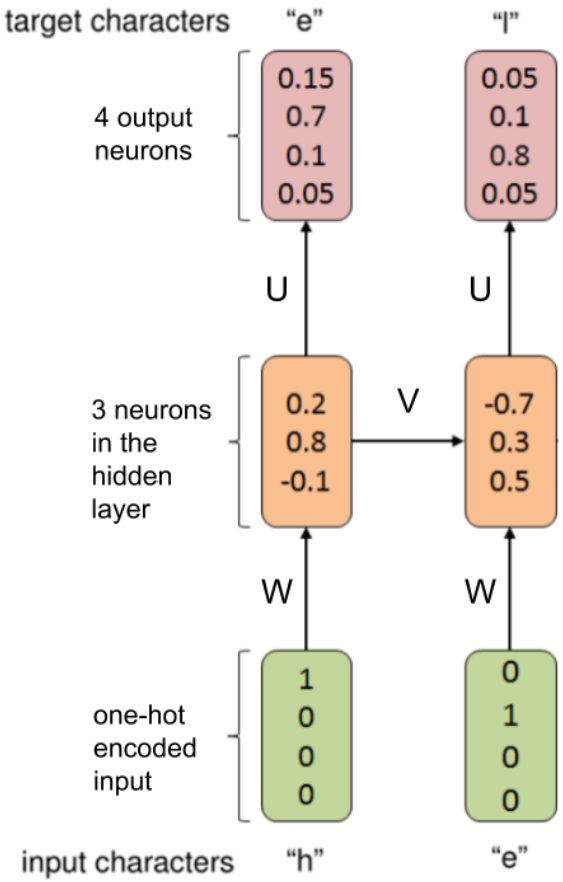
- In this example, we will feed the characters in the word "hello" one at a time to a 'seq-to-seq' RNN.
- For the sake of the visualization, the characters "h", "e", "l" and "o" are one-hot coded as a vectors of length 4 and the output layer only has 4 neurons, one for each character (we ignore the <eos> token).
- At each time step, the RNN has to output a probability distribution (softmax) over the 4 possible characters that might follow the current input.
- Naturally, if the RNN has been trained on words in the English language:
  - 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".

RNNS - LANGUAGE MODELLING



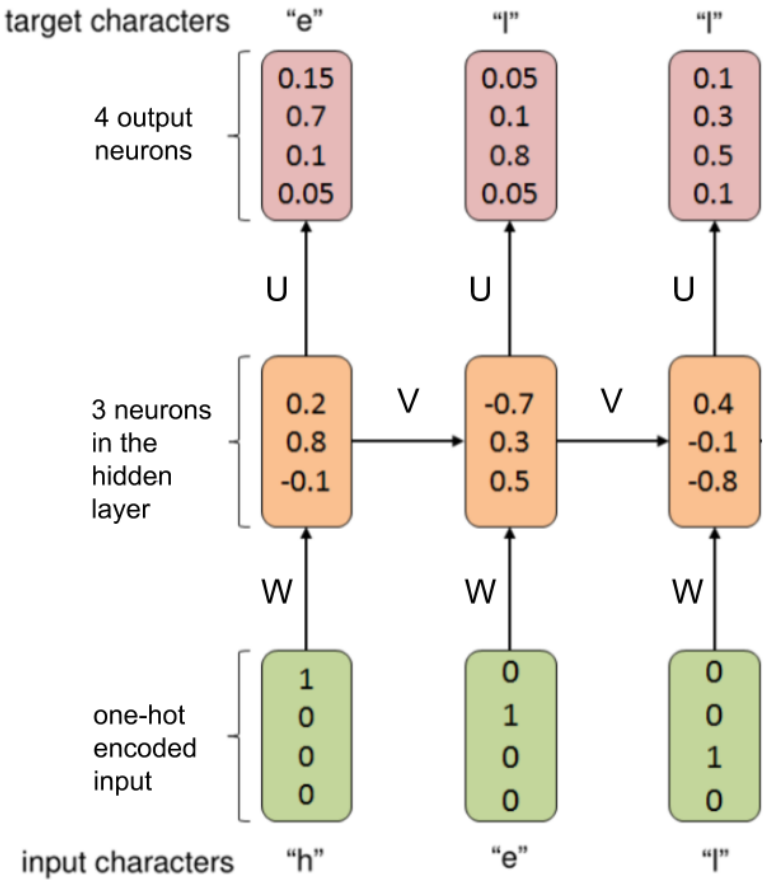
The probability of “e” should be high, given the context of “h”.

RNNS - LANGUAGE MODELLING



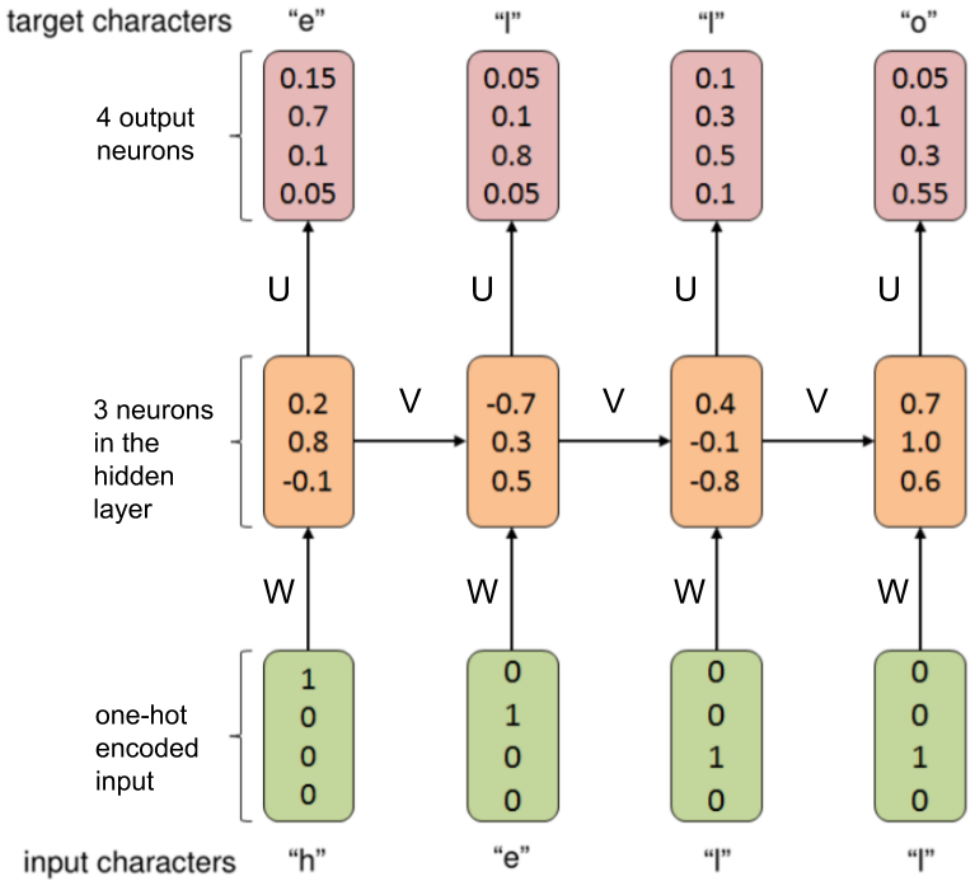
The probability of "l" should be high, given in the context of "he".

RNNS - LANGUAGE MODELLING



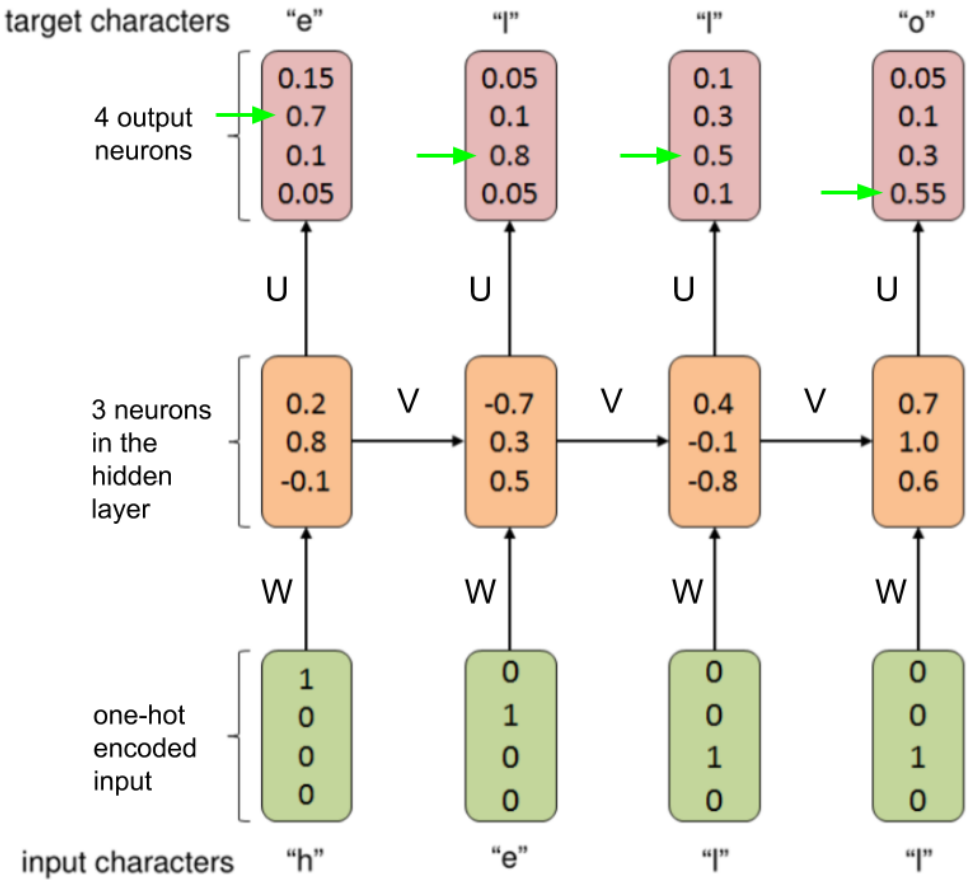
The probability of "l" should **also** be high, given in the context of "hel".

RNNS - LANGUAGE MODELLING



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

RNNS - LANGUAGE MODELLING

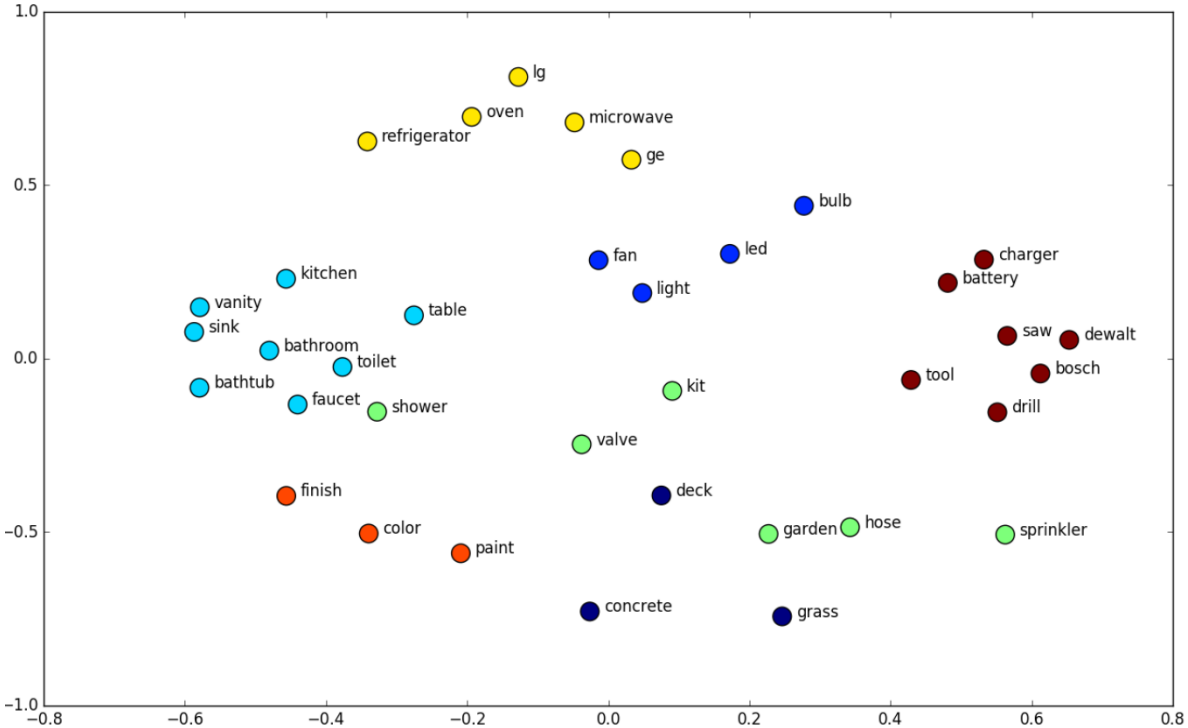


During training, our goal would be to increase the confidence for the correct letters (indicated by the green arrows) and decrease the confidence of all others.

WORD EMBEDDINGS

- In the example in the beginning, words were encoded as sparse one-hot vectors.
- However, this is simply not practical as it would require that the size of the input layer = # of words in the dictionary.
- More importantly, the one-hot representations do not capture the meanings/semantics of the words because all words would be equidistant from one another.
- For example, the word "pineapple" would be equidistant from the words "apple" and "weather".
- Therefore, it is standard practice to encode each word as a dense (as opposed to sparse) vector of fixed size that captures its underlying semantic content.
- The dimensionality of these embeddings is typically much smaller than the number of words in the dictionary.

WORD EMBEDDINGS

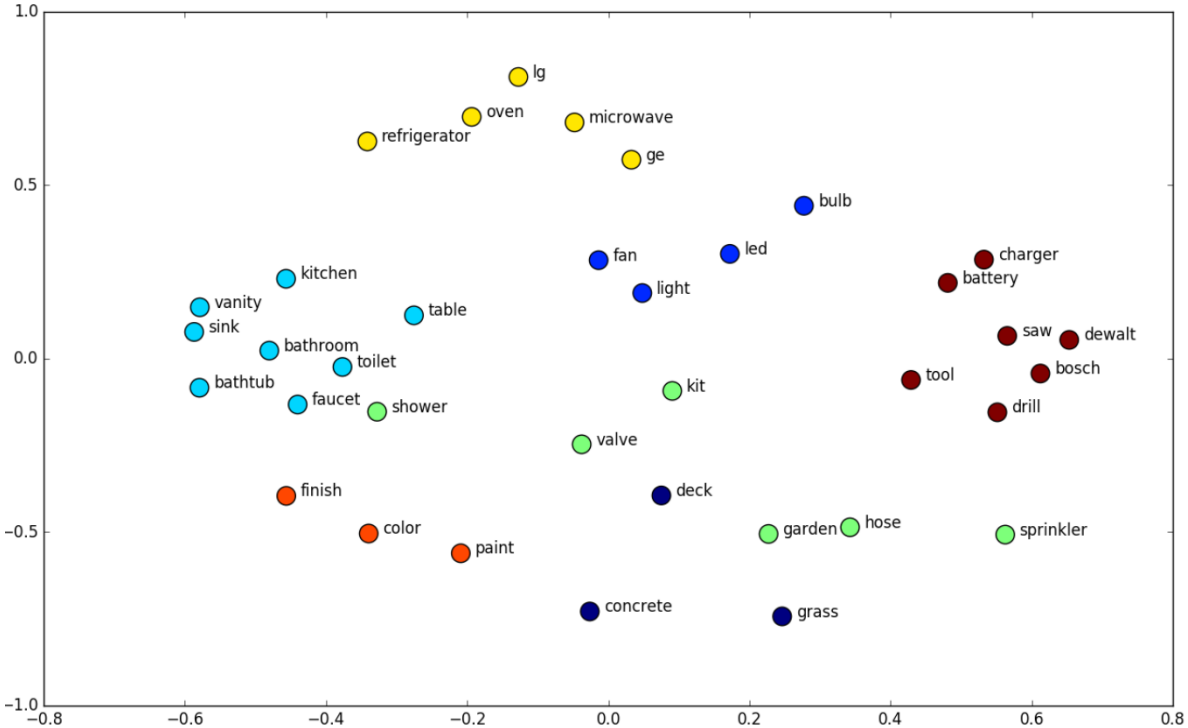


source: Kaggle

Figure: A two-dimensional embedding space is shown here for the sake of visualization. Typically, the embedding space is much higher dimensional.

A major advantage of using word embeddings is that similar words are embedded close to each other in the lower-dimensional embedding space and, therefore, using them gives you a "warm start" for any NLP task.

WORD EMBEDDINGS



source: Kaggle

Figure: A two-dimensional embedding space is shown here for the sake of visualization. Typically, the embedding space is much higher dimensional.

It is an easy way to incorporate prior knowledge into your model and a rudimentary form of **transfer learning**.

WORD EMBEDDINGS

- Two very popular approaches to learn word embeddings are **word2vec** by Google and **GloVe** by Facebook. These embeddings are typically 100 to 1000 dimensional.
- Even though these embeddings capture the meaning of each word to an extent, they do not capture the *semantics* of the word in a given context because each word has a static precomputed representation. For example, depending on the context, the word "bank" might refer to a financial institution or to a river bank.
- Recently, there have been significant breakthroughs in context-based embeddings. One such example are the embeddings provided by BERT [Devlin et al., 2018], a **transformer model** which was trained on a corpus of 3.3 billion words.
- BERT (a non-recurrent model!) obtained new state-of-the-art performance on 11 NLP tasks.

Bidirectional RNNs

BIDIRECTIONAL RNNS

- Another generalization of the simple RNN are bidirectional RNNs.
- These allow us 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.

BIDIRECTIONAL RNNS

Computational graph of an bidirectional RNN:

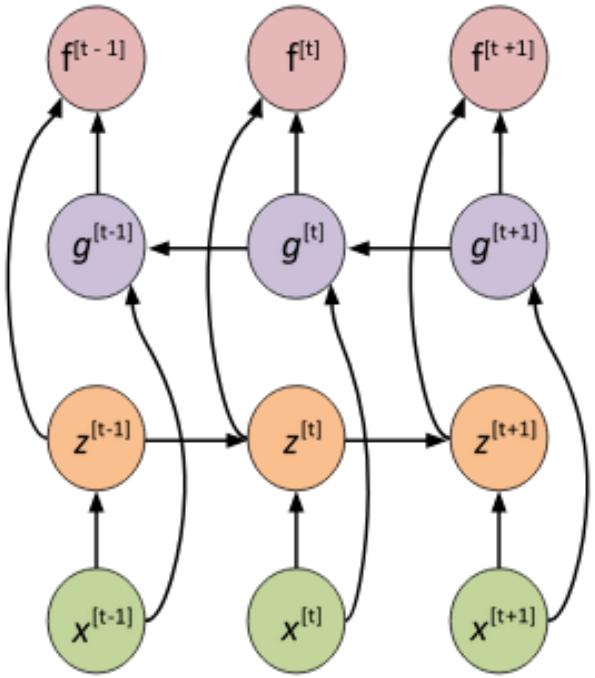
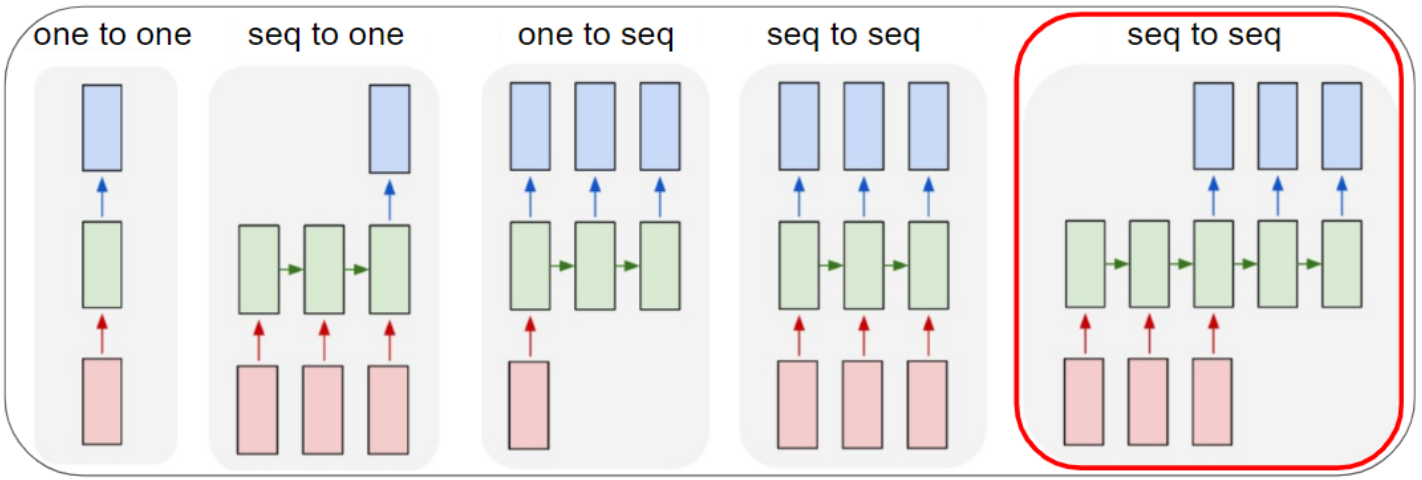


Figure: A bidirectional RNN consists of a forward RNN processing inputs from left to right and a backward RNN processing inputs backwards in time.

Encoder-Decoder Architectures

Seq-to-Seq (Type II)



ENCODER-DECODER NETWORK

- For many interesting applications such as question answering, dialogue systems, 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!

ENCODER-DECODER NETWORK

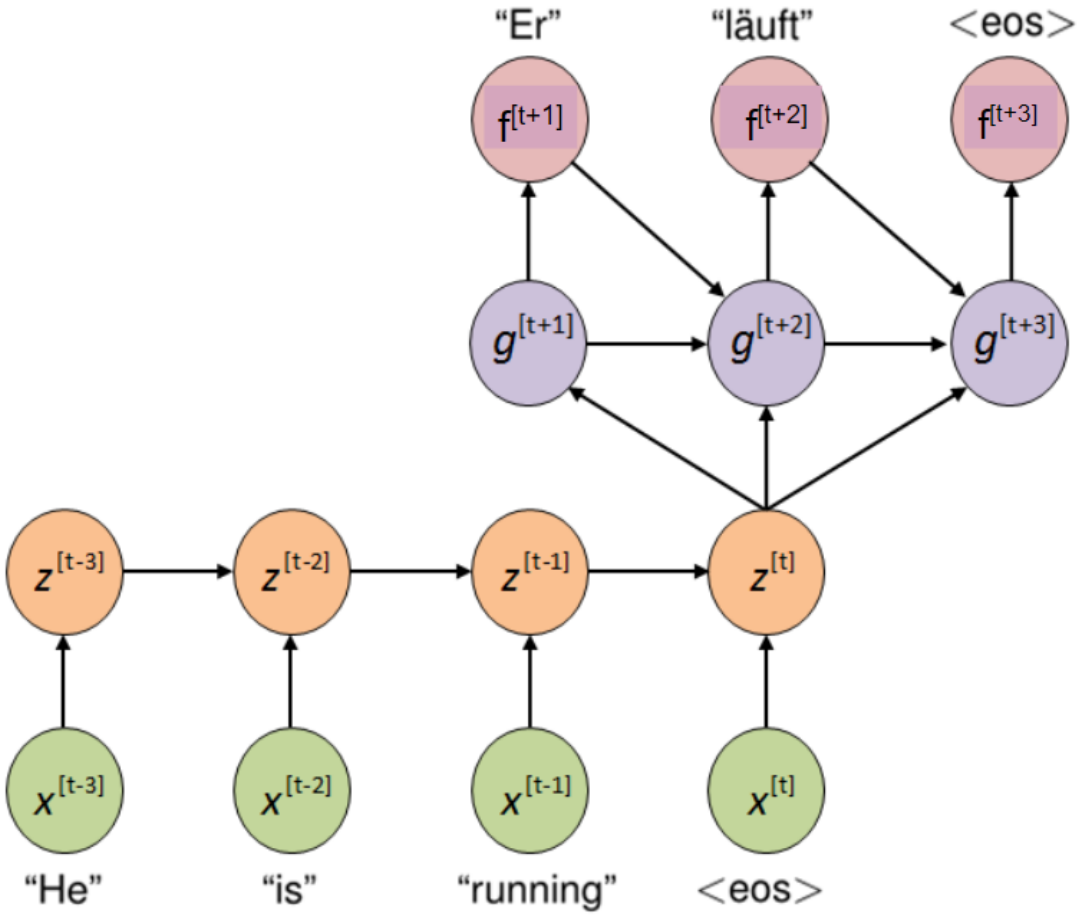


Figure: In the first part of the network, information from the input is encoded in the context vector, here the final hidden state, which is then passed on to every hidden state of the decoder, which produces the target sequence.

ENCODER-DECODER NETWORK

- An input/encoder-RNN processes the input sequence of length n_x and computes a fixed-length context vector C, usually the final hidden state or simple function of the hidden states.
- 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.
- The context vector summarizes important information from the input sequence, e.g. the intent of a question in a 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.

ENCODER-DECODER NETWORK

- In machine translation, 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.
- 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.

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.
- Each hidden state contains mostly information from recent inputs.
- Key idea: Allow the decoder to access all the hidden states of the encoder (instead of just the final one) so that it can dynamically decide which ones are relevant at each time-step of the decoding process.
- This means the decoder can choose to "focus" on different hidden states (of the encoder) at different time-steps of the decoding process similar to how the human eye can focus on different regions of the visual field.
- This is known as an **attention mechanism**.

ATTENTION

- The attention mechanism is implemented by an additional component in the decoder.
- For example, this can be a simple single-hidden layer feed-forward neural network which is trained along with the RNN.
- At any given time-step i of the decoding process, the network computes the relevance of encoder state $z^{[j]}$ as:

$$rel(z^{[j]})^{[i]} = v_a^T \, tanh(W_a[g^{[i-1]}; z^{[j]}])$$

  where v_a and W_a are the parameters of the feed-forward network, $g^{[i-1]}$ is the decoder state from the previous time-step and ';' indicates concatenation.
- The relevance scores (for all the encoder hidden states) are then normalized which gives the *attention weights* $(\alpha^{[j]})^{[i]}$:

$$(\alpha^{[j]})^{[i]} = \frac{\exp(rel(z^{[j]})^{[i]})}{\sum_{j'} \exp(rel(z^{[j']})^{[i]})}$$

ATTENTION

- The attention mechanism 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.

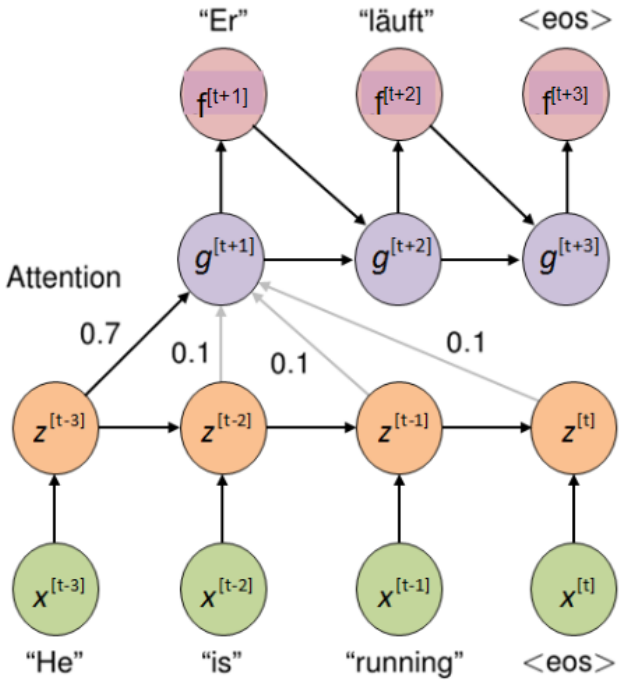


Figure: Attention at $i = t + 1$

ATTENTION

- At each time step i, a set of weights $(\alpha^{[j]})^{[i]}$ is computed which determine how to combine the hidden states of the encoder into a context vector $c^{[i]} = \sum_{j=1}^{n_x} (\alpha^{[j]})^{[i]} z^{[j]}$, which holds the necessary information to predict the correct output.

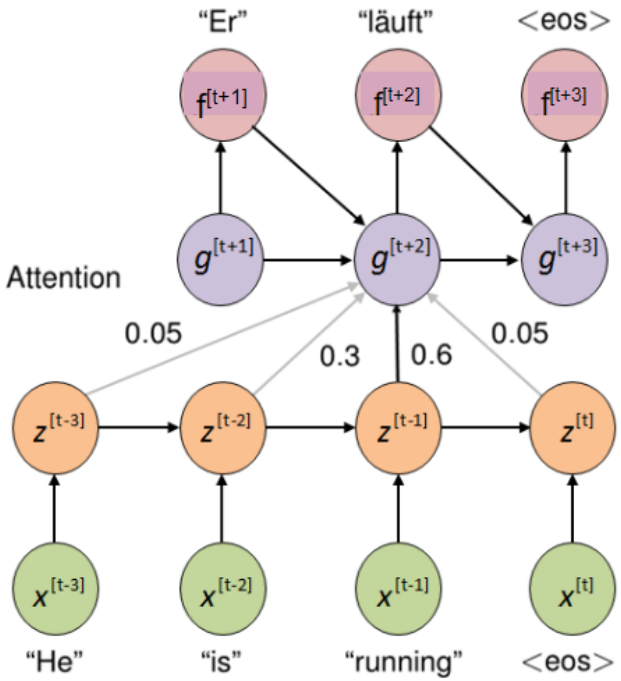
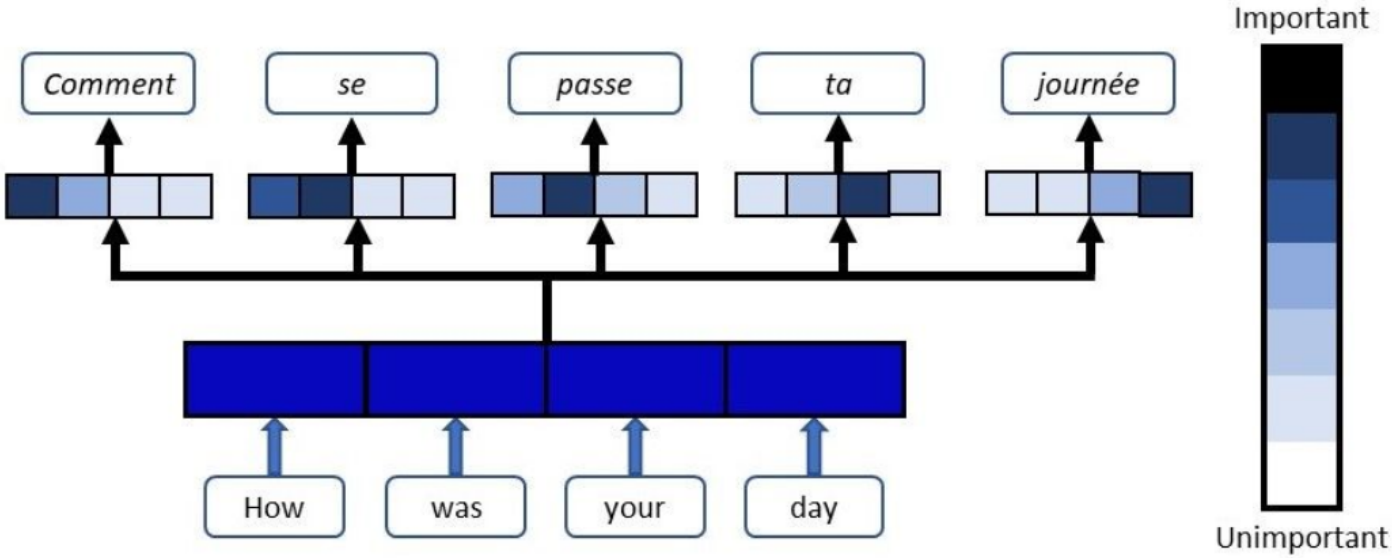


Figure: Attention at $i = t + 2$

ATTENTION



Credit: FloydHub

Figure: An illustration of a machine translation task using an encoder-decoder model with an attention mechanism. The attention weights at each time-step of the decoding/translation process indicate which parts of the input sequence are most relevant. (There are 4 attention weights because there are 4 encoder states.)

More application examples

SOME MORE SOPHISTICATED APPLICATIONS

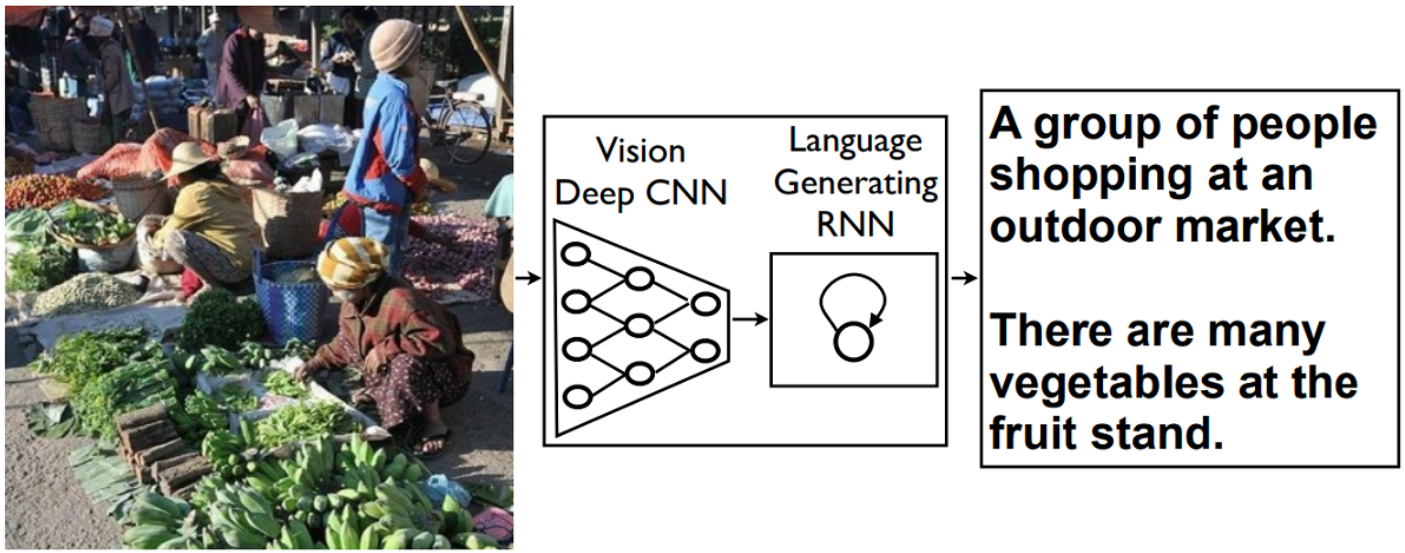


Figure: Show and Tell: A Neural Image Caption Generator (Oriol Vinyals et al. 2014). A language generating RNN tries to describe in brief the content of different images.

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

A woman is throwing a frisbee in a park.

A stop sign is on a road with a mountain in the background.

Figure: Attention for image captioning: the attention mechanism tells the network roughly which pixels to pay attention to when writing the text (Kelvin Xu al. 2015)

SOME MORE SOPHISTICATED APPLICATIONS

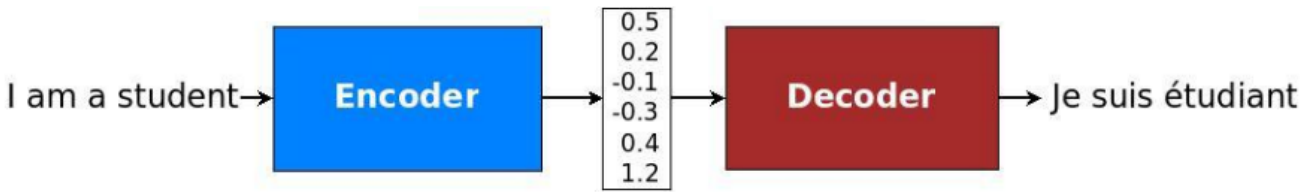


Figure: Neural Machine Translation (seq2seq): Sequence to Sequence Learning with Neural Networks (Ilya Sutskever et al. 2014). As we saw earlier, an encoder converts a source sentence into a “meaning” vector which is passed through a decoder to produce a translation.

SOME MORE SOPHISTICATED APPLICATIONS

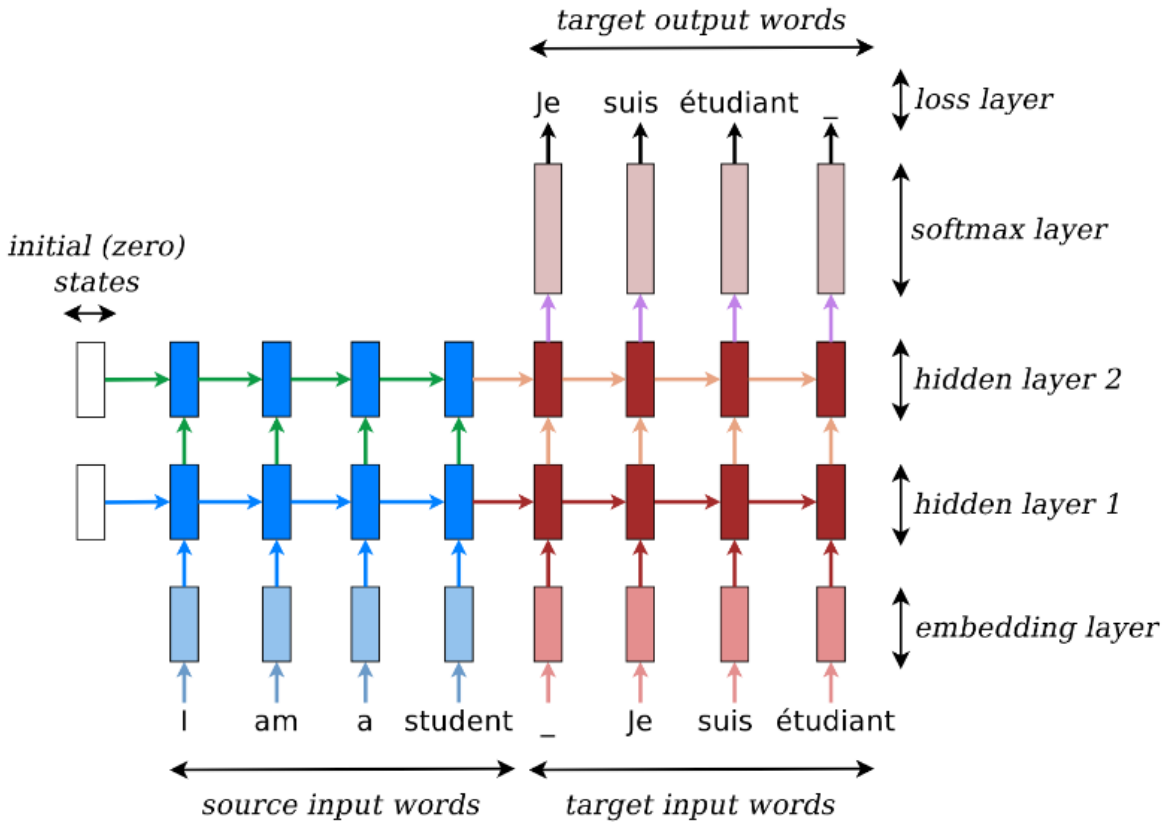


Figure: Neural Machine Translation (seq2seq): Sequence to Sequence Learning with Neural Networks (Ilya Sutskever et al. 2014). As we saw earlier, an encoder converts a source sentence into a “meaning” vector which is passed through a decoder to produce a translation.

SOME MORE SOPHISTICATED APPLICATIONS

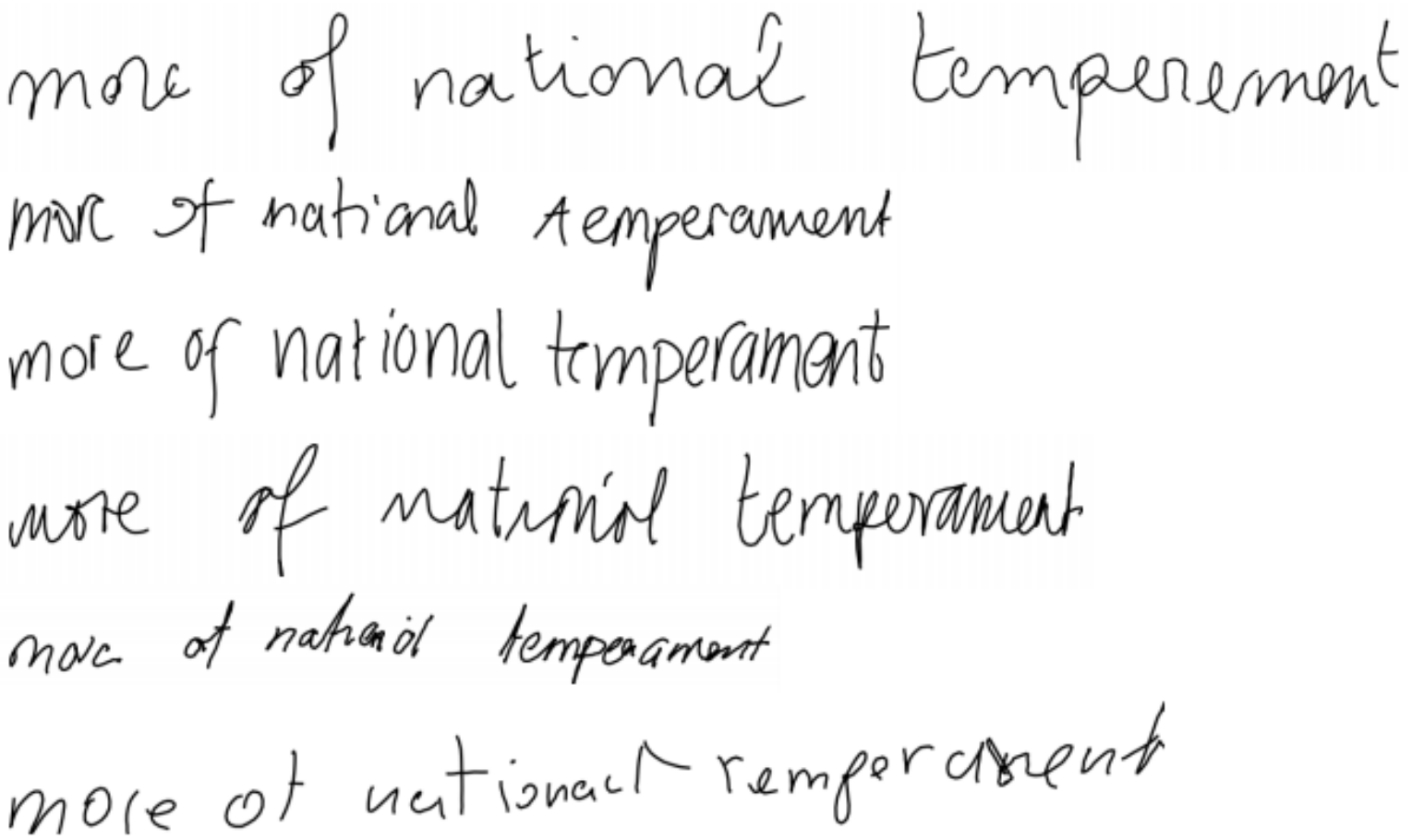


Figure: Generating Sequences With Recurrent Neural Networks (Alex Graves, 2013). Top row are real data, the rest are generated by various RNNs.

SOME MORE SOPHISTICATED APPLICATIONS

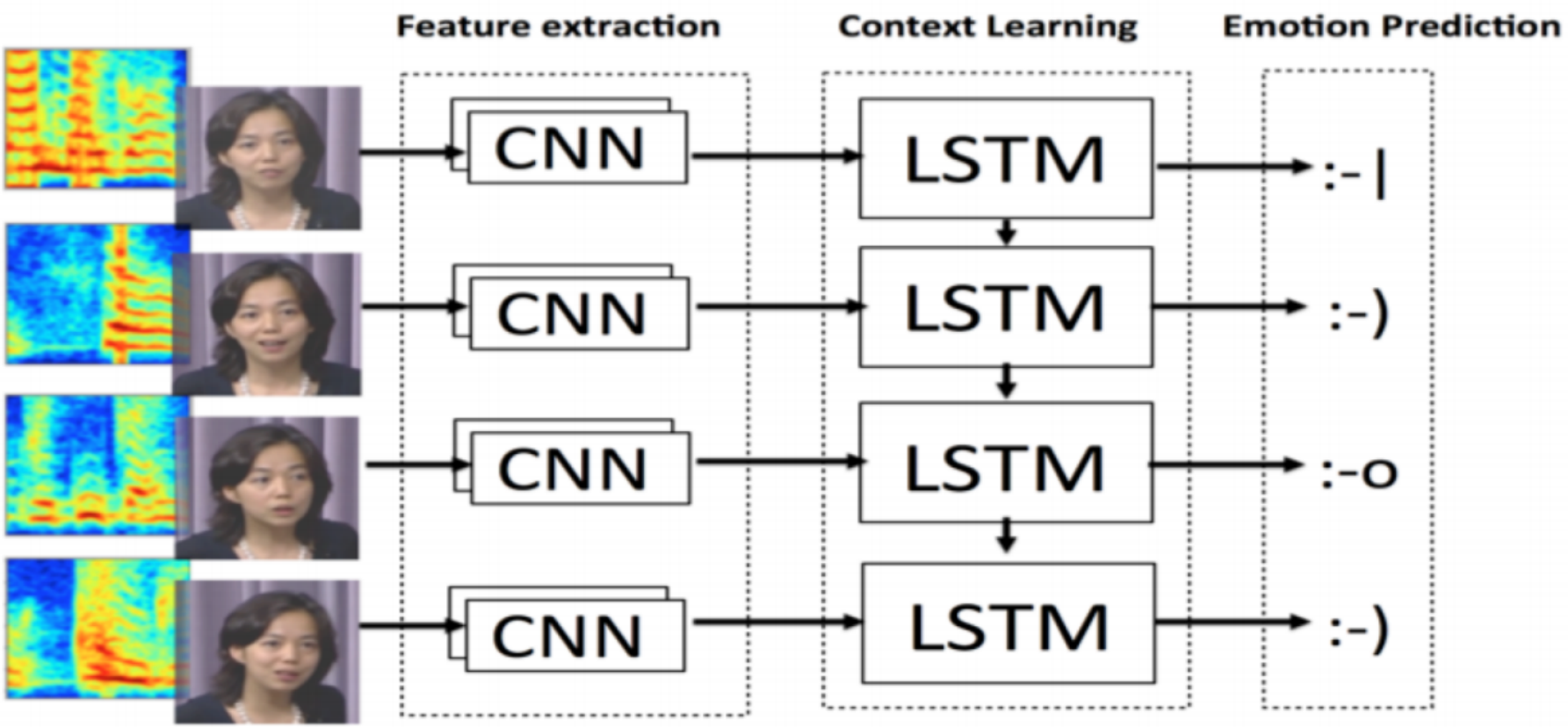


Figure: Convolutional and recurrent nets for detecting emotion from audio data (Namrata Anand & Prateek Verma, 2016). We already had this example in the CNN chapter!

SOME MORE SOPHISTICATED APPLICATIONS

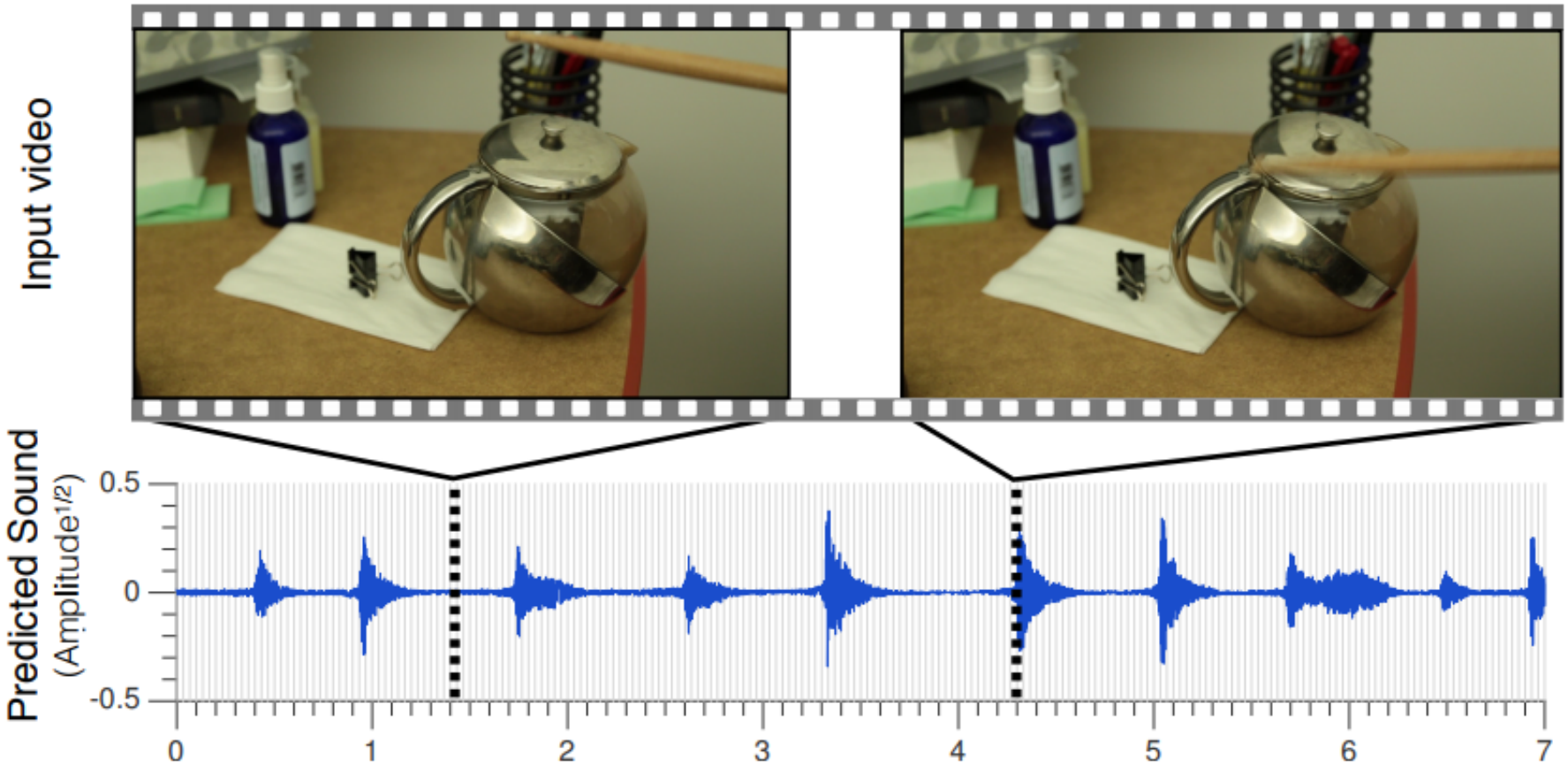


Figure: Visually Indicated Sounds (Andrew Owens et al. 2016). A model to synthesize plausible impact sounds from silent videos.(Click here.)

CNNs or RNNs?

CNNS OR RNNS?

- Historically, RNNs were the default models used in sequence processing tasks.
- However, some families of CNNs (especially those based on Fully Convolutional Networks (FCNs)) *can* be used to process variable-length sequences such as text or time-series data.
- If a CNN doesn't contain any fully-connected layers, the total number of weights in the network is independent of the spatial dimensions of the input because of weight-sharing in the convolutional layers.
- Recent research [Bai et al. , 2018] indicates that such convolutional architectures (which the authors term Temporal Convolutional Networks (TCNs)) can outperform RNNs on a wide range of tasks.
- A major advantage of TCNs is that the entire input sequence can be fed to the network at once (as opposed to sequentially.)

CNNS OR RNNS?

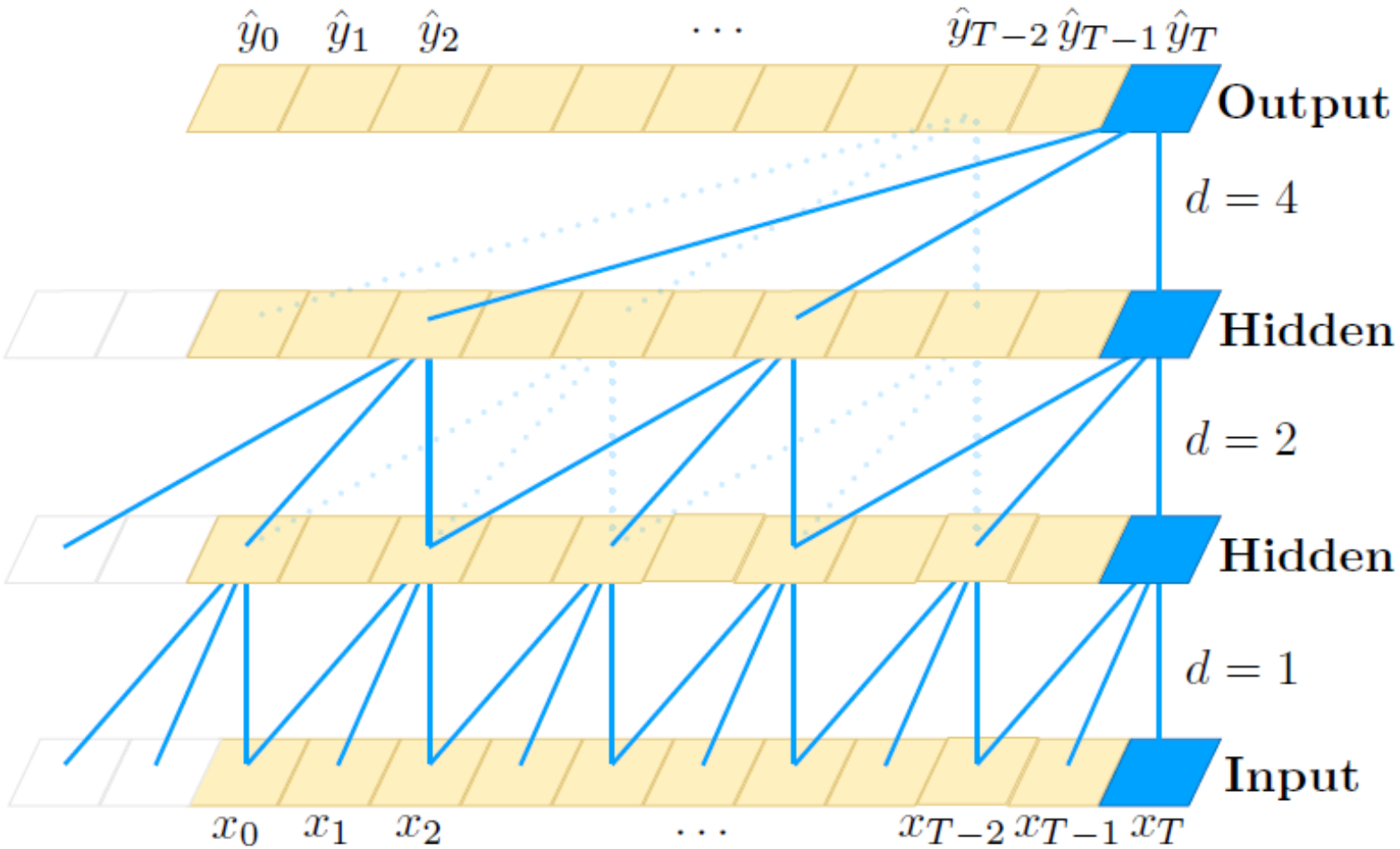


Figure: A TCN (we have already seen this in the CNN lecture!) is simply a variant of the one-dimensional FCN which uses a special type of dilated convolutions called **causal dilated** convolutions.

CNNS OR RNNS?

Sequence Modeling Task	Model Size ($\approx$)	Models			
		LSTM	GRU	RNN	**TCN**
Seq. MNIST (accuracy^h)	70K	87.2	96.2	21.5	**99.0**
Permuted MNIST (accuracy)	70K	85.7	87.3	25.3	**97.2**
Adding problem T=600 (loss$^\ell$)	70K	0.164	**5.3e-5**	0.177	**5.8e-5**
Copy memory T=1000 (loss)	16K	0.0204	0.0197	0.0202	**3.5e-5**
Music JSB Chorales (loss)	300K	8.45	8.43	8.91	**8.10**
Music Nottingham (loss)	1M	3.29	3.46	4.05	**3.07**
Word-level PTB (perplexity$^\ell$)	13M	**78.93**	92.48	114.50	88.68
Word-level Wiki-103 (perplexity)	-	48.4	-	-	**45.19**
Word-level LAMBADA (perplexity)	-	4186	-	14725	**1279**
Char-level PTB (bpc$^\ell$)	3M	1.36	1.37	1.48	**1.31**
Char-level text8 (bpc)	5M	1.50	1.53	1.69	**1.45**

Figure: Evaluation of TCNs and recurrent architectures on a wide range of sequence modelling tasks. ^{h} means higher is better and $^\ell$ means lower is better. Note: To make the comparisons fair, all models have roughly the same size (for a given task) and the authors used grid search to find good hyperparameters for the recurrent architectures.

SUMMARY

- RNNs are specifically designed to process sequences of varying lengths.
- For that recurrent connections are introduced into the network structure.
- The gradient is calculated by backpropagation through time.
- An LSTM replaces the simple hidden neuron by a complex system consisting of cell state, and forget, input, and output gates.
- An RNN can be used as a language model, which can be improved by word-embeddings.
- Different advanced types of RNNs exist, like Encoder-Decoder architectures and bidirectional RNNs.[1]

1. A bidirectional RNN processes the input sequence in both directions (front-to-back and back-to-front).

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

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

REFERENCES

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel and Yoshua Bengio (2015)

Show, Attend and Tell: Neural Image Caption Generation with Visual Attention

https://arxiv.org/abs/1502.03044

Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova (2018)

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

https://arxiv.org/abs/1810.04805

Shaojie Bai, J. Zico Kolter, Vladlen Koltun (2018)

An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling

https://arxiv.org/abs/1803.01271

Lilian Weng (2018)

Attention? Attention!

https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html