

Lecture 10

# Analysis of Sequential Data

RNNs, LSTMs, GRUs

## Machine Translation

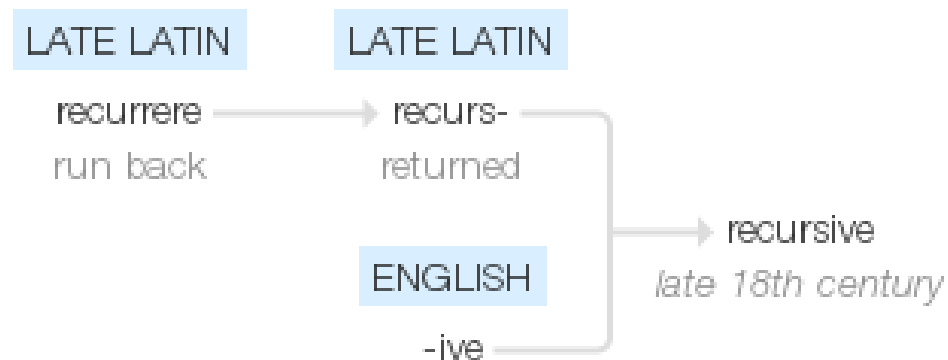
CSCI E-89 Deep Learning, Fall 2024

Zoran B. Djordjević

# Sources

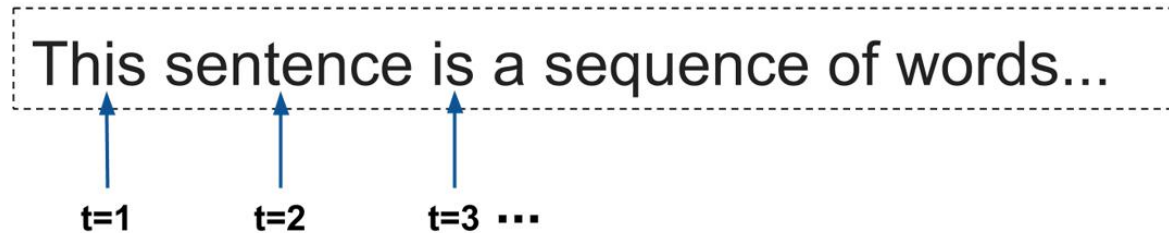
This lecture follows:

- Chapter 6, "Deep Learning with Python", by Francois Chollet, 1<sup>st</sup> Edition
- Chapter 10, "Deep Learning with Python", by Francois Chollet, 2<sup>nd</sup> Edition
- Chapter 15, "*Hands-on Machine Learning with Scikit-Learn & TensorFlow*" by Aurélien Géron, O'Reilly 2019, 2<sup>nd</sup> Edition
- "*The Unreasonable Effectiveness of Recurrent Neural Networks*", May 21, 2015, Andrej Karpaty, <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>
- "*Recurrent Neural Networks*", by Andrew Ng, Coursera.com online course.
- Jürgen Schmidhuber's Home Page: <http://people.idsia.ch/~juergen/>
  - re·cur·rent rə'kərənt/  
*adjective*: occurring often or repeatedly.
  - re·cur·sive rə'kərsiv/  
*adjective*: characterized by recurrence or repetition, in particular.

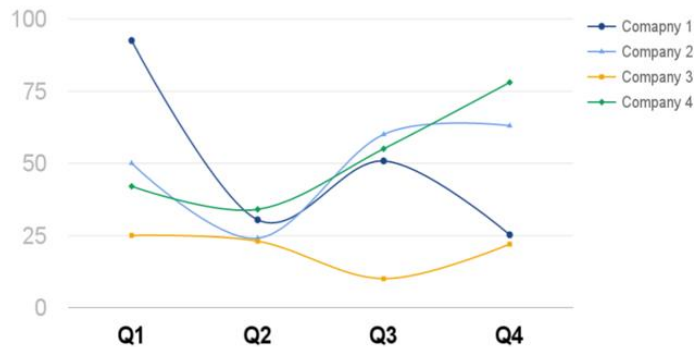


# Sequences

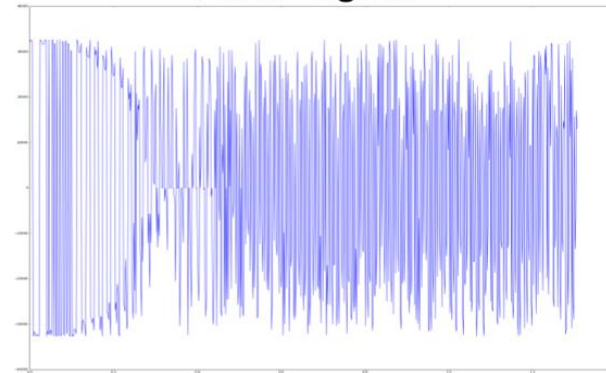
- Many very important and useful types of data have sequential structure.
- Sequential structures appears in many datasets, across all domains.
- In computer vision, video is a sequence of visual content evolving over time.
- In speech, we have audio signals; in genomics, gene sequences; time spaced medical records in healthcare; financial data in the stock market, and so on.



**Company performance over time**



**Audio signal**



# Transformations of Sequence Data

Speech to text



"The quick brown fox jumped over the lazy dog."

Music generation

∅



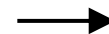
Sentiment classification

"There is nothing to like in this movie."



DNA sequence analysis

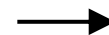
AGCCCCTGTGAGGAACTAG



AG**CCCCTGTGAGGAACT**AG

Machine translation

Voulez-vous chanter avec moi?



Do you want to sing with me?

Video activity recognition



Running

Name entity recognition

Yesterday, Harry Potter met Hermione Granger.



Yesterday, **Harry Potter** met **Hermione Granger**.

# Text

- A particularly important type of data with strong sequential structure is natural language—text data and spoken speech.
- Deep learning methods that exploit the sequential structure inherent in texts—characters, words, sentences, paragraphs, documents—are at the forefront of natural language processing (NLP) systems.
- There are many types of NLP tasks such as: document classification, building language models, answering questions automatically and generating human-level conversation agents, for example.
- NLP tasks are difficult and are of continuous interest of the entire AI community both in academia and industry.
- In what follows, we will show how to work with sequences—primarily text.
- We will begin with a historically important and once very popular class of deep learning models for sequences: Recurrent Neural Networks (RNNs).
- In these notes, we use RNN as a name for a specific class of neural networks as well as the general term which includes the original RNNs, LSTMs, GRUs and other recurrent networks.

# Importance of the (past) History

- In a sequence of events (e.g., sentence), to understand the meaning of the current event (word) we often need information on the past events (words).
- For example, to understand the last word in these two sentences:

*The river spread over its **bank** . . . .*

*The project was financed by the **bank** . . . .*

- we rely on the previous words to understand the meaning of the word “**bank**”.
- In Mathematics, this type of dependence of the current value (event, word) on the previous event(s) is called recurrence and is expressed using recurrence equations (or recursive equations) of the form:

$$x_k = f(x_{k-1}, x_{k-2}, x_{k-3}, \dots)$$

- This type of relationship can be generalized as a function describing dependence of discrete variable  $y_k$  on its previous value  $y_{k-1}$  and some current input  $x_k$  .

$$y_k = f(y_{k-1}, x_k)$$

# Importance of the Coming Future

- In some sequences, to understand the meaning of the current event we need the knowledge of the “future” events.
- For example, to understand what or who is the "Teddy" mentioned in the following two sentences

*He said, "Teddy was a great President."*

*He said, "Teddy bears are on sale!"*
- we need to hear the rest of those sentences, or the "future" utterances. Only then, we understand the third word in either sentence.
- We do not have to guess or forecast the future. The above means that not all sequences could be meaningfully analyzed by scanning them from the left to the right.
- Sequences, like the two sentences above, must be consumed as the whole before they could be meaningfully analyzed and understood, or as we will see shortly, scanned in both directions.
- In the sequence analysis, we sometimes consider immediate neighbors, events, and sometime far away events.
- As we will see in this and the following lectures, there are various versions of deep learning networks called RNNs (Recurring NNs) that could deal with many of those situations.

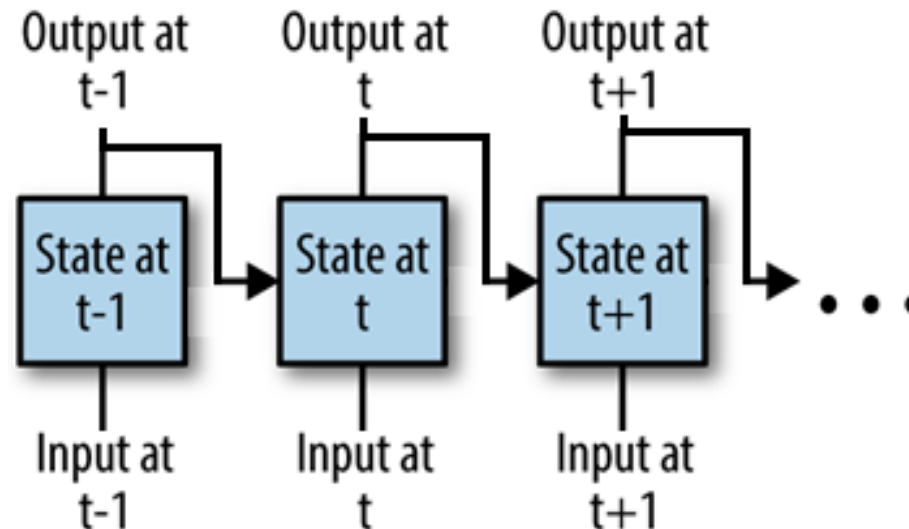
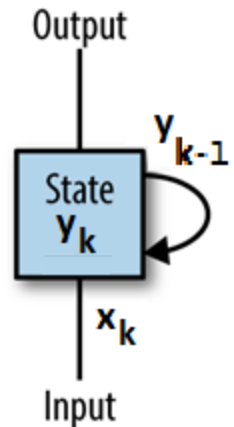
# Long distance relationships

- In some sequences (sentences) the meaning of the current event,  $y_k$ , is dependent on the current event,  $x_k$ , and close neighbors:  $y_{k-2}$ ,  $y_{k-1}$ ,  $y_{k+1}$ ,  $y_{k+2}$ .
- However, there are sequences (sentences) in which the dependences are very long range.
- For example, in the following two sentences:  
*The **cat**, which already ate Meow Mix and had ...., **was** still hungry.*  
*The **cats**, which already ate Meow Mix and had ...., **were** still hungry.*
- To finish the above arbitrarily long sentences with "**was** still hungry" or with "**were** still hungry", we need to keep track of the plurality of the subject word at the very beginning of the respective sentence.
- If the subject word is singular, e.g., "cat", we would use "**was**" at the end.
- If the subject word is plural, "cats", we would use "**were**" at the end.
- The distance between the subject and the verb in these sentences is arbitrary. This implies that a mechanism for dealing with long sequences has to possess a long memory.



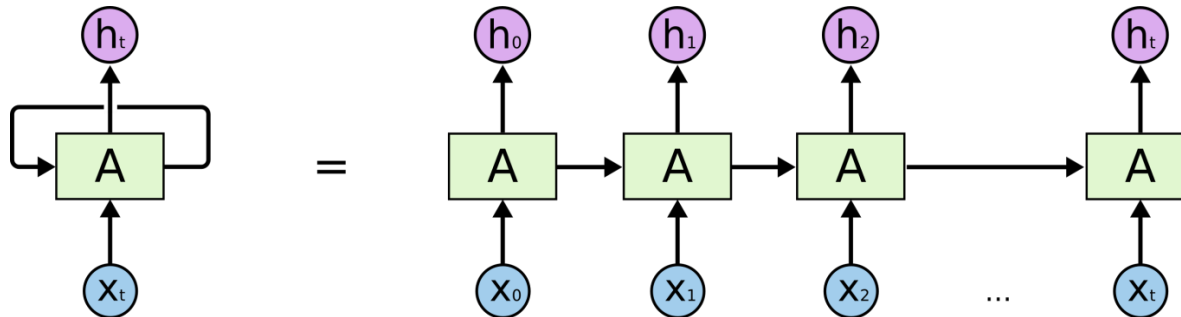
# Graphical representation of Recursive Relationships

- The recursive relationships can be represented by the diagram on the right. We call the value at step  $k$ ,  $y_k$ . The fact that  $y_k$  depends on the previous value  $y_{k-1}$  is represented by a looping-back arrow.
- At every position  $k$  (time  $t$  or  $t_k$ ) we also assume there is an internal state.
- As we move through positions  $k$  (time  $t$  or  $t_k$ ) the input  $x_k$  changes as well as the observed values  $y_k$  and the internal state. The internal state and the output are sometimes the same. The evolution of this recursive engine could also be presented as unrolled in  $k$  or  $time$ , using the following diagram:



# RNNs, unrolling through time

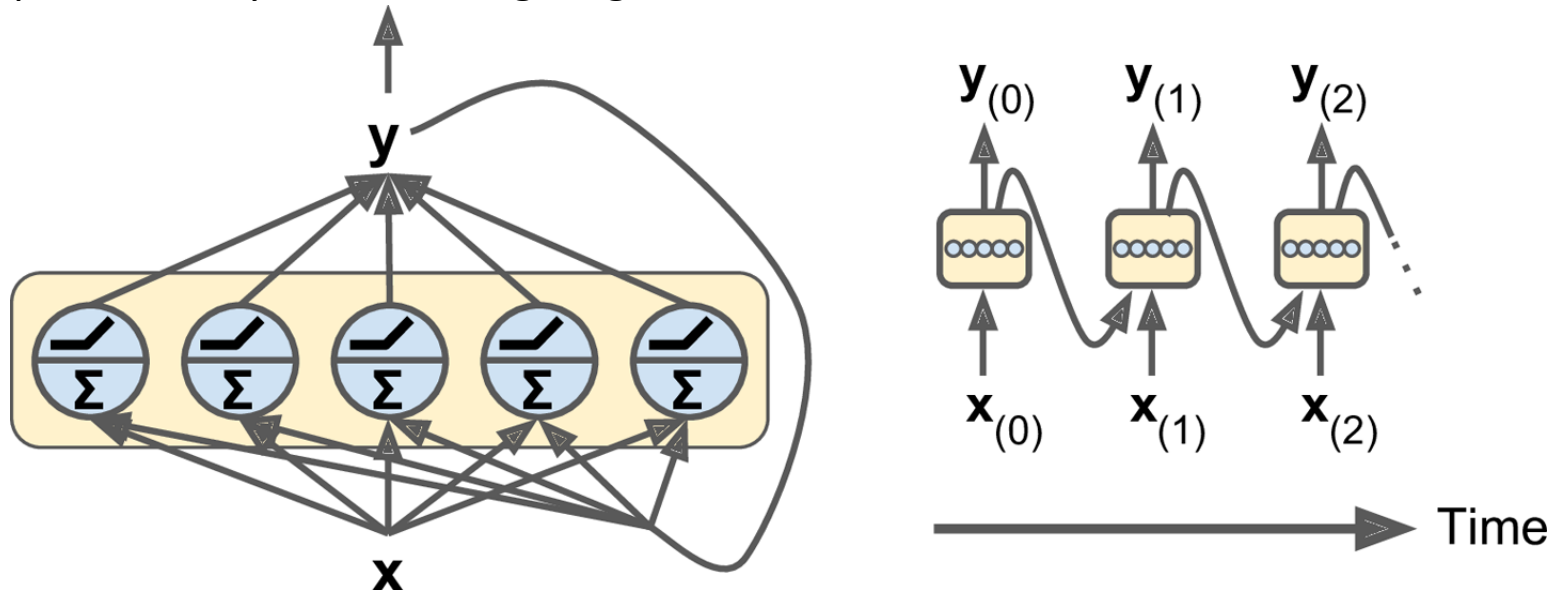
- Simple RNN can be represented as a loop where previous state gets fed into the current state or as an unrolled sequence of nodes.



- A recurrent neural network can be thought of as multiple copies of the same node, each passing a message to a successor. The diagram on the right represents what happens if we unroll the loop.
- This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. Unrolled (chained) neural network is not just a concept but could be an actual architecture used for the analysis of sequence data. Usually, unrolled recurrent networks require more memory .
- In the past few years, various forms of RNNs have been applied with a great success to a variety of problems: speech recognition, language modeling, text and speech translation, image captioning, and other.

# Simple Recursion as Recurrent Neural Network

- Every node of a recursive relationship could be implemented as a single neuron or as a collection (one or several layers) of neurons. Mapping of an input value  $x_k$  ( $x(t)$ ) and previous output  $y_{k-1}$  ( $y(t-1)$ ) into new output  $y_k$  ( $y(t)$ ) could then be represented by the following diagrams:



***A layer of recurrent neurons (left), and unrolled through time (right)***

- Each recurrent neuron has two sets of weights: one for the inputs  $x(t)$  and the other for the output of the previous time step,  $y(t-1)$ . Those weights are just like any other weights that we dealt with in the past. Eventually, they will be determined by the training process and backpropagation.

# Weights and Biases

- We will call those weight tensors  $W_x$  and  $W_y$ . Whether we consider the whole recurrent layer or just one recurrent neuron, we can place all the weight vectors in two weight matrices,  $W_x$  and  $W_y$ .
- The output vector of the whole recurrent layer can then be computed as shown in the following equation. Here  $\mathbf{b}$  is the bias vector and  $g(\cdot)$  is the activation function, e.g., *ReLU*, *tanh()*, or some other.

$$Y_{(t)} = g(X_{(t)}W_x^T + Y_{(t-1)}W_y^T + b)$$

- The above is different from our standard layer activation function, which does not have the second term  $Y_{(t-1)}W_y^T$ , where  $Y_{(t-1)}$  is the state of the previous stage or the state at the previous time.
- Just like in the feedforward neural networks, we can compute the output of a recurrent layer in one shot for a whole mini-batch of samples by placing all the inputs at time step  $t$  in an input matrix  $X_{(t)}$ . *Outputs of a layer of recurrent neurons for all instances in a mini-batch now read:*

$$Y_{(t)} = g(X_{(t)} \cdot W_x + Y_{(t-1)} \cdot W_y + b) = g([X_{(t)} | Y_{(t-1)}] \cdot W + b), \text{ where } W = \begin{bmatrix} W_x \\ W_y \end{bmatrix}$$

- Matrix  $[X_{(t)} | Y_{(t-1)}]$  is a concatenation of matrixes representing a mini batch of inputs at time  $t$  and the output vectors at the previous time  $t - 1$ .
- $W$  is a vertical concatenation of matrices combining weights for inputs  $X_{(t)}$  and outputs at time  $t - 1$ ,  $Y_{(t-1)}$

# Notation

In RNNs field , notation is complex and unfortunately not uniform.

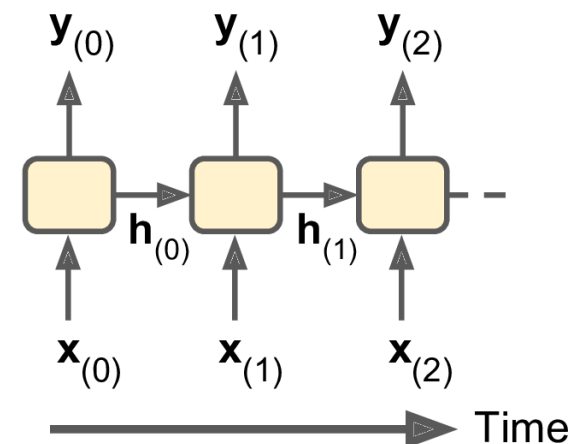
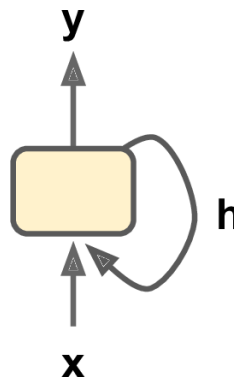
Some people use some symbols and other people use other symbols.

What we had on the previous page could be summarized as:

- $Y_{(t)}$  is an  $m \times n_{neurons}$  activation matrix containing the layer's outputs at time step  $t$  for each instance in the mini-batch ( $m$  is the number of instances in the mini-batch and  $n_{neurons}$  is the number of neurons in a layer at one node).
- $X_{(t)}$  is a vector of  $n_{inputs}$  containing the inputs for all instances ( $n_{inputs}$  is the number of input features).
- $W_x$  is an  $n_{inputs} \times n_{neurons}$  matrix containing the connection weights for the inputs of the current time step.
- $W_y$  is an  $n_{neurons} \times n_{neurons}$  matrix containing the connection weights for the outputs of the previous time step.
- $\mathbf{b}$  is a vector of size  $n_{neurons}$  containing each neuron's bias term.
- The weight matrices  $W_x$  and  $W_y$  are concatenated vertically into a single weight matrix  $\mathbf{W}$  of shape  $(n_{inputs} + n_{neurons}) \times n_{neurons}$ ,  $\mathbf{W} = \begin{bmatrix} W_x \\ W_y \end{bmatrix}$
- The notation  $[X_{(t)}|Y_{(t-1)}]$  represents the horizontal concatenation of the matrices  $X_{(t)}$  and  $Y_{(t-1)}$ .

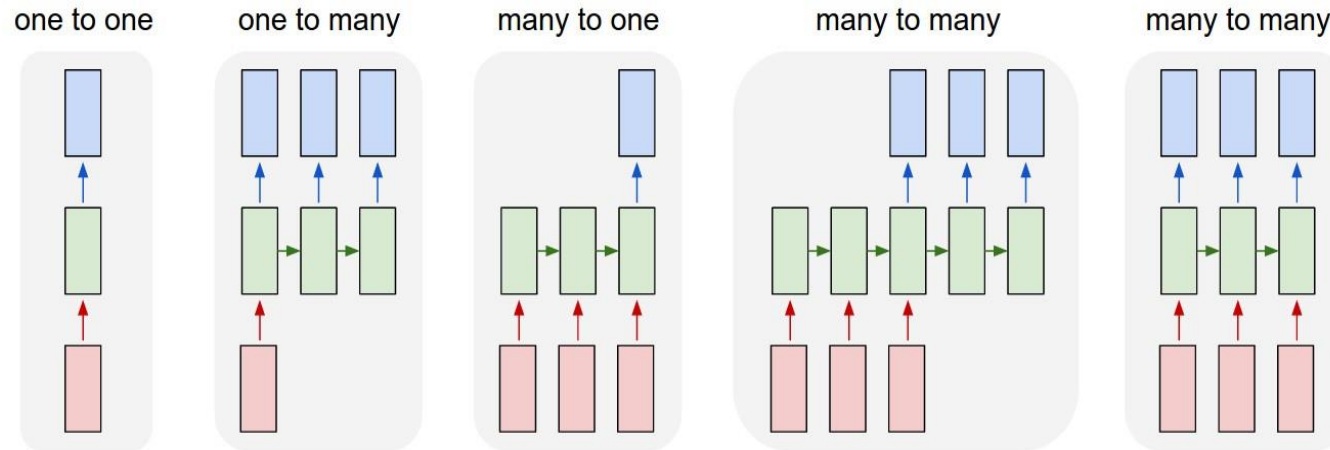
# Memory Cells

- $Y_{(t)}$  is a function of  $X_{(t)}$  and  $Y_{(t-1)}$ .  $Y_{(t-1)}$  is a function of  $X_{(t-1)}$  and  $Y_{(t-2)}$ , which is a function of  $X_{(t-2)}$  and  $Y_{(t-3)}$  and so on. This makes  $Y_{(t)}$  a function of all the inputs since time  $t = 0$  (that is,  $X_{(0)}, X_{(1)}, \dots, X_{(t)}$ ). At the first-time step,  $t = 0$ , there are no previous outputs, so they are typically all assumed to be zeros.
- Since the output of a recurrent neuron at time step  $t$  is a function of all the inputs from previous time steps, we say that **recurrent neurons have memory**.
- A part of a neural network that preserves some state across many time steps is called a **memory cell** (or simply a *cell*).
- In general, a cell's state at time step  $t$ , denoted  $h_{(t)}$  (the "h" stands for "hidden"), is a function of some inputs at that time step and cell's state at the previous time step:  $h_{(t)} = f(h_{(t-1)}, x_{(t)})$ . Cell's output at time step  $t$ , denoted  $y_{(t)}$  is also a function of the previous state and the current inputs.
- In the case of the basic cells, the output is simply equal to the hidden state, but in more complex cells this is not always the case.



# Different Inputs vs. Outputs in RNNs

- CNNs accept a fixed-sized vector as input (e.g., an image) and produce a fixed-sized vector as output (e.g., probabilities of different classes).
- RNNs operate over *sequences* of vectors: Sequences in the input, the output, or both.

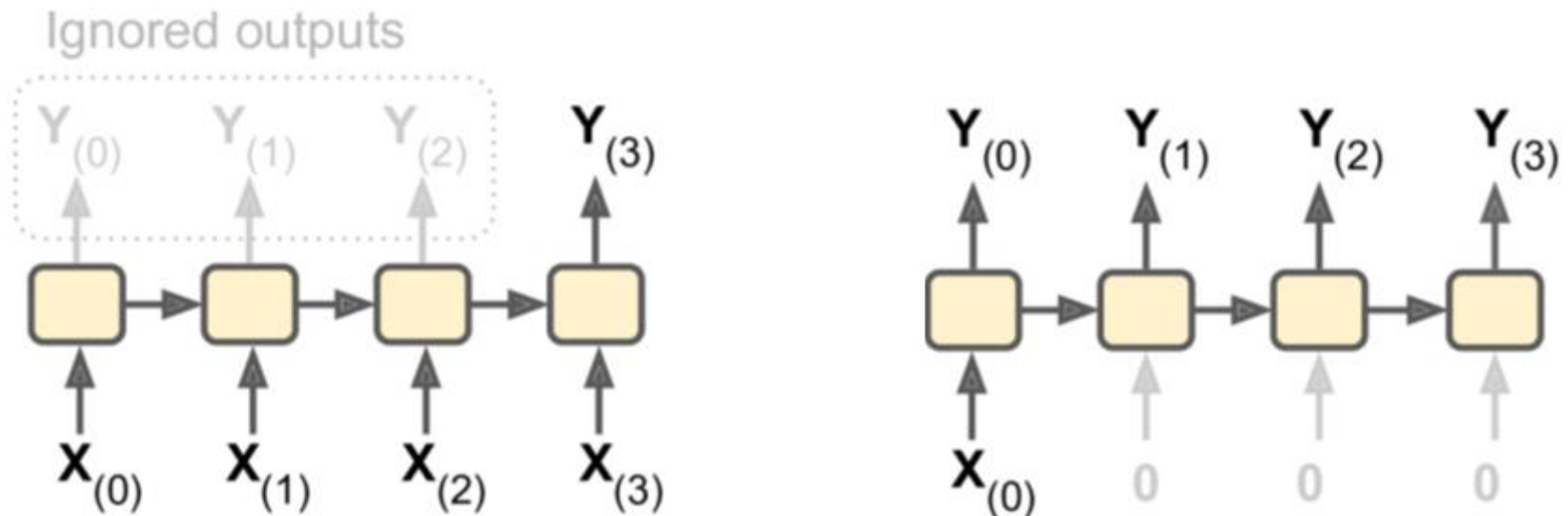


- Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state
- (1)** Vanilla mode of processing, fixed-sized input to fixed-sized output.
  - (2)** Sequence output (e.g., image captioning takes an image and outputs a sentence of words).
  - (3)** Sequence input (e.g., sentiment analysis, a sentence is classified as expressing positive or negative sentiment).
  - (4)** Sequence input and sequence output (e.g., Machine Translation: an RNN reads a sentence in English and outputs a sentence in French).
  - (5)** Synced sequence input and output (e.g. video classification where we wish to label each frame of the video).

There are (practically) no constraints on the lengths of sequences because the recurrent transformation (green) can be applied as many times as one likes.

# Inputs vs. Outputs, Again

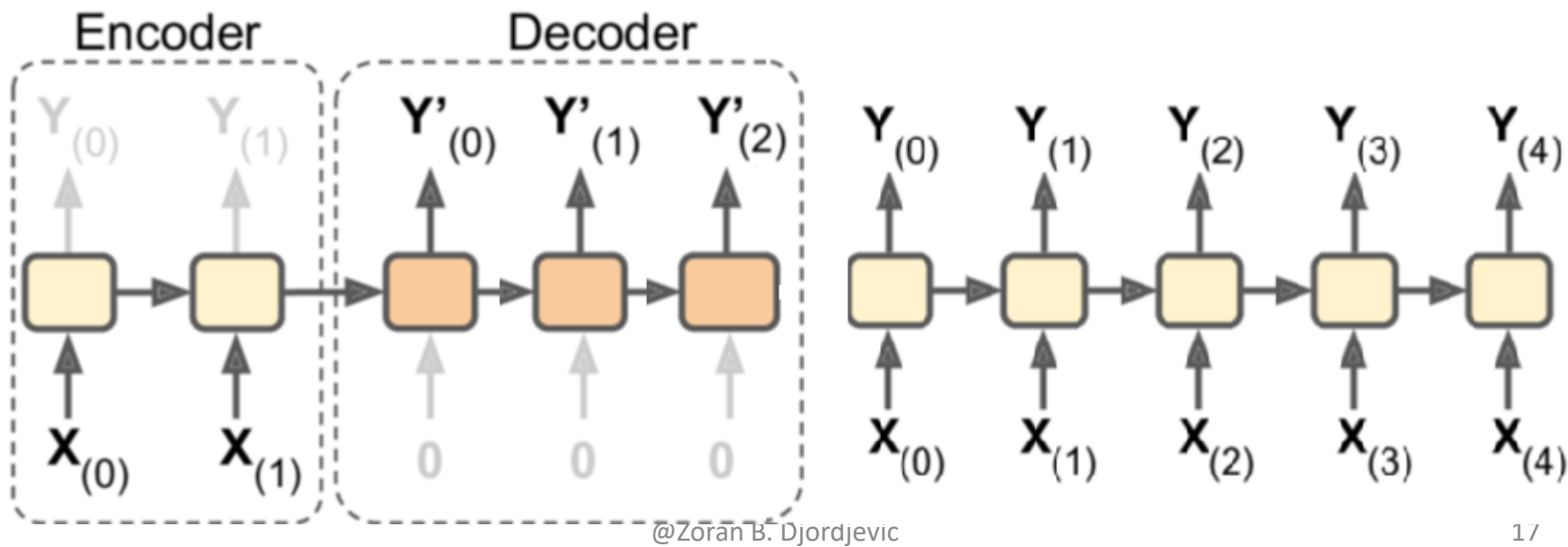
1. An RNN can be fed a sequence of inputs and ignore all outputs except for the last one (network on the left bellow).
  - This is a sequence-to-scalar (vector) network. We would use such network to predict tomorrow's temperature based on the temperatures over the last week or a month.
  - We could feed a similar network a sequence of words corresponding to a movie review, and the network would output a sentiment score (e.g., from  $-1$  [hate] to  $+1$  [love]).
2. Conversely, you could feed the network a single input at the first time-step (and zeros for all other time steps), and let it output a sequence (network on the right bellow).
  - This is a scalar (vector)-to-sequence network. For example, the input could be an image, and the output could be a caption for that image.





# Inputs vs. Outputs, Again

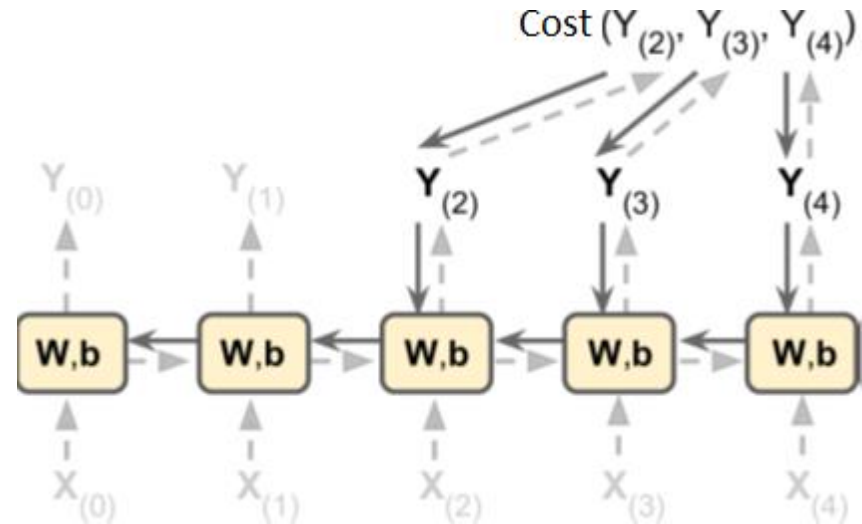
3. You could have a sequence-to-vector network, called an *encoder*, followed by a vector-to-sequence network, called a *decoder* (see the bottom-left network). For example, this can be used for translating a sentence from one language to another.
  - You would feed such network a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language.
  - This two-step model, called an Encoder–Decoder, works much better than trying to translate on the fly with a single sequence-to-sequence RNN (the top left on the previous slide). The last words of a sentence can affect the first words of the translation. You need to wait until you hear the whole sentence before translating it.
4. There are some scenarios where sequence to sequence RNNs (bottom-right below) give the best results.



# Training RNNs, Backpropagation through Time

- RNNs unrolled through time are trained using regular backpropagation.
- This strategy is called *backpropagation through time* (BPTT).

- Just like in the regular backpropagation, there is first a forward pass through the unrolled network (represented by the dashed arrows); then the cost function of the output sequence is evaluated
- $Cost(Y_{(0)}, Y_{(1)}, \dots, Y_{(N)})$



- The model parameters are updated using the gradients computed during BPTT.
- The gradients flow backward through all the outputs used by the *Cost* function, not just through the final output.
- In the above figure, the *Cost* function is computed using the last three outputs of the network,  $Y_{(2)}$ ,  $Y_{(3)}$ , and  $Y_{(4)}$ , so gradients flow through these three outputs, but not through  $Y_{(0)}$  and  $Y_{(1)}$ .
- Notice that the same parameters  $\mathbf{W}$  and  $\mathbf{b}$  are used at each time steps. The backpropagation includes the effect of all time steps.

# Typical *Cost Function*

- *Cost functions* are yours to choose.
- You define a *cost function* per every cell and then sum those over all outputs used for the evaluation of the network.
- Frequently used cost function is the cross entropy of the hidden state  $h_{(t)}$  and the cell output  $y_{(t)}$ :

$$L_{(t)}(h_t, y_{(t)}) = -y_{(t)} \log(h_{(t)}) - (1 - y_{(t)}) \log(1 - h_{(t)})$$

- Those entropies are then summed over all cells used in generation of the output sequence:

$$L = \sum_{t=k}^{T_y} L_{(t)}(h_{(t)}, y_{(t)})$$

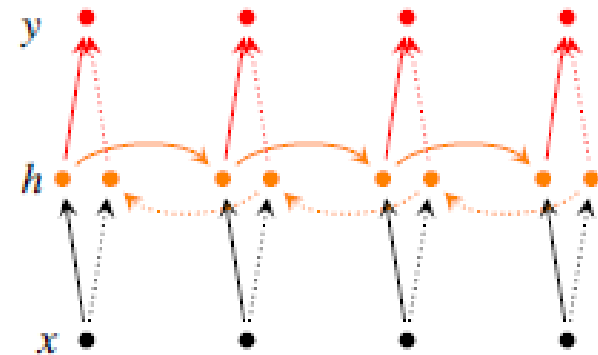
# RNN Architectures, Bi-directional RNNs

- So far, we considered RNNs that investigate the past events (words) to predict the next event (word) in the sequence (sentence). It is also possible to make predictions based on future events (words) with RNNs that read through the corpus backwards (as well).
- Irsoy et al. (<https://arxiv.org/pdf/1312.0493.pdf>) paper introduced a bi-directional recursive neural network. At each time-step,  $t$ , this network maintains two hidden layers, one for the left-to-right and another for the right-to-left propagation.
- To maintain two hidden layers at any time, this network consumes twice as much memory space for its weight and bias parameters.
- The final classification result,  $\hat{y}_t$ , is generated through combining the score results produced by both RNN hidden layers. Bi-directional RNNs cells are governed by the equations tracking forward and backward hidden states  $\vec{h}_t$  and  $\overleftarrow{h}_t$

$$\vec{h}_t = f(\vec{W}x_t + \vec{V}\vec{h}_{t-1} + \vec{b})$$

$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b})$$

$$\hat{y}_t = g(Uh_t + c) = g(U[\vec{h}_t; \overleftarrow{h}_t] + c)$$



A bi-directional RNN model

- Bi-directional RNNs show noticeable advantage in some notably language related, classification tasks like sentiment analysis. In some tasks, they do not prove useful.

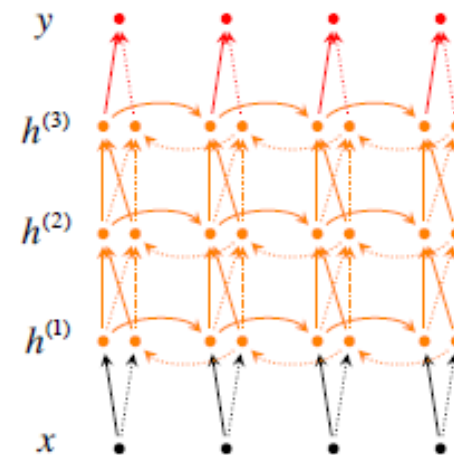
# Deep Bi-directional RNNs

- Just as a curiosity, one could extend the idea further and create multi-layer bi-directional RNN where each lower layer feeds the next layer. As shown in the figure below, in this network architecture, at time-step  $t$  each intermediate neuron receives one set of parameters from the previous time-step (in the same RNN layer), and two sets of parameters from the previous RNN hidden layer; one input comes from the left-to-right RNN and the other from the right-to-left RNN.
- In Deep RNN with  $L$  layers, the previous relationships are modified so that the input to each intermediate neuron at level  $i$  is the output of the RNN at  $i - 1$  at the same time-step,  $t$ . The output,  $\hat{y}_t$ , at time-step  $t$  is the result of propagating input parameters through all hidden layers.

$$\vec{h}_t^{(i)} = f(\vec{W}^{(i)} h_t^{(i-1)} + \vec{V}^{(i)} \vec{h}_{t-1}^{(i)} + \vec{b}^{(i)})$$

$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

$$\hat{y}_t = g(Uh_t + c) = g(U[\vec{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$



A deep bi-directional RNN with three layers.

- Deeper layers improve prediction accuracy due to their higher learning capacity. This implies that a large training corpus must be used to train the model.

# Issues with RNNs

- What we described in the past slides is history. In more recent practice, most of the work on sequences uses a slightly different network (layer) architecture called *Long Short-Term Memory* (LSTM) networks.
- The LSTM is a variant of recurrent neural network that works better in practice, owing to its more powerful update features.
- The main issue with RNNs is the lack of long-range memory.
- Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the next word in the sentence:

"the clouds are in the . . . ,"

- We don't need any further context – it is obvious the next word should be "sky". In such cases, where the gap between the relevant information and the place that needs that information is small, RNNs effectively learn to use the past information.

- There are cases where we need more context. Consider predicting the last word in the text:

"I grew up in France . . . . I speak fluent ...."

- Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back.
- It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.
- Unfortunately, as that gap grows, classical RNNs become unable to connect the information.
- Small gradients cannot propagate through long sequences of RNN cells.

# The Long Short-Term Memory (LSTM) Cell

# LSTM Cells

- The *Long Short-Term Memory* (LSTM) cell was proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber, and it was gradually improved over the years by several researchers, such as Alex Graves, Haşim Sak, Wojciech Zaremba, and others.
- If LSTM cells are treated as black boxes, they can be used like the basic RNN cells, except they perform much better.
- The training of LSTM cells converges faster.
- LSTM cells detect long-term dependencies in the data.
- In TensorFlow, Keras and PyTorch, you can simply use a `LSTM` instead of a `RNN Cell`:

```
lstm = tf.keras.layers.LSTM(4, return_sequences=True, return_state=True)
```

- LSTM cells manage two state vectors. For performance reasons they are kept separate by default. When creating the `LSTM cell`, you can change this default behavior by setting `stateful=False`

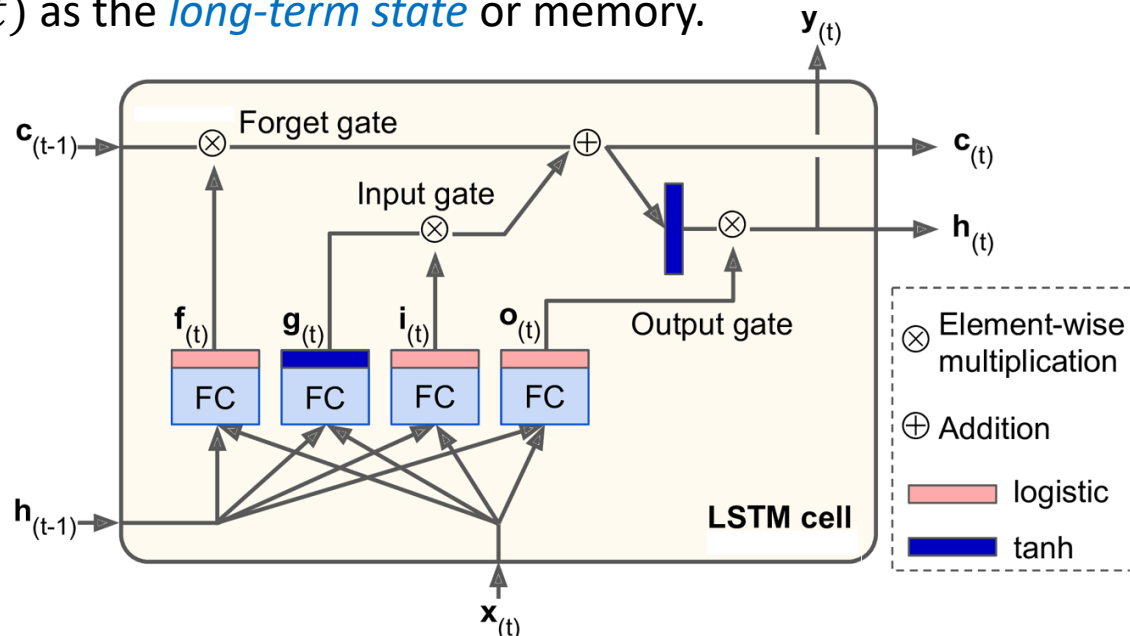
## References

- "Long Short-Term Memory," S. Hochreiter and J. Schmidhuber (1997).
- "Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling," H. Sak et al. (2014).
- "Recurrent Neural Network Regularization," W. Zaremba et al. (2015).



# Externals of LSTM cell

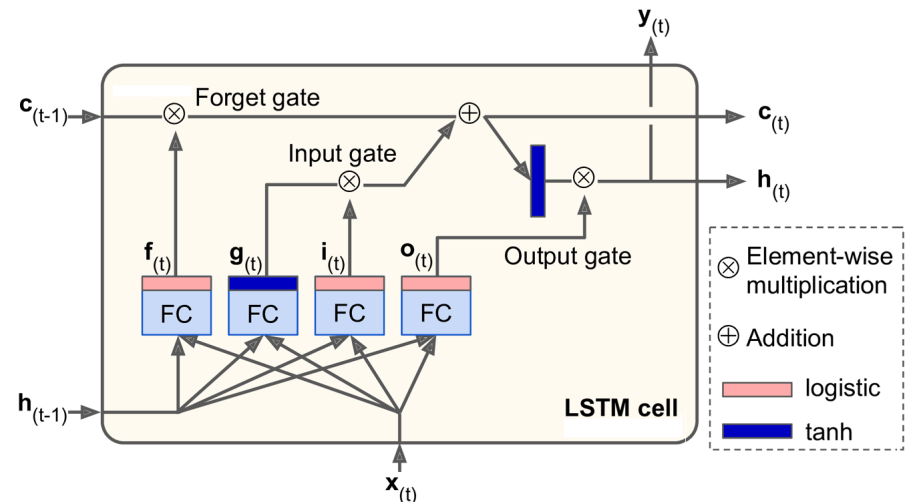
- As a black box, the LSTM cell looks exactly like a regular cell, except that its state is split in two vectors:  $\mathbf{h}(t)$  and  $\mathbf{c}(t)$  ("c" stands for "cell" or "carry").
  - $\mathbf{h}(t)$  as the **short-term (history) state** and
  - $\mathbf{c}(t)$  as the **long-term state** or memory.



- Logistic (**sigmoid**) layers produce values between 0 and 1. They decide how much of each component should be let through: 0 - don't let anything through, 1 - pass everything.
- tanh** layers produces the values between -1 and 1.
- This Figure is from chapter 15 of Aurélien Géron's book, 2<sup>nd</sup> Edition

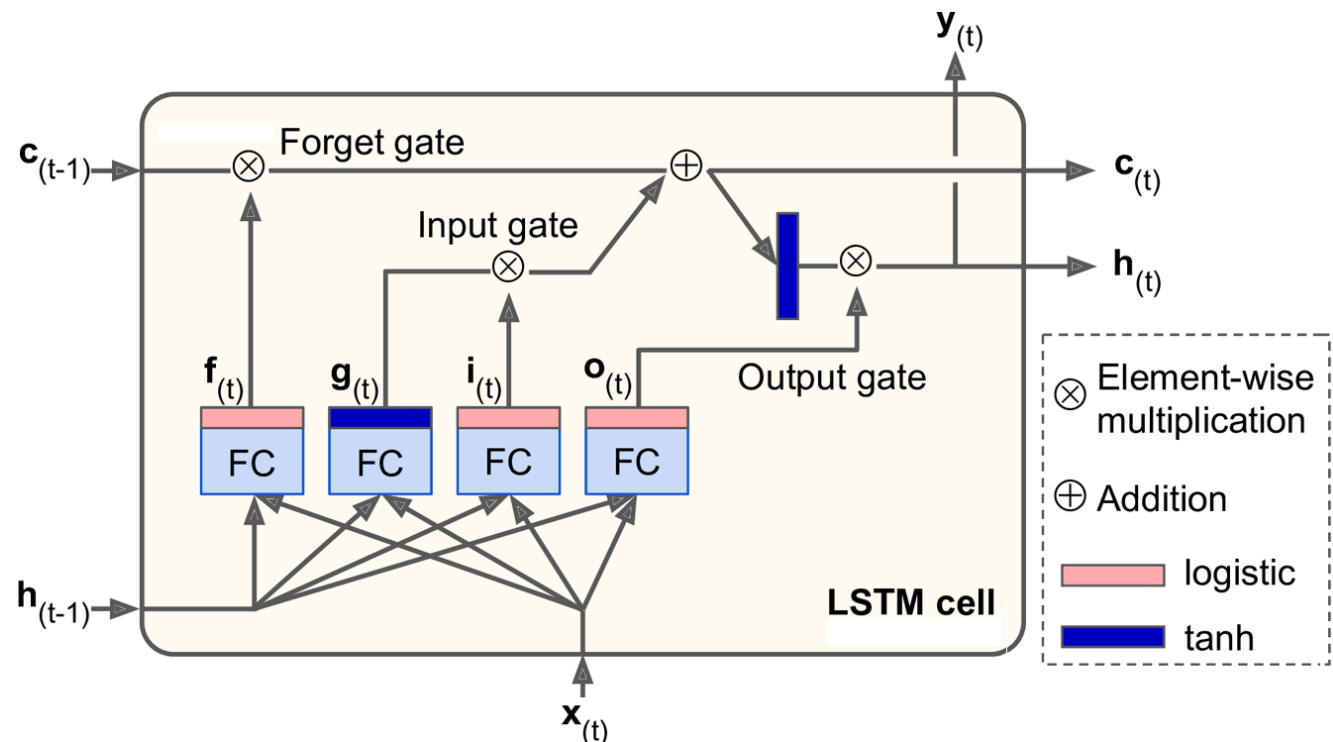
# Internals of LSTM cell

- The key idea is that the network can learn what to store in the long-term state  $c(t)$ , what to throw away, and what to read from it into the output  $y(t)$ .
- As the *long-term state*  $c(t-1)$  traverses the network from left to right, it is first affected by the *forget gate*, which might force it to drop some memories. Forget gate acts on  $c(t-1)$  through element-wise multiplication.
- Modified  $c(t-1)$  is then added some new memories, selected by the *input gate*.
- The result,  $c(t)$ , is sent straight out, without further transformation.
- At each time step, some memories are dropped, and some memories are added.
- After the addition operation, the *long-term state*,  $c(t)$ , is copied and passed through the  $\tanh()$  function, and then the result is filtered by the *output gate*, though element-wise multiplication.
- This produces the *short-term state*  $h(t)$  (which is identical to the cell's *output* for this time step,  $y(t)$ ).



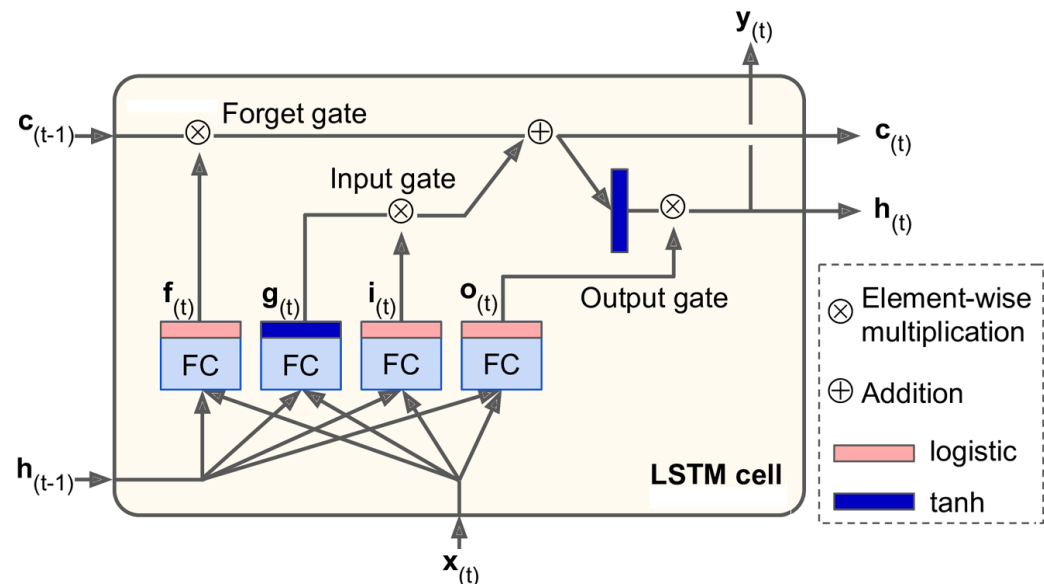
# Main Layer and Its output $g(t)$

- The current input vector  $x(t)$  and the previous short-term state  $h(t-1)$  are fed to four different fully connected layers. They all serve different purposes.
- The main layer is the one that outputs  $g(t)$ . It has the usual role of analyzing the current inputs  $x(t)$  and the previous (short-term) state  $h(t-1)$ .
- In a basic RNN cell, there is nothing else but this layer, and its output goes straight out to  $y(t)$  and  $h(t)$ .
- In contrast, in an LSTM cell this layer's output does not go straight out, but instead it is partially stored in the long-term state.



# Gates

- The other three layers are called *gate controllers*. They use the logistic (**sigmoid**) activation functions with outputs between 0 to 1. Those outputs:  $f(t)$ ,  $i(t)$  and  $o(t)$  are fed to element-wise multiplication operations. If they output 0, they close the gate, and if they output 1, they open it. Specifically:
  - The **forget gate** (expressed by  $f(t)$ ) controls which parts of the long-term state should be erased.
  - The **input gate** (expressed by  $i(t)$ ) controls which parts of  $g(t)$  should be added to the long-term state (that is why we say it is only "partially stored").
  - Finally, the **output gate** (expressed by  $o(t)$ ) controls which parts of the long-term state should be read and output at this time step (both to  $h(t)$  and  $y(t)$ ).
- An LSTM cell can learn to recognize an important input (that's the role of the input gate), store it in the long-term state, learn to preserve it for as long as it is needed (that's the role of the forget gate), and learn to extract it whenever it is needed.
- LSTMs have been very successful at capturing long-term patterns in time series, long texts, audio recordings, and other applications.



# Equations Describing LSTM Cell

- Equations below summarize how to compute the cell's long-term state  $c(t)$ , its short-term state  $h(t)$ , and its output  $y(t)$  at each time step for a single instance.

$$\mathbf{i}_{(t)} = \sigma(\mathbf{W}_{xi}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_i)$$

$$\mathbf{f}_{(t)} = \sigma(\mathbf{W}_{xf}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_f)$$

$$\mathbf{o}_{(t)} = \sigma(\mathbf{W}_{xo}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{ho}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_o)$$

$$\mathbf{g}_{(t)} = \tanh(\mathbf{W}_{xg}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_g)$$

$$\mathbf{c}_{(t)} = \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)}$$

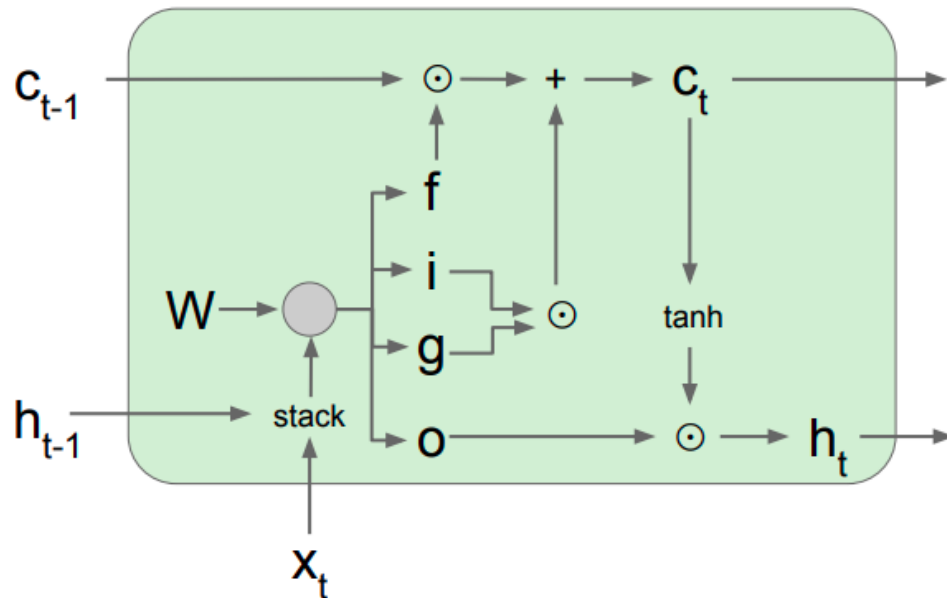
$$\mathbf{y}_{(t)} = \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh(\mathbf{c}_{(t)})$$

- $\mathbf{W}_{xi}$ ,  $\mathbf{W}_{xf}$ ,  $\mathbf{W}_{xo}$ ,  $\mathbf{W}_{xg}$  are the weight matrices of each of the four layers for their connection to the input vector  $\mathbf{x}(t)$ .
- $\mathbf{W}_{hi}$ ,  $\mathbf{W}_{hf}$ ,  $\mathbf{W}_{ho}$ , and  $\mathbf{W}_{hg}$  are the weight matrices of each of the four layers for their connection to the previous short-term state  $\mathbf{h}(t-1)$ .
- $\mathbf{b}_i$ ,  $\mathbf{b}_f$ ,  $\mathbf{b}_o$ , and  $\mathbf{b}_g$  are the bias terms for each of the four layers. Note that TensorFlow initializes  $\mathbf{b}_f$  to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training.

# Another Graph of LSTM Cell

## Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$
$$c_t = f \odot c_{t-1} + i \odot g$$
$$h_t = o \odot \tanh(c_t)$$

- **Forget gate**,  $f(t)$ , controls which parts of the long-term state should be erased.
- **Input gate**,  $i(t)$ , controls which parts of  $g(t)$  should be added to the long-term state
- **output gate**,  $o(t)$ , controls which parts of the long-term state should be read and output at this time step (both to  $h(t)$  and  $y(t)$ ).

# Peephole Connections

- In a basic LSTM cell, the gate controllers can look only at the input  $\mathbf{x}(t)$  and the previous short-term state  $\mathbf{h}(t-1)$ . It may be a good idea to give them a bit more context by letting them peek at the long-term state as well. This idea was proposed by Felix Gers and Jurgen Schmidhuber in 2006
- They proposed an LSTM variant with extra connections called *peephole connections*: the previous long-term state  $\mathbf{c}(t-1)$  is added as an input to the controllers of the forget gate and the input gate, and the current long-term state  $\mathbf{c}(t)$  is added as input to the controller of the output gate.
- To implement peephole connections in TensorFlow or Keras, instead of `LSTMCell` use the `PeepholeLSTMCell`

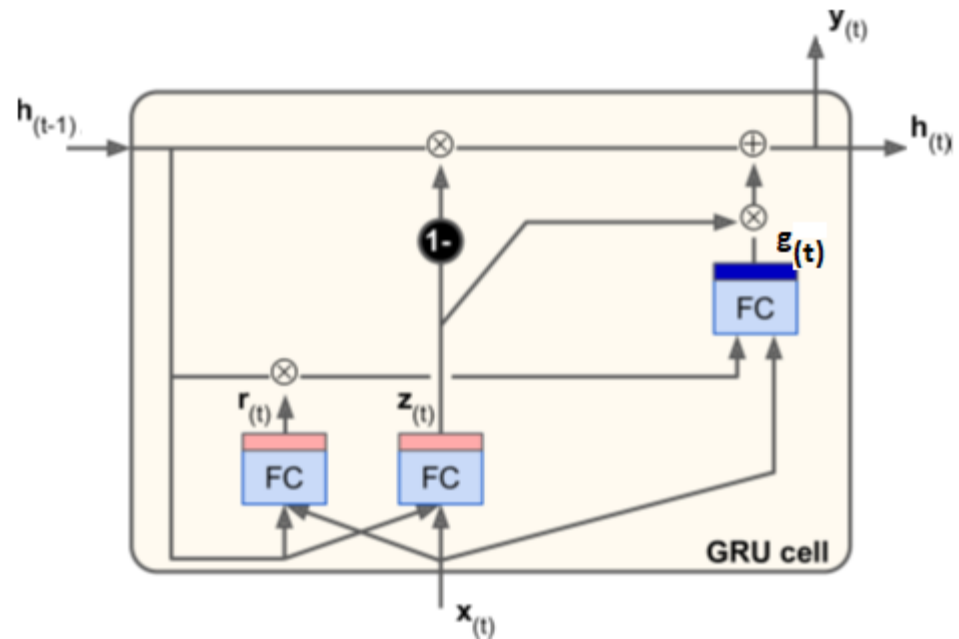
```
lstm_cell = tf.keras.experimental.PeepholeLSTMCell
```

- There are many other variants of the LSTM cell.
- One particularly popular variant is the GRU cell

# Gated Recurrent Unit

- The *Gated Recurrent Unit* (GRU) cell is a simplified LSTM and was proposed by Kyunghyun Cho et al. in a 2014 paper. (<https://arxiv.org/pdf/1406.1078.pdf>)

Both state vectors are merged into a single vector  $\mathbf{h}(t)$ . A single gate controller  $\mathbf{z}(t)$  controls both the forget gate and the input gate. If the gate controller outputs a 1, the input gate is open and the forget gate is closed. If it outputs a 0, the opposite happens. In other words, whenever a memory must be stored, the location where it will be stored is erased first. There is no output gate; the full state vector is output at every time step. There is a new gate controller  $\mathbf{r}(t)$  that controls which part of the previous state will be shown to the main layer,  $\mathbf{g}(t)$ , with  $\tanh()$  activation.





# GRU Cell

- Equations governing behavior of a GRU cell are shown below

$$\mathbf{z}_{(t)} = \sigma(\mathbf{W}_{xz}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_z)$$

$$\mathbf{r}_{(t)} = \sigma(\mathbf{W}_{xr}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_r)$$

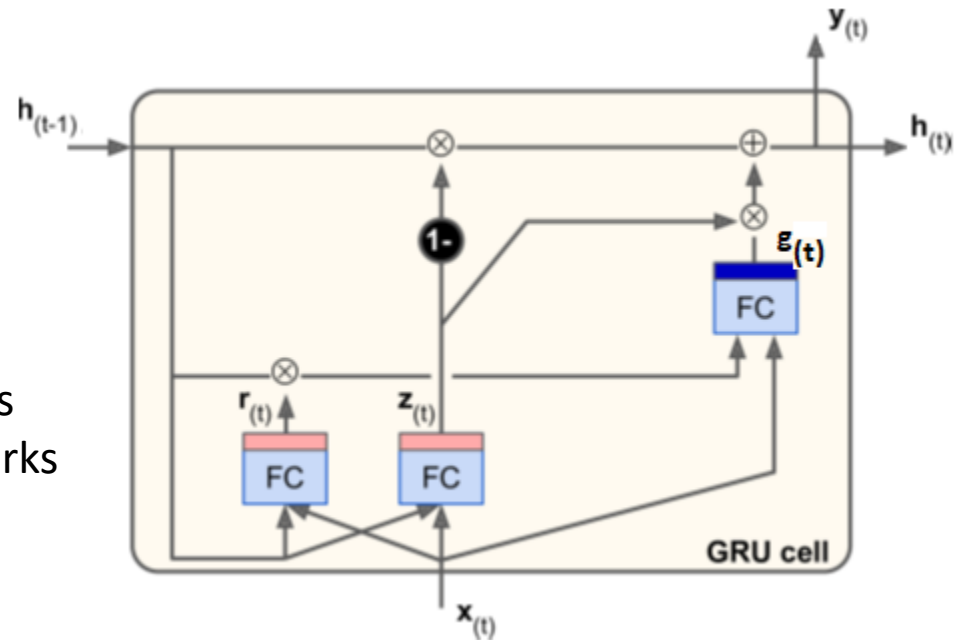
$$\mathbf{g}_{(t)} = \tanh(\mathbf{W}_{xg}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \cdot (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g)$$

$$\mathbf{h}_{(t)} = (1 - \mathbf{z}_{(t)}) \otimes \mathbf{h}_{(t-1)} + \mathbf{z}_{(t)} \otimes \mathbf{g}_{(t)}$$

- Creating a GRU cell in Keras with:

```
gru_cell = tf.keras.layers.GRUCell(  
    units=n_neurons)
```

- LSTM or GRU cells are the main drivers behind the success of recursive networks in recent years, especially in *natural language processing* (NLP).



# Where were LSTMs Used

LSTMs were used by billions of people through products of the world's largest public companies. For example:

- Excellent speech recognition on over several billion Android and iPhone phones ([Siri & Quicktype](#)) (since mid 2015),
- Excellent machine translation through [Google Translate and similar products](#) (since Nov 2016),
- Machine translation through [Facebook, Google, Amazon and other products](#) (several billion LSTM-based translations per day since 2017),
- Answers of [Amazon's Alexa](#), [Siri](#) and numerous other applications.

# Time series analysis with LSTM

# An example of time series analysis with LSTM

- We implied that LSTMs could deal with (predict) behavior of time sequences.
- Time series analysis refers to the analysis of change in the trend of the data over a time period.
- Time series analysis has a variety of applications. One such application is the prediction of the future value of an item based on its past values.
- We will demonstrate time series analysis with the help of an LSTM network. We will be predicting the future stock prices of the Apple Company (AAPL), based on its stock prices of the past several years.
- For various manipulations we will need `Numpy` and will plot the results using `matplotlib`.
- We will also use Keras API for LSTM layers.
- As a dataset we will fetch a long string of stock values for a company at Yahoo Finance. For example: <https://finance.yahoo.com/quote/AAPL/history?p=AAPL&.tsrc=fin-srch>
- We could adjust the start date and the end data of the report and download the dataset. We could for example select a 8+ year period between November 4th, 2016 to September 13th, 2024.
- Downloaded data will be stored in the file `APPL.csv`
- To test the predictive power of LSTM, as testing data, we will use the Apple stock prices for the period between September 13<sup>st</sup> to November 6<sup>th</sup>, 2024. We will download the actual stock prices for that period, as a separate file `APPL_test.csv`

# Loading the Data

- We import standard libraries:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd
```

- Downloaded data are loaded in a panda. Besides the date index, downloaded data have **6 features**: Open, High, Low, Close, Adj Close, Volume. We do not care about all of them, now. We will look only at the opening price and ignore the others.

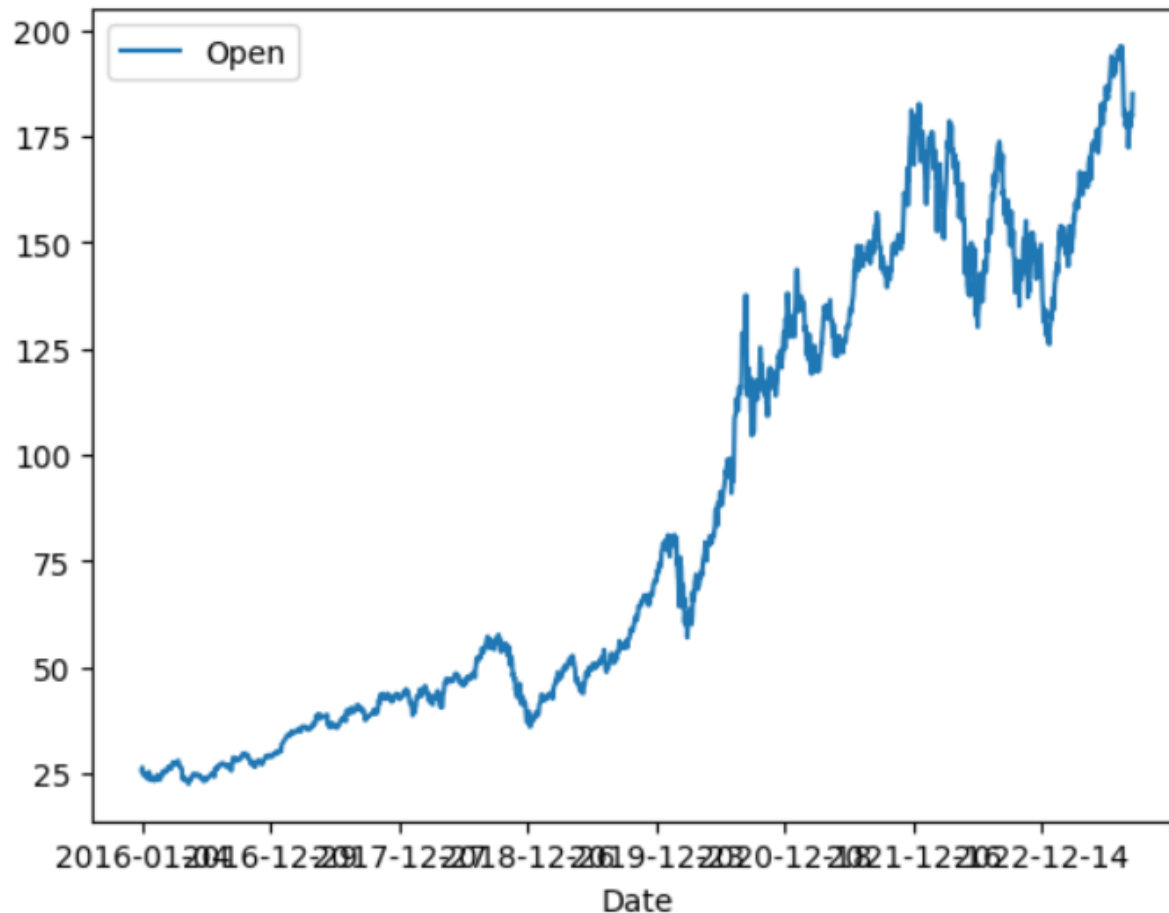
```
apple_training_complete = pd.read_csv(r'AAPL.csv')
apple_training_processed = apple_training_complete.iloc[:, 1:2].values
print(apple_training_processed)
print("length of the dataset: ", apple_training_processed.size)
```

```
[[ 25.6525  ]
 [ 26.4375  ]
 [ 25.139999]
 ...
 [180.089996]
 [179.699997]
 [184.940002]]
length of the dataset: 1928
```

# Visualize the data

- Stock prices are highly non-linear and most probably impossible to predict. If we plot all of Open prices since Jan 4<sup>nd</sup>, 2016 to August 31<sup>st</sup>, 2023, we will get a diagram like the following.

```
apple_training_complete.plot(x='Date', y='Open')
```



# Normalizing (Scaling) the Data

- We want to normalize our data and will use SciKitLearn `MinMaxScaler` from `sklearn.preprocessing` library

```
from sklearn.preprocessing import MinMaxScaler  
scaler = MinMaxScaler(feature_range = (0, 1))
```

```
apple_training_scaled = scaler.fit_transform(apple_training_processed)
```

- In a time series problem, we usually predict a value at time  $T$ , based on the data from the previous  $N$  time steps.
- In our case, we will predict `Open` price on day  $T$ , based on `Open` prices on days  $T-1, T-2, \dots, T-N$ .
- The number of steps (days),  $N$ , could be arbitrary. Concretely, we will predict the opening stock price on a day based on the opening stock prices for the past 60 days.
- One presumes that larger  $N$  will give higher precision of the prediction. However, larger  $N$  will most probably result in a longer training and possibly some loss of numerical precision. In practice, we conduct experiments with different  $N$  values and choose an optimal value.

# Feature Set and Labels

- In the script below we create two lists: `feature_set` and `labels`. There are 1928 records in the training data. We execute a loop that starts from 61st record and stores 60 records at the beginning to the `feature_set` list. The 61st record is stored in the `labels` list. Note that `apple_trained_scaled` has shape `(1928,1)`. It is a vertical vector of vectors.

```
features_set = []
labels = []
for i in range(60, 1928):
    features_set.append(apple_training_scaled[i-60:i, 0])
    labels.append(apple_training_scaled[i, 0])
```

- Each 60-element vector in the `feature_set`, will serve as the current data we will use to train the network to predict the value on 61-st position, contained in the `labels` list. Notice that values in 60-element vectors are overlapping. Those are rolling intervals of length 60.
- We need to convert both the `feature_set` and the `labels` lists to the Numpy arrays before we can use them for training. This is accomplished with the following line:

```
features_set, labels = np.array(features_set), np.array(labels)
```

- What are the dimensions of the `features_set`?

```
print(features_set.shape)
(1868, 60)
```

- We have transformed a 1868 long vertical vector `apple_trained_scaled` into a collection (a Numpy array) of 1868 horizontal vectors. `labels` is a vertical array of 1-dim vectors.

```
print(labels.shape)
print(labels)
(1868, 1)
[[0.01345737]
 [0.00988904]
 . . . .]
```



# Convert Training Data to Right Shape

- In order to train LSTM, we need to convert our data into the shape accepted by the LSTM. LSTM accepts a three-dimensional format.
- The first dimension is the number of records or rows in the dataset which is 1868 in our case.
- The second dimension is the number of time steps which is 60.
- The last dimension is the number of indicators. Since we are only using one feature, i.e., Open price, the number of indicators will be 1.
- We create the 3d Numpy array by using `np.reshape()` function:

```
features_set = np.reshape(features_set, (features_set.shape[0],  
                                         features_set.shape[1], 1))
```

# Building the LSTM Network

- We will use tf.Keras API. The LSTM model we will create is sequential model with multiple layers. We will use four LSTM layers followed by a dense layer that predicts the future stock price. Let's first import the libraries that we need in order to create our model:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Dropout
```

- As a first step, we need to instantiate the `Sequential` class. This is our model class and we will add LSTM, Dropout and Dense layers to this model. The first parameter to the LSTM layer is the number of neurons or nodes that we want in the layer. The second parameter is `return_sequences`, which is set to `True` since we will add more layers to the model.
- The first parameter to the `input_shape` is the number of time steps while the last parameter is the number of indicators (dimension of the label).

```
model = Sequential()
model.add(LSTM(units=50, return_sequences=True,
               input_shape=(features_set.shape[1], 1), unroll=False))
```

- `return_sequences=True`: Boolean. Whether to return the last output in the output sequence, or the full sequence. That return sequences return the hidden state output for each input time step. When stacking LSTM layers you must set `return_sequence` to `True`.
- We will leave `unroll=False`: Boolean (default `False`). If `True`, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up an RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.
- The number of neurons, `units = 50`, in every LSTM subcell, is somewhat arbitrary. You should play with that parameter and find an optimal value

# Adding Dropout and Further LSTM Layers

- To avoid over-fitting, we will add dropout layers.

```
model.add(Dropout(0.2))
```

```
model.add(LSTM(units=50, return_sequences=True))
```

```
model.add(Dropout(0.2))
```

```
model.add(LSTM(units=50, return_sequences=True))
```

```
model.add(Dropout(0.2))
```

```
model.add(LSTM(units=50))
```

```
model.add(Dropout(0.2))
```

## Adding a Dense Layer

- We add a Dense layer at the end of the model. The number of neurons in the Dense layer will be set to 1 since we want to predict a single value in the output.

```
model.add(Dense(units = 1))
```

## model.summary()

```
model.summary()  
Model: "sequential"
```

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 60, 50)	10400
dropout (Dropout)	(None, 60, 50)	0
lstm_1 (LSTM)	(None, 60, 50)	20200
dropout_1 (Dropout)	(None, 60, 50)	0
lstm_2 (LSTM)	(None, 60, 50)	20200
dropout_2 (Dropout)	(None, 60, 50)	0
lstm_3 (LSTM)	(None, 50)	20200
dropout_3 (Dropout)	(None, 50)	0
dense (Dense)	(None, 1)	51
=====		
Total params: 71,051		
Trainable params: 71,051		
Non-trainable params: 0		

# Model Compilation and Training

- Finally, we need to compile the LSTM model before we could train it on the training data. The following code compiles the model.

```
model.compile(optimizer = 'adam', loss = 'mean_squared_error')
```

- Now is the time to train the model. We call the `fit()` method on the model and pass to it our training features and labels as shown below:

```
history = model.fit(features_set, labels, epochs = 100, batch_size = 32)
Epoch 1/100
 64/64 [=====>.....] - ETA: 1s 23ms/step - loss: 0.0051
. . . .
Epoch 100/100
64/64 [=====] - 1s 23ms/step - loss: 3.7179e-04
```

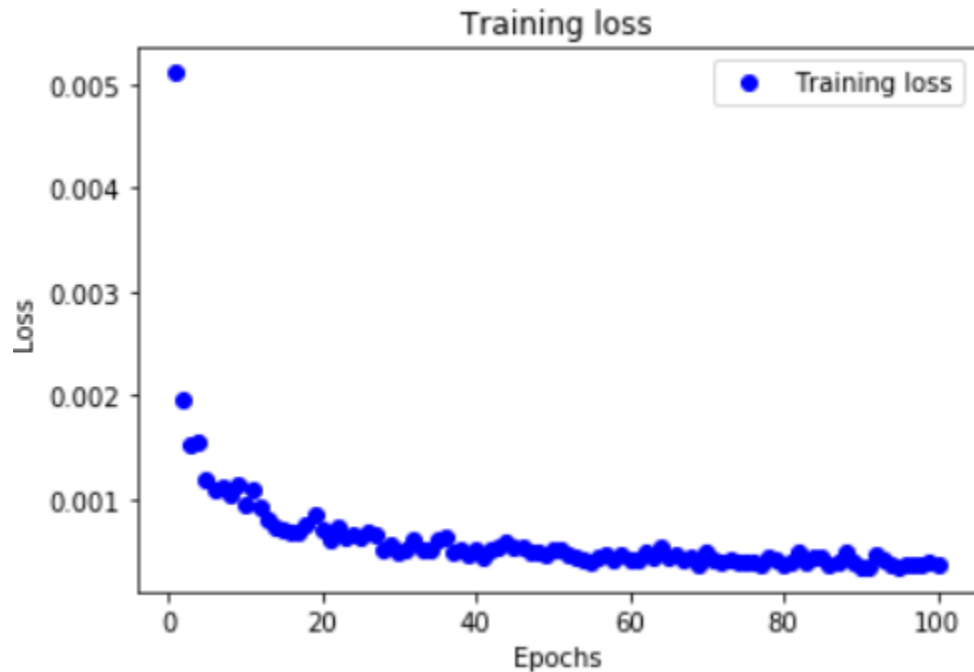
- Our `history` object contains only the training `loss` information:

```
history_dict = history.history
history_dict.keys()
dict_keys(['loss'])
```

- Had we had validation data passed to the training process, we would have had validation loss and perhaps accuracy available as well.

# Training Loss as a function of epoch

```
history_dict = history.history
history_dict.keys()
dict_keys(['loss'])
import matplotlib.pyplot as plt
loss = history.history['loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.title('Training loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



# Testing LSTM Network with Test Data

- We have successfully trained our LSTM, now we will test the performance of our network on the test set containing the opening stock prices from the beginning of September to 20<sup>th</sup> of October 2023.
- As we did with the training data, we need to convert our test data to the right format. We first import our test data. The test data contain only prices for 34 days and has no overlap with the training data.

```
apple_testing_complete = pd.read_csv(r'AAPL_test.csv')
apple_testing_processed = apple_testing_complete.iloc[:, 1:2].values
print("Number of data points: ", apple_testing_processed.size)
Number of data points: 34
```

- For each day in the test set, we want our feature set to contain the opening stock prices for the previous 60 days, what is almost 3 months.
- To accomplish this, we need to concatenate our training data and test data before preprocessing. The following does it:

```
apple_total = pd.concat((apple_training_complete['Open'],
                        apple_testing_complete['Open']), axis=0)
```

# Prepare the Test Inputs

- The input for each day should contain the opening stock prices for the previous 60 days. That means we need opening stock prices for the 34 test days and stock prices from the last 60 days from the training set. Execute the following code to fetch those 94 values.

```
test_inputs = apple_total[len(apple_total) -  
                           len(apple_testing_complete) - 60:].values
```

- As we did for the training set, we need to scale our test data. Execute the following:

```
test_inputs = test_inputs.reshape(-1,1)  
test_inputs = scaler.transform(test_inputs)
```

- We scaled our data, now let's prepare our final test input set that will contain previous 60 stock prices from the training data. Execute the following:

```
test_features = []  
for i in range(60, 94):  
    test_features.append(test_inputs[i-60:i, 0])
```

- Finally, we need to convert our data into the three-dimensional format which can be used as input to the LSTM. Execute the following code

```
test_features = np.array(test_features)  
test_features = np.reshape(test_features, (test_features.shape[0],  
test_features.shape[1], 1))  
print(test_features.shape)  
(34, 60, 1)
```



# Testing Predictive Power of LSTM

- We preprocessed our test data and now we can use it to make predictions. To do so, we simply need to call the `predict()` method on the model that we trained.

```
predictions = model.predict(test_features)
```

- Since we scaled our data, the predictions made by the LSTM are also scaled. We need to reverse the scaled prediction back to their actual values. To do so, we can use the `inverse_transform()` method of the `scaler` object we created during training. This is accomplished with the following

```
predictions = scaler.inverse_transform(predictions)
```

- Finally we plot the predicted values and compare them with the actual values.

```
plt.figure(figsize=(10,6))
```

```
plt.plot(apple_testing_processed, color='blue', label='Actual Apple Stock Price')
```

```
plt.plot(predictions , color='red', label='Predicted Apple Stock Price')
```

```
plt.title('Apple Stock Price Prediction')
```

```
plt.xlabel('Date')
```

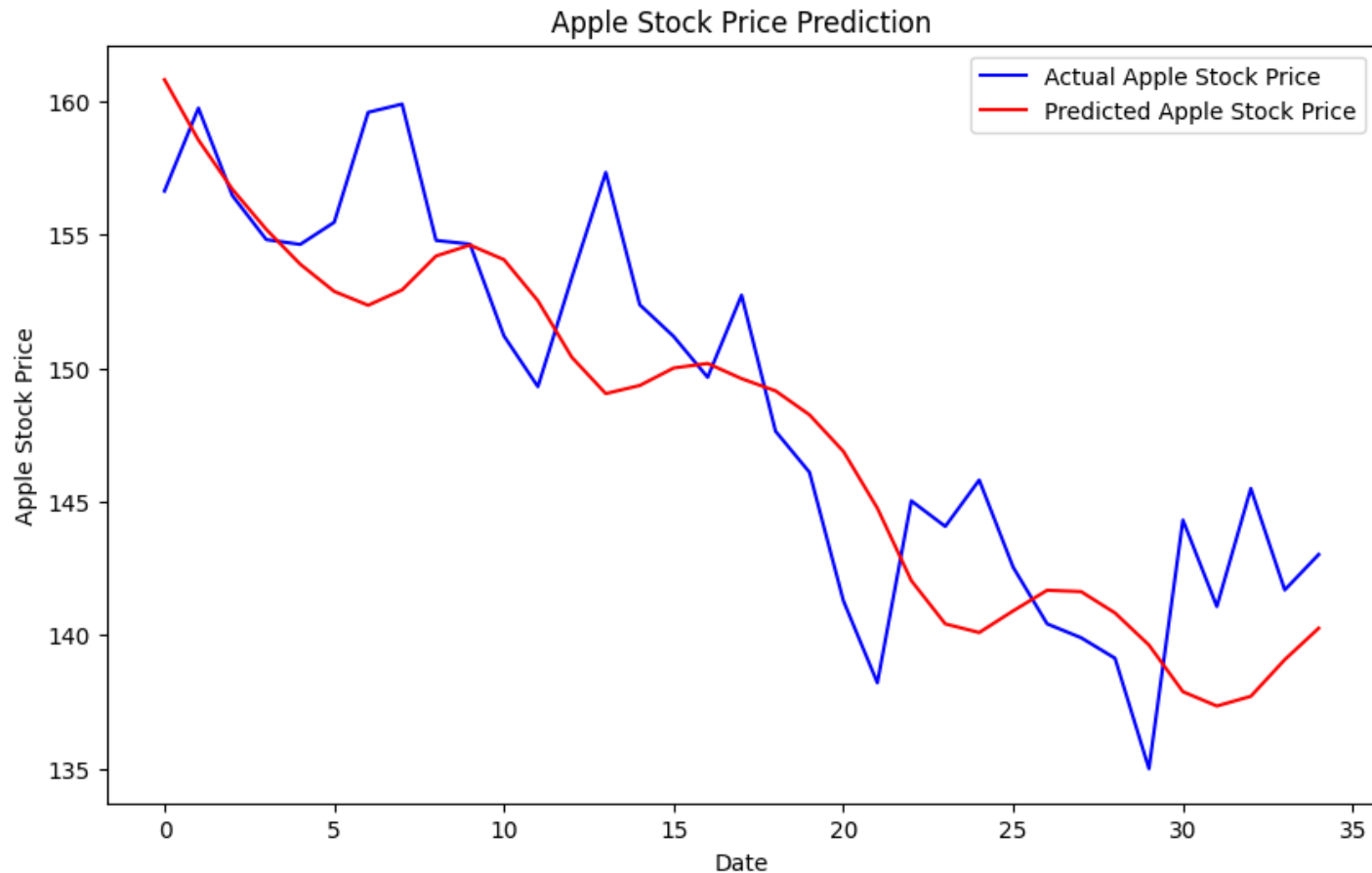
```
plt.ylabel('Apple Stock Price')
```

```
plt.legend()
```

```
plt.show()
```

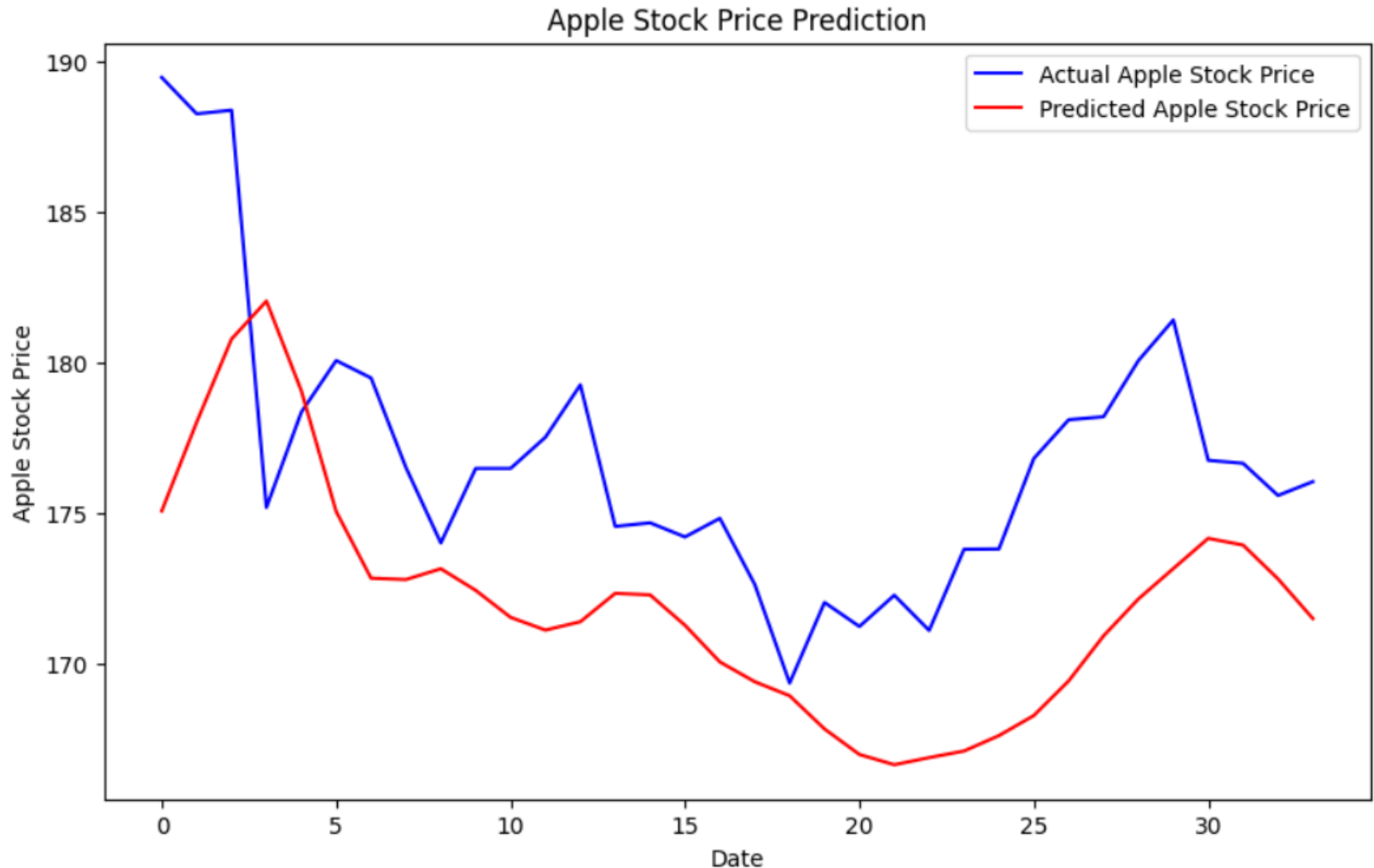
# Visualizing the Prediction, September-October 2022

- This is how our algorithm predicted the future stock prices in September and October 2022.
- Not very impressive. Find some other field to practice your LSTM knowledge.

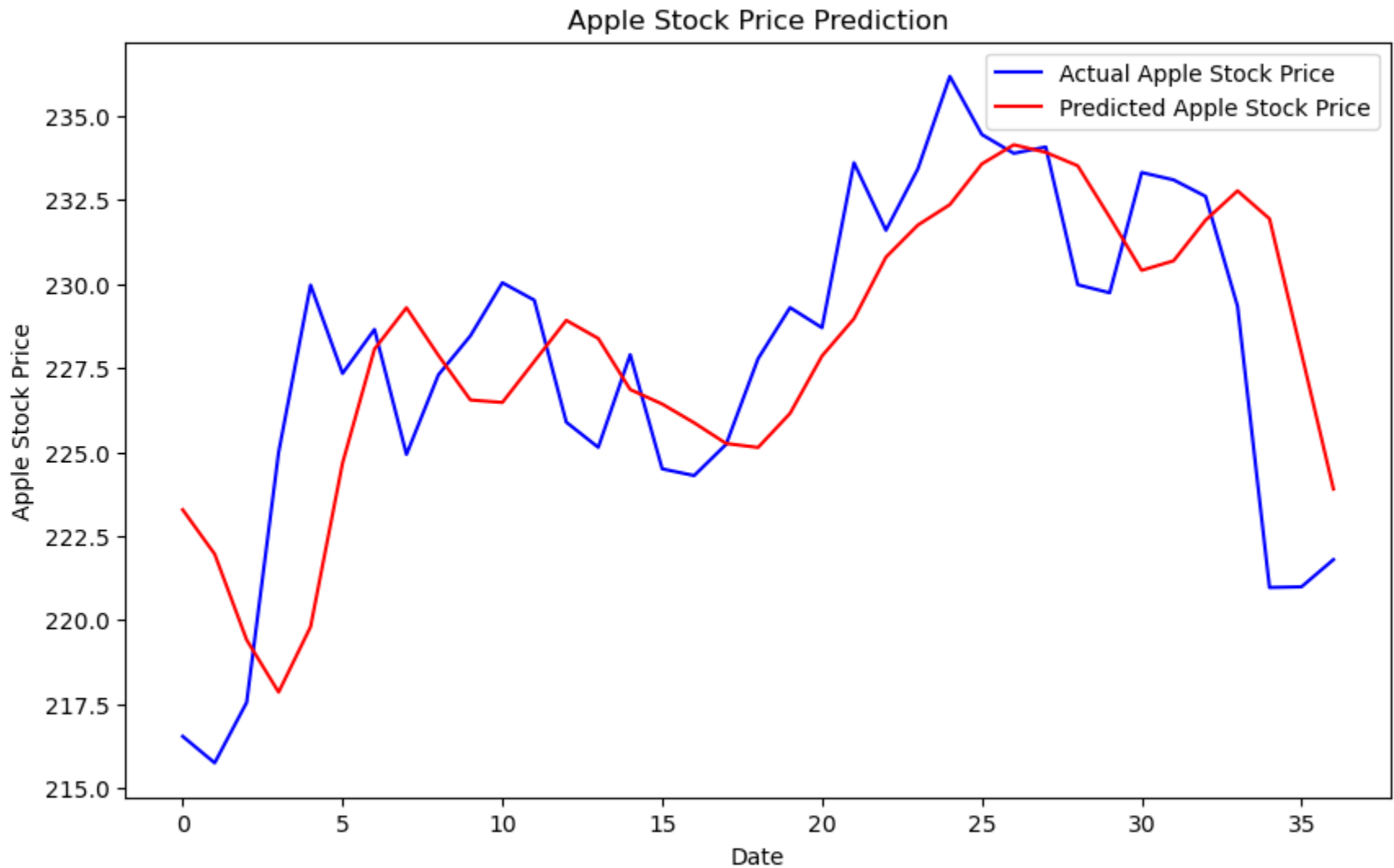


# Visualizing the Prediction, September-October 2023

- Many experts will tell you that this is misleading:



# Prediction September –October 2024



```
tf.keras.utils.timeseries_dataset_from_array
```

## `tf.keras.utils.timeseries_dataset_from_array`

- `tf.keras.utils.timeseries_dataset_from_array` is a utility function which creates a dataset of sliding windows over a time-series provided as array.
- This function appears to be migrated from the package `tf.keras.preprocessing`  

```
tf.keras.utils.timeseries_dataset_from_array(  
    data, targets, sequence_length, sequence_stride=1, sampling_rate=1,  
    batch_size=128, shuffle=False, seed=None, start_index=None, end_index=None  
)
```
- This function takes in a sequence of data-points gathered at equal intervals, along with time series parameters such as length of the sequences/windows, spacing between two sequence/windows, etc., to produce batches of time-series inputs and targets (labels).

# `tf.keras.utils.timeseries_dataset_from_array`

## Arguments

- **data** Numpy array or eager tensor containing consecutive data points (timesteps). Axis 0 is expected to be the time dimension.
- **targets** Targets (labels) corresponding to timesteps in data. `targets[i]` should be the target corresponding to the window that starts at index `i`. Pass `None` if you don't have target data (in this case the dataset will only yield the input data).
- **sequence\_length** Length of the output sequences (in number of timesteps).
- **sequence\_stride** Period between successive output sequences. For stride `s`, output samples would start at index `data[i]`, `data[i + s]`, `data[i + 2 * s]`, etc.
- **sampling\_rate** Period between successive individual timesteps within sequences. For rate `r`, timesteps `data[i]`, `data[i + r]`, ... `data[i + sequence_length]` are used for create a sample sequence.
- **batch\_size** Number of timeseries samples in each batch (except maybe the last one).
- **shuffle** Whether to shuffle output samples, or instead draw them in chronological order.
- **seed** Optional int; random seed for shuffling.
- **start\_index** Optional int; data points earlier (exclusive) than `start_index` will not be used in the output sequences. This is useful to reserve part of the data for test or validation.
- **end\_index** Optional int; data points later (exclusive) than `end_index` will not be used in the output sequences. This is useful to reserve part of the data for test or validation.

## Returns

- A `tf.data.Dataset` instance. If `targets` was passed, the dataset yields tuple `(batch_of_sequences, batch_of_targets)`. If not, the dataset yields only `batch_of_sequences`.

## Examples of use of `timeseries_dataset_from_array`

- Consider an initial sequence of integers `a = [0, 1, ..., 6]`.
- Argument `data` needs to be give the range of first positions of the first element of the input sequence.
- Argument `targets` needs to be give the range of first positions of the first element of the output sequence.

```
ini_sequence = np.arange(10)
dataset = keras.utils.timeseries_dataset_from_array(
    data= ini_sequence[:3],
    targets=ini_sequence[3:],
    sequence_length=3,
    batch_size=2
)
for inputs, targets in dataset:
    for i in range (inputs.shape[0]):
        print([int(x) for x in inputs[i]], int(targets[i]))
```

[0, 1, 2] 3  
[1, 2, 3] 4  
[2, 3, 4] 5  
[3, 4, 5] 6  
[4, 5, 6] 7



# Issues with RNNs

# Resolving Difficulties with classical RNNs

- The biggest issue with RNN are long training times needed for training on long sequences. To capture long sequences, we need to run RNNs over many time steps, making the unrolled RNN a very deep network. Just like any deep neural network it may suffer from the vanishing/exploding gradients problem and might take forever to train.
- Many techniques could alleviate this problem: good parameter initialization, non-saturating activation functions (e.g., ReLU), Batch Normalization, Gradient Clipping, etc. However, for moderately long sequences (e.g., 100 inputs), RNN training becomes very slow.
- The simplest and most common solution to this problem is to unroll the RNN only over a limited number of time steps during training. This is called *truncated backpropagation through time*. In TensorFlow and Keras you can implement *truncated backpropagation through time* simply by truncating the input sequences. For example, in the time series prediction problem, you simply reduce  $n_{steps}$ , the number of steps during training.
- However, the truncated model will not be able to learn long-term patterns. One workaround could be to make sure that shortened sequences contain both old and recent data, so that the model can learn to use both (e.g., the sequence could contain monthly data for the last five months, then weekly data for the last five weeks, then daily data over the last five days).
- This workaround has its limits. Sometime fine-grained data from last year is actually useful? There could be a brief but significant event that absolutely must be taken into account, even years later (e.g., the result of an election)?

# Resolving Difficulties with RNNs, continued

- Besides the long training time, a second major problem faced by deep classical RNNs is the fact that the memory of the first inputs gradually fades away. Indeed, due to the transformations that the data goes through when traversing an RNN, some information is lost after each time step. After a while, the RNN's state contains virtually no trace of the first inputs.
- This can be a showstopper. For example, you want to perform sentiment analysis on a long review that starts with the four words "I loved this movie," but the rest of the review lists the many things that could have made the movie even better. RNN gradually forgets the first four words and will completely misinterpret the review.
- To solve this problem, various types of cells with long-term memory have been introduced. They have proved so successful that the basic RNN cells are rarely used anymore.
- The most popular of these long memory cells is the LSTM cell.

# Neural Machine Translation

# References

This lecture follows material presented in:

- Chapter 16, Hands on Machine Learning with Scikit-learn & TensorFlow, 3<sup>rd</sup> Edition, by Aurélien Géron, O'Reilly 2022
- Neural Machine Translation Tutorial on TensorFlow.org site  
<https://www.tensorflow.org/tutorials/text/transformer>
- Chapter 11, “Deep Learning with Python” 2<sup>nd</sup> Edition by Francois Chollet, 2021, Manning Publishing,
- Chapter 10, Transformers and Pretrained Language Models, “Speech and Language Processing”. Daniel Jurafsky & James H. Martin. Copyright © 2023. All rights reserved. Draft of January 7, 2023.

# Objectives

In this lecture we will introduce:

- ***Encoder-decoder models for machine translation***
- ***Attention***
- In our discussion of RNN-s we dealt with prediction of the single most likely next word in a sentence given the past few words.
- However, there's a whole class of NLP tasks that require outputs that are sequences of varying length. For example,
  - **Translation:** take a sentence in one language as input and output the sentence of the same meaning in another language.
  - **Conversation:** take a statement or question as input and respond to it.
  - **Summarization:** take a large body of text as input and output a summary of that text.

# Encoder-Decoder for Neural Machine Translation

# Neural Machine Translation (NMT)

- By 2015, deep neural network models achieved state-of-the-art results better than the Statistical Language Models which relied on complex statistical processing and grammar based rules. Google and Facebook called those the *Neural Machine Translation* (NMT) systems.
- Neural machine translation identified a single model rather than a pipeline of fine-tuned models and achieved the best overall results.
- The key benefit of NMT approach is that a single system can be trained directly on source and target text, no longer requiring the pipeline of specialized systems used in statistical machine learning.
- Unlike the traditional phrase-based translation system which consists of many small sub-components that are tuned separately, neural machine translation attempts to build and train a single, large neural network that reads a sentence and outputs a correct translation.
- As such, neural machine translation systems are said to be end-to-end systems as only one model is required for the translation.
- The strength of NMT lies in its ability to learn directly, in an end-to-end fashion, the mapping from input text to associated output text.

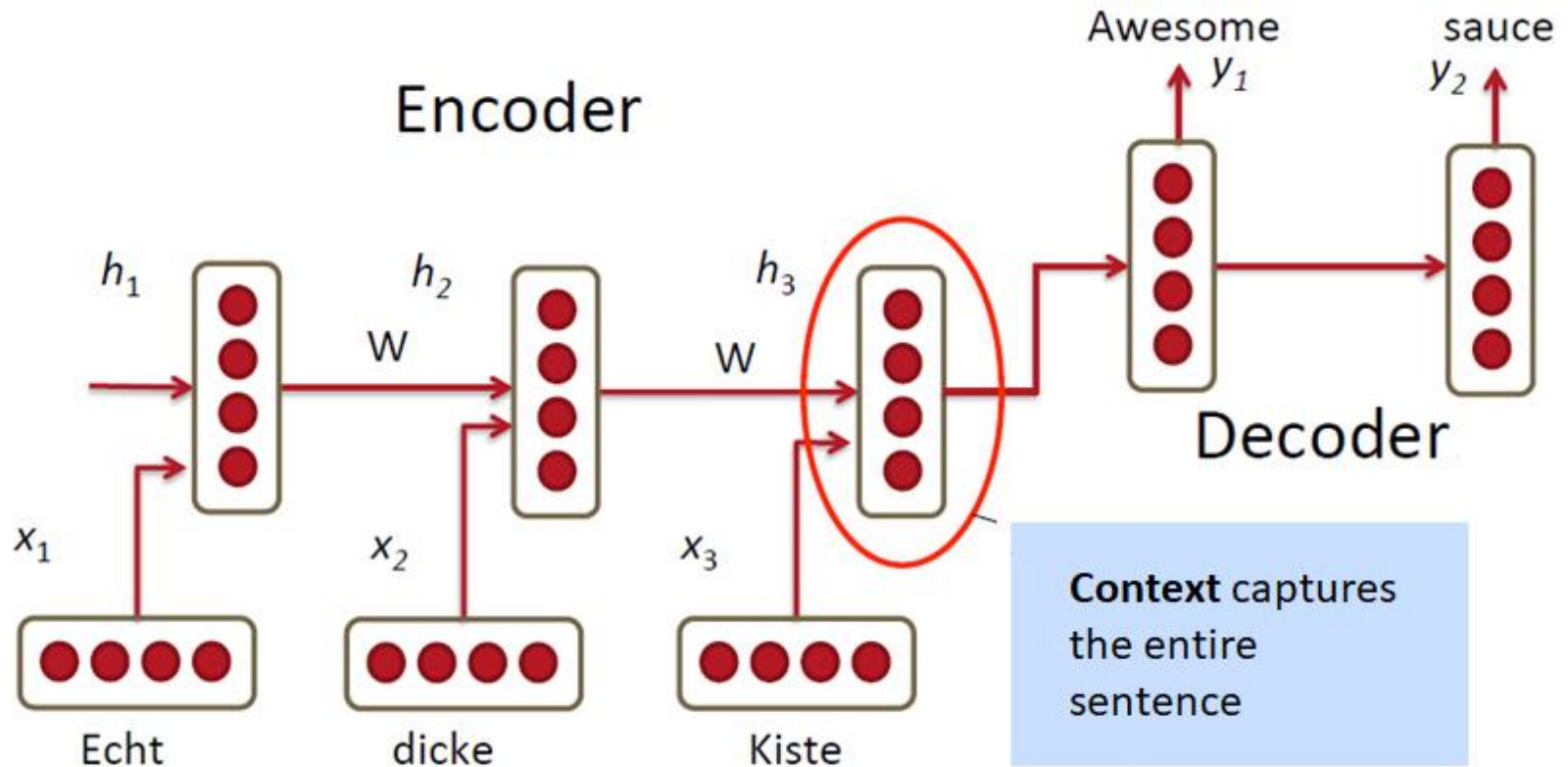


# Encoder-Decoder Models

- Early neural network models for language translation were greatly improved upon introduction of Recurrent Neural Networks (RNNs) organized into encoder-decoder architecture that allows for variable length input and output sequences.
- *An encoder neural network reads and encodes a source sentence into a fixed-length representation referred to as **the context vector (tensor)**.*
- *A decoder network reads the **context vector** and outputs a translation in another language.*
- The whole encoder-decoder system, which consists of the encoder and the decoder for a language pair, is jointly trained to maximize the probability of a correct translation given a source sentence. Weights of two sub-models, encoder and decoder are different but are usually trained together.

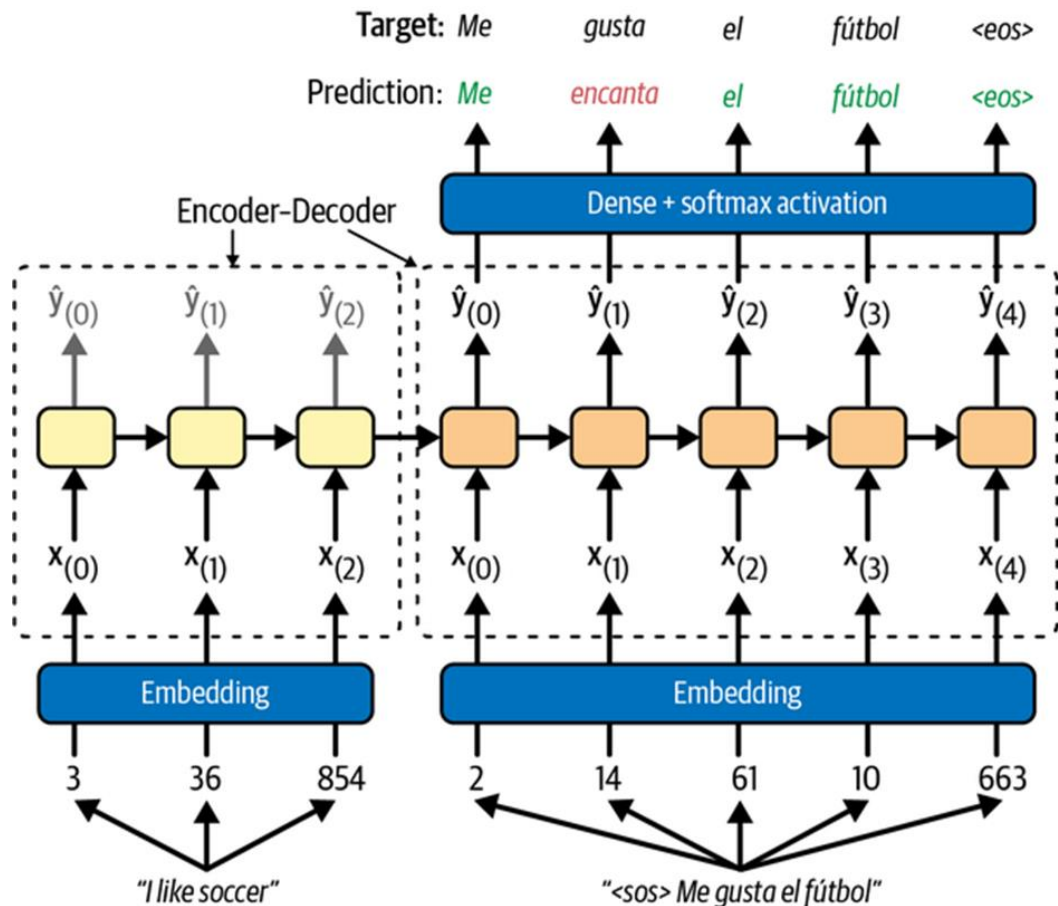
# Encode -> Context -> Decoder

- Context vector (tensor) plays the role of the latent space vector in classical Autoencoders.
- Once the original text is encoded, different decoding systems could be used to translate the context into different languages.*



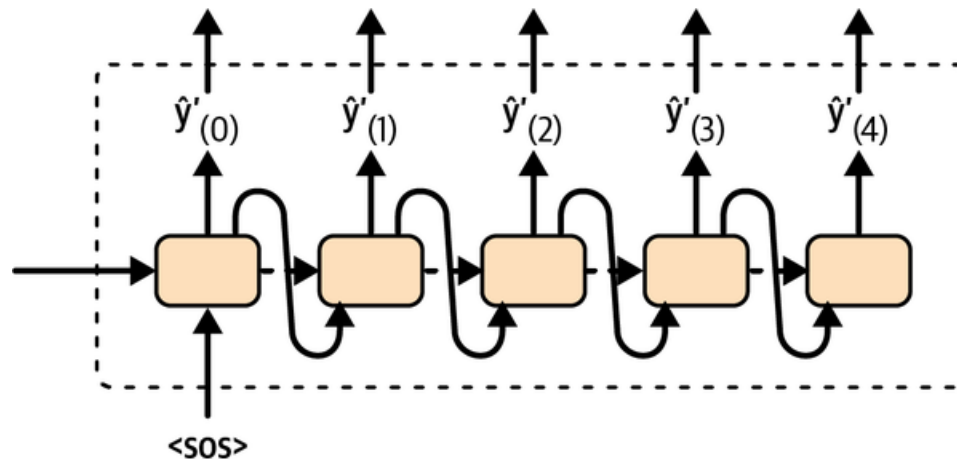
# Simple Encoder-Decoder Model

- We begin with a simple encoded-decoder that translates English sentences to Spanish (see Figure below). English sentences are fed as inputs to the encoder, and the decoder outputs the Spanish translations. Spanish translations are also used as inputs to the decoder during training but shifted back by one step.
- During training, the decoder is given as input the word that it *should* have produced at the previous step, regardless of what it actually produced. This is called *teacher forcing*—a technique that significantly speeds up training and improves the model's performance.
- For the very first word, the decoder is given the start-of-sequence (SOS) token, and the decoder is expected to end the sentence with an end-of-sequence (EOS) token.



# Inference Processing

- Note that at inference time (after training), we do not have the target sentence to feed to the decoder. Instead, we need to feed the decoder the word that it has just output at the previous step. The first “word” in that case is always SOS (Start of Sentence). Sentences fed to the decoder at the inference time also require an embedding transformation.



# <http://www.manythings.org/anki/>

- This is the site where our data are coming from. The site offers data for many language pairs. Some of the datasets are large and useful. Others are not. You can go directly to the site and download any of stored language pairs.




















ManyThings.org

Reading ▾ Sentences ▾ More ▾

## Tab-delimited Bilingual Sentence Pairs

These are selected sentence pairs from the [Tatoeba Project](#).

Updated: 2023-07-30

-  Afrikaans - English [afr-eng.zip](#) (915)
-  Albanian - English [sqi-eng.zip](#) (449)
-  Algerian Arabic - English [arq-eng.zip](#) (155)
-  Arabic - English [ara-eng.zip](#) (12413)
-  Assamese - English [asm-eng.zip](#) (3304)
-  Azerbaijani - English [aze-eng.zip](#) (2195)
-  Basque - English [eus-eng.zip](#) (683)
-  Belarusian - English [bel-eng.zip](#) (3866)
-  Bengali - English [ben-eng.zip](#) (5936)
-  Berber - English [ber-eng.zip](#) (152151)
-  Bosnian - English [bos-eng.zip](#) (151)
-  Breton - English [bre-eng.zip](#) (197)
-  Bulgarian - English [bul-eng.zip](#) (15173)
-  Burmese - English [mya-eng.zip](#) (453)
-  Cantonese - English [yue-eng.zip](#) (3295)
-  Catalan - English [cat-eng.zip](#) (1342)
-  Cebuano - English [ceb-eng.zip](#) (207)
-  Central Dusun - English [dtp-eng.zip](#) (1507)
-  Chavacano - English [cbk-eng.zip](#) (1900)

### Introducing Anki

- If you don't already use Anki, visit the website at <http://ankisrs.net/> to download this free application for Macintosh, Windows or Linux.

### About These Files

- Any flashcard program that can import tab-delimited text files, such as [Anki](#) (free) can use these files.
- Warning!** There are errors in the Tatoeba Corpus. ([Detailed Warning](#))
- In order to minimize the number of errors**, I only used sentences that were owned by [identified native speakers working on the Tatoeba Project](#) and English sentences that I've personally checked and did not reject.
- Warning!** Please remember that even doing this may not have eliminated all errors.

### How the Data Looks

English + TAB + The Other Language + TAB + Attribution

This work isn't easy.	この仕事は簡単じゃない。	CC-BY 2.0 (France) Attribution
Those are sunflowers.	それはひまわりです。	CC-BY 2.0 (France) Attribution
Tom bought a new car.	トムは新車を買った。	CC-BY 2.0 (France) Attribution
This watch is broken.	この時計は壊れている。	CC-BY 2.0 (France) Attribution

The attribution gets imported into Anki as a tag, by default This attribution contains the domain name of the source material, the sentences' ID numbers and the sentence

# Prepare the Environment

- The following are standard steps enabling work with Keras

```
import sys
import tensorflow as tf
import matplotlib.pyplot as plt
plt.rc('font', size=14)
plt.rc('axes', labelsizes=14, titlesize=14)
plt.rc('legend', fontsize=14)
plt.rc('xtick', labelsizes=10)
plt.rc('ytick', labelsizes=10)
```

```
import sklearn
from tensorflow import keras
```

```
# Common imports
import numpy as np
import os
```

```
# to make the output stable across runs
np.random.seed(42)
tf.random.set_seed(42)
```

```
# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsizes=14)
mpl.rc('xtick', labelsizes=12)
mpl.rc('ytick', labelsizes=12)
```

# Fetching Data

- Our data are collections of pairs of sentences in two languages. Here we work with English-Spanish pairs. To fetch the data, we use `tf.keras.utils.get_file()`

```
url = "https://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip"
path = tf.keras.utils.get_file("spa-eng.zip", origin=url,
                               cache_dir="datasets", extract=True)
```

- This results in a directory `spa-eng` which contains a plain text file `spa.txt` with some 120,000 sentence pairs ordered from the shortest to the longest.

```
text = (Path(path).with_name("spa-eng") / "spa.txt").read_text(encoding='utf8')
```

- We start by removing the Spanish characters “ı” and “¿”, which the `TextVectorization` layer does not handle, then we parse and shuffle the sentence pairs. Finally, we split them into two separate lists, one per language:

```
import numpy as np
text = text.replace("ı", "").replace("¿", "")
pairs = [line.split("\t") for line in text.splitlines()]
np.random.seed(42) # extra code - ensures reproducibility on CPU
np.random.shuffle(pairs)
sentences_en, sentences_es = zip(*pairs) # separates the pairs into 2 lists
```

- The first three sentence pairs are:

```
for i in range(3):
    print(sentences_en[i], "=>", sentences_es[i])
```

How boring! => Qué aburrimiento!

I love sports. => Adoro el deporte.

Would you like to swap jobs? => Te gustaría que intercambiamos los trabajos?

# TextVectorization Layers

- Next, we create two `TextVectorization` layers—one per language—and adapt them to the text:

```
vocab_size = 1000
max_length = 50
text_vec_layer_en = tf.keras.layers.TextVectorization(
    vocab_size, output_sequence_length=max_length)
text_vec_layer_es = tf.keras.layers.TextVectorization(
    vocab_size, output_sequence_length=max_length)
text_vec_layer_en.adapt(sentences_en)
text_vec_layer_es.adapt([f"startofseq {s} endofseq" for s in sentences_es])
```

- `adapt()` methods of `TextVectorization` layers create respective vocabularies.
- We limit the vocabulary size to 1,000, which is quite small. That's because the training set is not very large, and because using a small value will speed up training. State-of-the-art translation models typically use a much larger vocabulary (e.g., 30,000), a much larger training set (gigabytes), and a much larger model (hundreds or even thousands of megabytes).
- For similar reasons we limit the sentences in the dataset to a maximum of 50 words, we set `output_sequence_length=50`. The input sequences will automatically be padded with zeros until they are all 50 tokens long. If there was any sentence longer than 50 tokens in the training set, it would be cropped to 50 tokens.
- For the Spanish (target) text, we add “startofseq” and “endofseq” to each sentence when adapting the `TextVectorization` layer. We use these words as SOS and EOS tokens. You could use any other words, as long as they are not actual Spanish words.



# Vocabularies

- Let's inspect the first 10 tokens in both vocabularies.
- They start with the padding token, the unknown token, the SOS and EOS tokens (only in the Spanish vocabulary), then the actual words, sorted by decreasing frequency:

```
text_vec_layer_en.get_vocabulary()[:10]
```

```
['', '[UNK]', 'the', 'i', 'to', 'you', 'tom', 'a', 'is', 'he']
```

```
text_vec_layer_es.get_vocabulary()[:10]
```

```
['', '[UNK]', 'startofseq', 'endofseq', 'de', 'que', 'a', 'no', 'tom', 'la']
```

# Training Set and Validation Set

- Next, let's create the training set and the validation set (you could also create a test set if you needed it).
- We will use the first 100,000 sentence pairs for training, and the rest for validation.
- The decoder's inputs are the Spanish sentences plus an SOS token prefix. The targets are the Spanish sentences plus an EOS suffix:

```
X_train = tf.constant(sentences_en[:100_000])
X_valid = tf.constant(sentences_en[100_000:])
X_train_dec = tf.constant([f"startofseq {s}" for s in
sentences_es[:100_000]])
X_valid_dec = tf.constant([f"startofseq {s}" for s in
sentences_es[100_000:]])
Y_train = text_vec_layer_es([f"{s} endofseq" for s in
sentences_es[:100_000]])
Y_valid = text_vec_layer_es([f"{s} endofseq" for s in
sentences_es[100_000:]])
```

# Model, Inputs

- We are now ready to build our translation model. We will use the functional API for that since the model is not sequential. It requires two text inputs—one for the encoder and one for the decoder.

```
tf.random.set_seed(42) # extra code - ensures reproducibility on CPU
encoder_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
decoder_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
```

- Next, we need to encode these sentences using the `TextVectorization` layers we prepared earlier, followed by an `Embedding` layer for each language, with `mask_zero=True` to ensure masking is handled automatically. The embedding size is a hyperparameter you can tune:

```
embed_size = 128
encoder_input_ids = text_vec_layer_en(encoder_inputs)
decoder_input_ids = text_vec_layer_es(decoder_inputs)
encoder_embedding_layer = tf.keras.layers.Embedding(vocab_size, embed_size,
                                                    mask_zero=True)
decoder_embedding_layer = tf.keras.layers.Embedding(vocab_size, embed_size,
                                                    mask_zero=True)

encoder_embeddings = encoder_embedding_layer(encoder_input_ids)
decoder_embeddings = decoder_embedding_layer(decoder_input_ids)
```

- The embedding matrix is equivalent to one-hot encoding followed by a linear layer with no bias term and no activation function that maps the one-hot vectors to the embedding space. The output layer does the reverse. So, if the model can find an embedding matrix whose transpose is close to its inverse (such a matrix is called an orthogonal matrix), then there's no need to learn a separate set of weights for the output layer.

# Encoder & Decoder

- Now let's create the encoder and pass to it the embedded inputs:

```
encoder = tf.keras.layers.LSTM(512, return_state=True)
encoder_outputs, *encoder_state = encoder(encoder_embeddings)
```

- **Note:** \* collects all the positional arguments in a tuple.
- We use a single LSTM layer but could stack several of them. We set `return_state=True` to get a reference to the layer's final state. Since we're using an LSTM layer, there are actually two states: the short-term state and the long-term state. The layer returns these states separately, which is why we had to write `*encoder_state` to group both states in a list. Now we can use this (double) state as the initial state of the decoder:

- The decoder is defined as:

```
decoder = tf.keras.layers.LSTM(512, return_sequences=True)
decoder_outputs = decoder(decoder_embeddings, initial_state=encoder_state)
```

- The decoder's outputs are passed through a Dense layer with the softmax activation function to get the word probabilities for each step:

```
output_layer = tf.keras.layers.Dense(vocab_size, activation="softmax")
Y_proba = output_layer(decoder_outputs)
```

- The model has two inputs and one output:

```
model = tf.keras.Model(inputs=[encoder_inputs, decoder_inputs],
                       outputs=[Y_proba])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
```

# Train the model

- This training should be run on a machine with a GPU card or in Google Colab.
- On an Nvidia 1080 Ti card, the speed is as following:

```
model.fit((X_train, X_train_dec), Y_train, epochs=10,
          validation_data=((X_valid, X_valid_dec), Y_valid))
Epoch 1/10
3125/3125 [=====] - 73s 20ms/step - loss: 0.4153 - accuracy: 0.4252 - val_loss:
0.3114 - val_accuracy: 0.5209
Epoch 2/10
3125/3125 [=====] - 60s 19ms/step - loss: 0.2687 - accuracy: 0.5690 - val_loss:
0.2416 - val_accuracy: 0.6008
Epoch 3/10
3125/3125 [=====] - 60s 19ms/step - loss: 0.2116 - accuracy: 0.6387 - val_loss:
0.2091 - val_accuracy: 0.6454
Epoch 4/10
3125/3125 [=====] - 60s 19ms/step - loss: 0.1777 - accuracy: 0.6852 - val_loss:
0.1939 - val_accuracy: 0.6667
Epoch 5/10
3125/3125 [=====] - 60s 19ms/step - loss: 0.1538 - accuracy: 0.7193 - val_loss:
0.1874 - val_accuracy: 0.6763
Epoch 6/10
3125/3125 [=====] - 62s 20ms/step - loss: 0.1345 - accuracy: 0.7481 - val_loss:
0.1858 - val_accuracy: 0.6794
Epoch 7/10
3125/3125 [=====] - 64s 20ms/step - loss: 0.1182 - accuracy: 0.7730 - val_loss:
0.1870 - val_accuracy: 0.6823
Epoch 8/10
3125/3125 [=====] - 61s 20ms/step - loss: 0.1038 - accuracy: 0.7969 - val_loss:
0.1909 - val_accuracy: 0.6826
Epoch 9/10
3125/3125 [=====] - 61s 19ms/step - loss: 0.0912 - accuracy: 0.8176 - val_loss:
0.1958 - val_accuracy: 0.6793
Epoch 10/10
3125/3125 [=====] - 62s 20ms/step - loss: 0.0802 - accuracy: 0.8371 - val_loss:
0.2021 - val_accuracy: 0.6759
<keras.callbacks.History at 0x1716adb0c7
```

# Use trained model for translation

- After training, we can use the model to translate new English sentences to Spanish. But it's not as simple as calling `model.predict()`, because the decoder expects as input the word that was predicted at the previous time step. One way to do this is to write a custom memory cell that keeps track of the previous output and feeds it to the encoder at the next time step. However, to keep things simple, we can just call the model multiple times, predicting one extra word at each round. The following utility function does that:

```
def translate(sentence_en):  
    translation = ""  
    for word_idx in range(max_length):  
        X = np.array([sentence_en]) # encoder input  
        X_dec = np.array(["startofseq " + translation]) # decoder input  
        y_proba = model.predict((X,X_dec))[0,word_idx] # last token's probas  
        predicted_word_id = np.argmax(y_proba)  
        predicted_word =  
            text_vec_layer_es.get_vocabulary()[predicted_word_id]  
        if predicted_word == "endofseq":  
            break  
        translation += " " + predicted_word  
    return translation.strip()
```

- The function simply keeps predicting one word at a time, gradually completing the translation, and it stops once it reaches the EOS token

# Translation

```
>>> translate("I like soccer")  
'me gusta el fútbol'
```

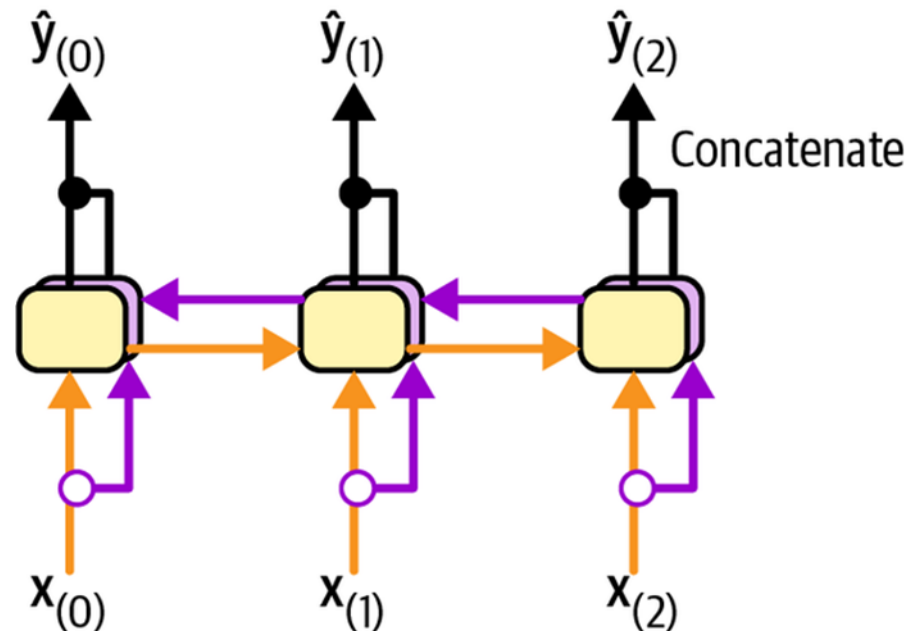
- Model works with very short sentences. If you try playing with this model for a while, you will find that it struggles with longer sentences. For example:

```
>>> translate("I like soccer and also going to the beach")  
1/1 [=====] - 0s 20ms/step  
1/1 [=====] - 0s 19ms/step  
1/1 [=====] - 0s 21ms/step  
1/1 [=====] - 0s 21ms/step  
1/1 [=====] - 0s 21ms/step  
1/1 [=====] - 0s 21ms/step  
1/1 [=====] - 0s 20ms/step  
1/1 [=====] - 0s 21ms/step  
'me gustan las [UNK] y al tenis'
```

- How can you improve it?
- One way is to increase the training set size and add more LSTM layers in both the encoder and the decoder. But this will only get you so far, so let's look at more sophisticated techniques, starting with bidirectional recurrent layers.

# Bidirectional RNNs

- At each time step, a regular recurrent layer only looks at past and present inputs before generating its output. In other words, it is causal, meaning it cannot look into the future. This type of RNN makes sense when forecasting time series, or in the decoder of a sequence-to-sequence (seq2seq) model. But for tasks like text classification, or in the encoder of a seq2seq model, it is often preferable to look ahead at the next words before encoding a given word.
- For example, consider the phrases “the right arm”, “the right person”, and “the right to criticize”: to properly encode the word “right”, you need to look ahead. One solution is to run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left, then combine their outputs at each time step, typically by concatenating them. This is what a bidirectional recurrent layer does.





# Implementation of Bidirectional RNN

- To implement a bidirectional recurrent layer in Keras, just wrap a recurrent layer in a `tf.keras.layers.Bidirectional` layer. For example, the following Bidirectional layer could be used as the encoder in our translation model:

```
encoder = tf.keras.layers.Bidirectional(  
    tf.keras.layers.LSTM(256, return_state=True))
```

- The Bidirectional layer will create a clone of the LSTM layer (but in the reverse direction), and it will run both layers and concatenate their outputs. Although the LSTM layer has 10 units, the Bidirectional layer will output 20 values per time step.
- There is one problem. This layer will now return four states instead of two: the final short-term and long-term states of the forward LSTM layer, and the final short-term and long-term states of the backward LSTM layer. We cannot use this quadruple state directly as the initial state of the decoder's LSTM layer, since it expects just two states (short-term and long-term).
- We can concatenate the two short-term states, and concatenate the two long-term states:

```
encoder_outputs, *encoder_state = encoder(encoder_embeddings)  
encoder_state = [tf.concat(encoder_state[:, :2], axis=-1), # short-term (0 & 2)  
                 tf.concat(encoder_state[1: :2], axis=-1)] # long-term (1 & 3)
```

# Bidirectional Model

- We cannot make the decoder bidirectional, since it must remain causal. Otherwise, it would cheat during training, and it would not work
- A completed bidirectional LSTM model would read:

```
decoder = tf.keras.layers.LSTM(512, return_sequences=True)
decoder_outputs = decoder(decoder_embeddings, initial_state=encoder_state)

output_layer = tf.keras.layers.Dense(vocab_size, activation="softmax")
Y_proba = output_layer(decoder_outputs)

model = tf.keras.Model(inputs=[encoder_inputs, decoder_inputs],
                        outputs=[Y_proba])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
```

- This bidirectional model might train a bit longer than the previous model.

# Training of Bidirectional Model

```
model.fit((X_train, X_train_dec), Y_train, epochs=10,
          validation_data=((X_valid, X_valid_dec), Y_valid))
Epoch 1/10
3125/3125 [=====] - 90s 26ms/step - loss: 0.3115 - accuracy:
0.5363 - val_loss: 0.2218 - val_accuracy: 0.6281
Epoch 2/10
3125/3125 [=====] - 81s 26ms/step - loss: 0.1933 - accuracy:
0.6667 - val_loss: 0.1885 - val_accuracy: 0.6712
. . . . .
Epoch 9/10
3125/3125 [=====] - 82s 26ms/step - loss: 0.0767 - accuracy:
0.8425 - val_loss: 0.1881 - val_accuracy: 0.6942
Epoch 10/10
3125/3125 [=====] - 82s 26ms/step - loss: 0.0687 - accuracy:
0.8563 - val_loss: 0.1940 - val_accuracy: 0.6919
<keras.callbacks.History at 0x173f54b2f50>
```

- **We can use previously defined `translate()` function to test translation capabilities.**

```
translate("I like soccer")
1/1 [=====] - 3s 3s/step
1/1 [=====] - 0s 29ms/step
1/1 [=====] - 0s 24ms/step
1/1 [=====] - 0s 21ms/step
1/1 [=====] - 0s 21ms/step
'me gusta el fútbol'
```

# Method for Assessing Quality of Translations

## BLUE Score

# Bilingual Evaluation Understudy (BLEU) Score

- In 2002, IBM researchers developed the Bilingual Evaluation Understudy (BLEU) that remains, with its many variants, one of the most respected and reliable methods for assessing the quality of machine translation.
  - The BLEU algorithm evaluates the precision score of a candidate machine translation against a reference human translation. The reference human translation is assumed to be a model example of a translation.
  - BLUE method uses n-gram matches as the metric to show how similar a candidate translation is to the human translation.
  - BLUE algorithm labels reference sentences and the corresponding candidate translations
  - The algorithm evaluates the quality of a translation, by comparing the number of n-grams between the candidate translation and the reference.
  - The BLEU algorithm looks for whether n-grams in the machine translation also appear in the reference translation.
  - Color-coded below are some examples of different size n-grams that are shared between the reference (A) and candidate translation (B).
- A. *there are many ways to evaluate **the quality of a translation**, like comparing the number **of n-grams** between a candidate translation and **reference**.*
- B. ***the quality of a translation** evaluates the number **of n-grams** in a **reference** and the translation.*

# BLUE Algorithm

- The BLEU algorithm identifies all such matches of the n-grams, including the unigram matches. The precision score is the fraction of n-grams in the translation that also appear in the reference.
- The algorithm satisfies two other constraints.
  - a) For each n-gram size (1,2,3,4,...), any n-gram in the reference translation cannot be matched more than once. For example, the unigram "a" appears twice in B but only once in A. This only counts for one match between the sentences.
  - b) Additionally, we impose a brevity penalty so that very small sentences that could achieve a 1.0 precision (a "perfect" matching) are not considered as good translations. For example, the single word sentence "There." would achieve a 1.0 precision match. However, we do not include it in the final score.

Reference:

- <https://en.wikipedia.org/wiki/BLEU>

# Computation of BLUE Score

- Let  $k$  denotes the maximum n-gram that we want to evaluate our score on. That is, if  $k = 4$ , the BLUE score only counts the number of n-grams with length less than or equal to 4 and ignores larger n-grams.
- In candidate translation, quantity  $p_n$

$$p_n = \# \text{ matched ngrams} / \# n_{\text{grams}}$$

denotes the precision score for the grams of length  $n$ .

- Finally, let  $w_n = 1/2^n$  be a geometric weighting for the precision of the  $n$ 'th gram.
- The brevity penalty is defined as

$$\beta = e^{\min(0, 1 - \frac{\text{len}_{ref}}{\text{len}_{MT}})}$$

- where  $\text{len}_{ref}$  is the length of the reference translation and  $\text{len}_{MT}$  is the length of the machine translation.
- The BLEU score is then defined as  $BLUE = \beta \prod_{n=1}^k p_n^{w_n}$
- The BLEU score correlates well with human judgment of good translations.
- BLUE score does have many limitations. It only works well on the corpus level because any zeros in precision scores will zero the entire BLEU score. Additionally, this BLEU score compares a candidate translation against a single reference, which results in a noisy representation of relevant n-grams that need to be matched.

# Attention

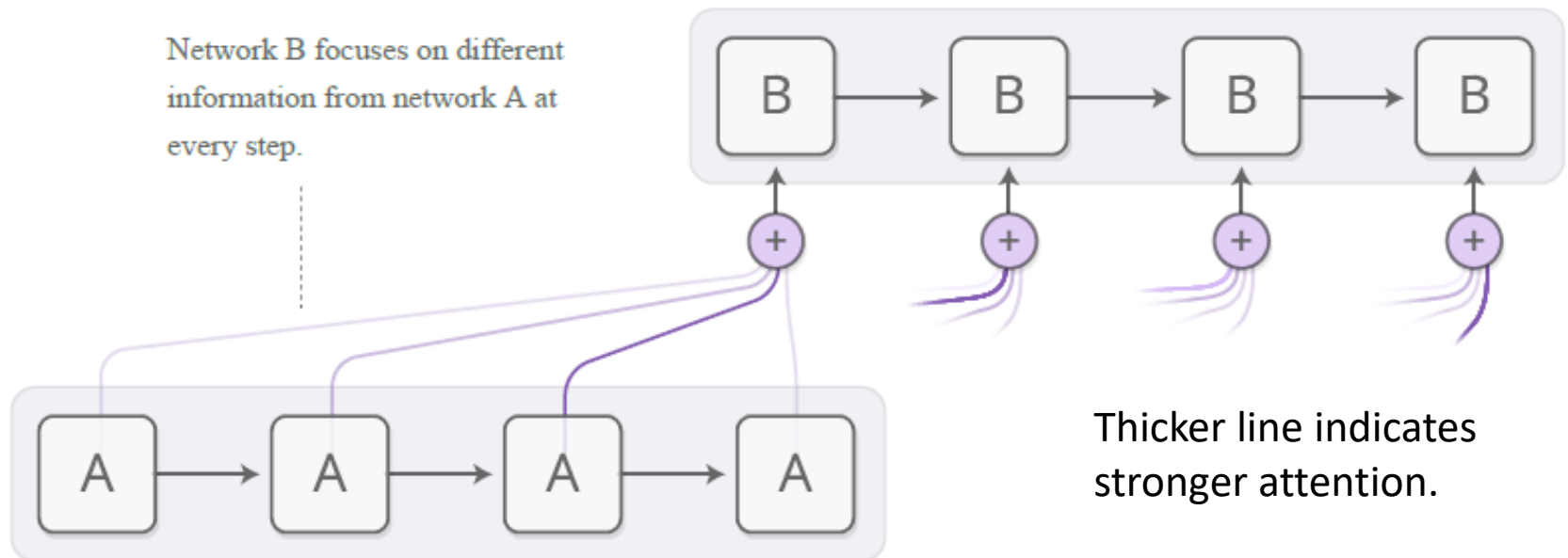


# Attention Mechanism

- All RNNs, including LSTMs have issues with very long texts or sentences.
- The problem of sentence length was addressed by [Dzmitry Bahdanau](#), [Kyunghyun Cho](#), & [Yoshua Bengio](#) in a paper entitled:  
*Neural Machine Translation by Jointly Learning to Align and Translate*, 2015 (<https://arxiv.org/abs/1409.0473>) in which they introduced the attention mechanism.
- *Instead of encoding the input sentence to a fixed length vector, a fuller representation of the encoded input is kept and the decoder learns to pay attention to different parts of the input for each word in the output.*
- *Each time the proposed model generates a word in a translation, it (soft-)searches for a set of positions in a source sentence where the most relevant information is concentrated.*
- The model then predicts a target word based on the context vectors associated with these source positions and all the previous generated target words.

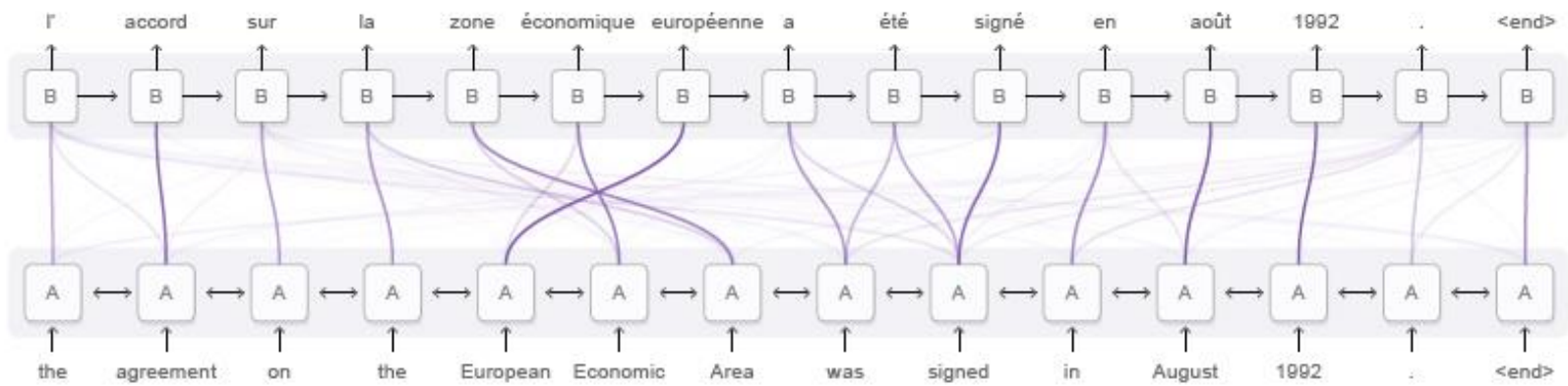
# Attention Mechanism

- When translating a sentence, we pay special attention to the word we are presently translating.
- When transcribing an audio recording, we listen carefully to the segment we are actively writing down.
- If one asks us to describe the room we are sitting in, we glance around at the objects we are describing as we do so.
- Neural networks can behave in the above fashion by using *attention*, i.e., focusing on a subset of the information they are given. For example, an RNN can concentrate over the portion of the output of another RNN. At every time step, it focuses on different positions in the output of the other RNN.



# Attention in Translations

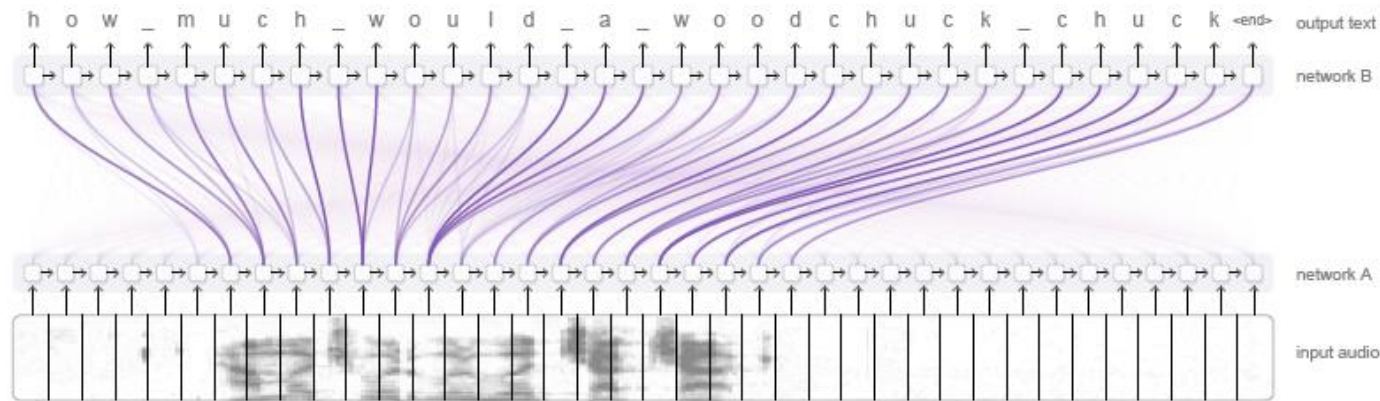
- The main use of attention is translation . A traditional sequence-to-sequence model maps the entire input into a single vector and then expands it back out.
- Attention avoids this by allowing the RNN processing the input to pass along information about each word it sees, and then for the RNN generating the output to focus on words as they become relevant.



- Other uses of this kind of attention is parsing text , where it allows the model to glance at words as it generates the parse tree, and for conversational modeling , where it lets the model focus on previous parts of the conversation as it generates its response.

# Attention in Voice Recognition

- Attention has several other applications. It can be used in voice recognition , allowing one RNN to process the audio and then have another RNN skim over it, focusing on relevant parts as it generates a transcript.



- Jointly Learning to Align and Translate

# An interface between a CNN and an RNN

- Attention can also be used on the interface between a convolutional neural network and an RNN. This allows the RNN to look at different position of an image every step.
- One popular use of this kind of attention is for image captioning. First, a conv net processes the image, extracting high-level features. Then an RNN runs, generating a description of the image. As it generates each word in the description, the RNN focuses on the conv net's interpretation of the relevant parts of the image.



A woman is throwing a frisbee in a park.



A dog is standing on a hardwood floor.



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

# Self Attention

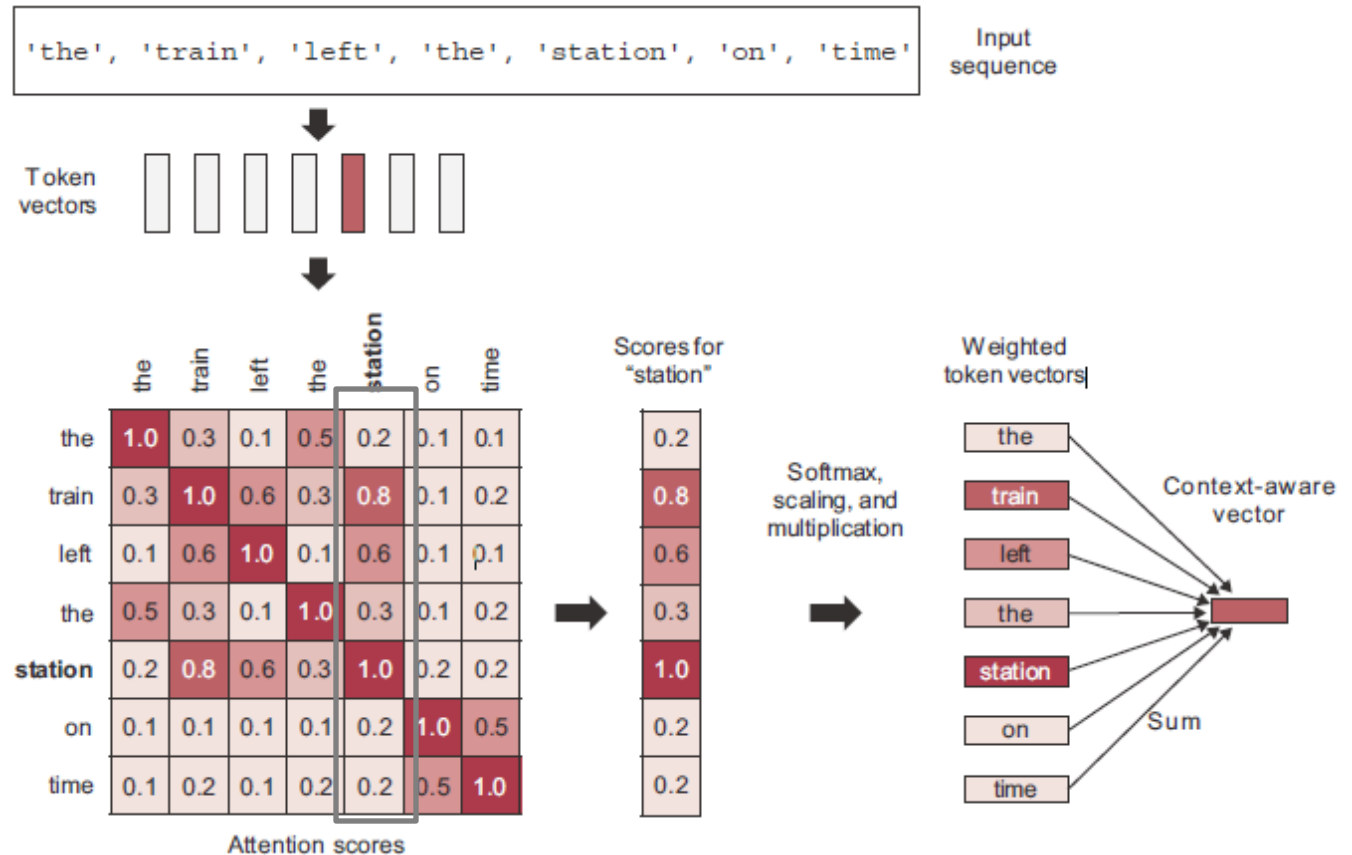
- A serious problem with embedding spaces in word2vec is that words with different meaning (river **bank** vs. financial **bank**) but the same spelling, get mapped into the same embedding vector.
- Smart embedding space would provide a different vector representation for a word depending on the other words surrounding it. That is where *self-attention* comes in.
- The purpose of self-attention is to modulate the representation of a token by using the representations of related tokens in the sequence. Self-attention produces context aware token representations.
- Consider an example sentence: “The train left the station on time.” Now, consider one word in the sentence: station. What kind of station are we talking about? Could it be a radio station? We use attention to modify the embedding representation of word “station”, so it reflects the context around it.

We do it in the following way:

- Step 1 is to compute relevancy scores between the vector for “station” and every other word in the sentence. These are our “attention scores.” We use the dot product between every two word vectors as a measure of the strength of their relationship. This is a very computationally efficient distance function. These scores also go through a scaling function and a softmax.
- Step 2 is to compute the sum of all word vectors in the sentence, weighted by our relevancy scores. Words closely related to “station” will contribute more to the sum (including the word “station” itself), while irrelevant words will contribute almost nothing. The resulting vector is our new representation for “station”: a representation that incorporates the surrounding context. In particular, it includes part of the “train” vector, clarifying that it is, in fact, a “train station.”

# Calculation of Self-Attention

- We repeat this process for every word in the sentence, producing a new sequence of vectors encoding the sentence.




- Self-attention: attention scores are computed between "station" and every other word in the sequence. Those scores for word "station" are scaled by the square root of sentence length and passed through softmax function which generates "probabilities". Those probabilities are weights of all word vectors in the sentence in the sum that becomes the new "station" vector.

# Generalized Attention

- The self-attention mechanism performs the following, schematically:

```
outputs = sum(inputs * pairwise_scores(inputs, inputs))
```



- This means “for each token in `inputs` (A), compute how much that token is related to every token in `inputs` (B) and use these scores as weights in a sum of tokens from `inputs` (C).” A, B, and C do not have to refer to the same input sequence. In the general case, you could be doing this with three different sequences. We call them “query,” “keys,” and “values.” The operation becomes “for each element in the query (A) compute how much the element is related to every key (B) and use these scores to weight a sum of values C”.

```
outputs = sum(values * pairwise_scores(query, keys))
```

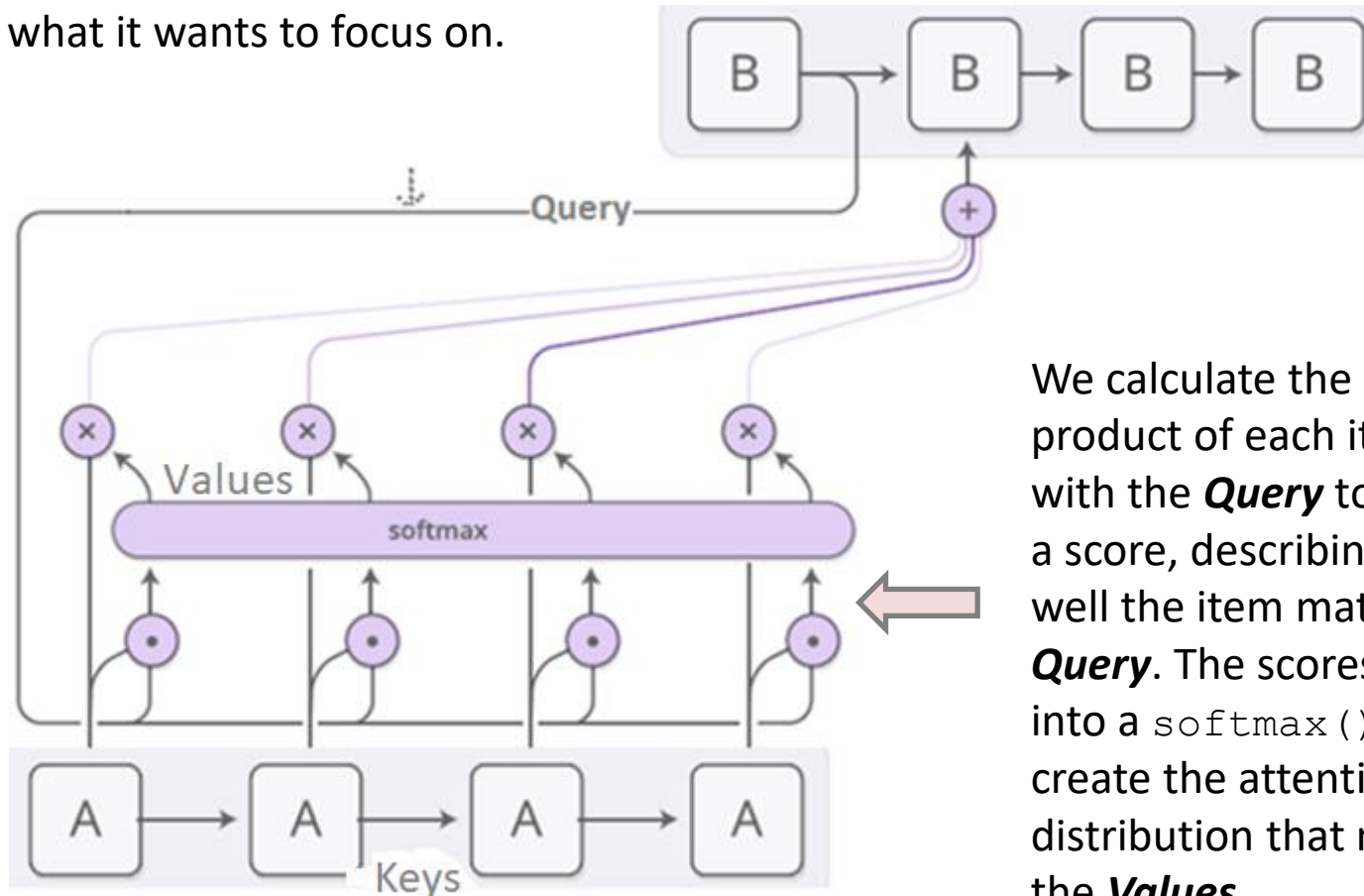
- Conceptually, this is what generalized attention is doing.
- A reference sequence describes something you are looking for: a `query`.
- You are trying to extract information from a body of knowledge: the `values`.
- Each `value` is assigned a `key` that describes the value in a format that can be readily compared to a `query`. You simply match the `query` to the `keys`. Then you return a weighted sum of `values`.
- In practice, the `keys` and the `values` are often the same sequence. In machine translation, for instance, the `query` would be the target sequence, and the source sequence would play the roles of both `keys` and `values`: for each element of the target. When translating to Spanish each element of the target (“tiempo”) will require us to go back to the source (“How is the weather today?”) and identify different tokens that are related. “tiempo” and “weather” should have a strong match.



# Calculation of Generalized Attention

- The distribution of attention is usually generated with content-based attention.

The attending RNN  
generates a *Query* describing  
what it wants to focus on.



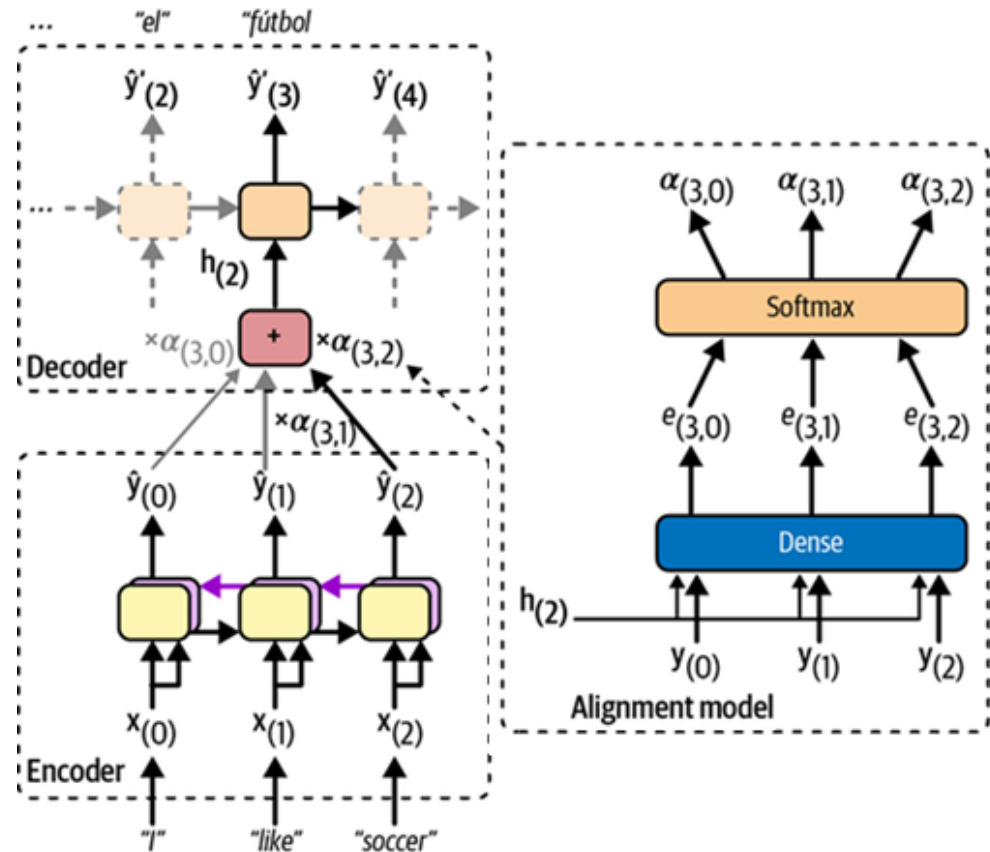
We calculate the dot product of each item (**Key**) with the **Query** to produce a score, describing how well the item matches the **Query**. The scores are fed into a `softmax()` unit to create the attention distribution that modifies the **Values**

# Encoder-Decoder with Attention

# Encoder-Decoder with Attention

- Image below shows encoder-decoder model with an added attention mechanism. On the left, you have the encoder and the decoder. Instead of just sending the encoder's final hidden state to the decoder, as well as the previous target word at each step, we now send all of the encoder's outputs to the decoder as well.

The decoder needs those outputs aggregated. At each time step, the decoder's memory cell computes a weighted sum of all the encoder outputs. This determines which words it will focus on at this step. The weight  $\alpha(t,i)$  is the weight of the  $i$ -th encoder output at the  $t$ -th decoder time step. At time step  $t$ , decoder pays much more attention to the encoder's output for word with the highest  $\alpha(t,i)$ . The rest of the decoder works just like earlier: at each time step it receives the above input plus the hidden state from the previous time step, and finally it receives the target word from the previous time step (or at inference time, the output from the previous time step.)



# Attention Layer

- Weights  $\alpha(t, i)$  are generated by a small network called an *Attention Layer* (or Alignment Model), which is trained jointly with the rest of the model.
- This attention layer is illustrated on the righthand side of the previous slide. It starts with a Dense layer composed of a single neuron that processes each of the encoder's outputs, along with the decoder's previous hidden state (e.g.,  $h(2)$ ). This layer outputs a score (or energy) for each encoder output (e.g.,  $e(3, 2)$ ): this score measures how well each output is aligned with the decoder's previous hidden state.
- For example, the model has already output “me gusta el” (meaning “I like”), so it's now expecting a noun: the word “soccer” is the one that best aligns with the current state, so it gets a high score.
- Finally, all the scores go through a *softmax* layer to get a final weight for each encoder output (e.g.,  $\alpha(3,2)$ ). All the weights for a given decoder time step add up to 1.

# Bahdanau and other Attentions

- There are three popular attention calculations called: Dot, Luong (General) and Bahdanau (Concatenative attention).
- Attentions expressed over the hidden state of the encoder at time  $t$ ,  $h(t)$  and the outputs of the encoder  $y(i)$ , read:

$$\begin{aligned}\tilde{\mathbf{h}}_{(t)} &= \sum_i \alpha_{(t,i)} \mathbf{y}_{(i)} \\ \text{with } \alpha_{(t,i)} &= \frac{\exp(e_{(t,i)})}{\sum_{i'} \exp(e_{(t,i')})} \\ \text{and } e_{(t,i)} &= \begin{cases} \mathbf{h}_{(t)}^\top \mathbf{y}_{(i)} & \text{dot} \\ \mathbf{h}_{(t)}^\top \mathbf{W} \mathbf{y}_{(i)} & \text{general} \\ \mathbf{v}^\top \tanh(\mathbf{W} [\mathbf{h}_{(t)}; \mathbf{y}_{(i)}]) & \text{concat} \end{cases}\end{aligned}$$

- The last expressions for  $e_{(t,i)}$  has three forms. The first is “Dot”, the second is Luong or “General” and the last is Bahdanau or “Concatenative”.
- Dzmitry Bahdanau is the leading author of the first Attention paper.

# Constructing Encoder-Decoder with Attention

- Since we will need to pass all the encoder's outputs to the Attention layer, we first need to set `return_sequences=True` when creating the encoder:

```
tf.random.set_seed(42) # extra code - ensures reproducibility on CPU
encoder = tf.keras.layers.Bidirectional(
    tf.keras.layers.LSTM(256, return_sequences=True, return_state=True))

# this part of the model is exactly the same as earlier
encoder_outputs, *encoder_state = encoder(encoder_embeddings)
encoder_state = [tf.concat(encoder_state[::2], axis=-1), #short-term (0 & 2)
                 tf.concat(encoder_state[1::2], axis=-1)] # long-term(1& 3)
decoder = tf.keras.layers.LSTM(512, return_sequences=True)
decoder_outputs = decoder(decoder_embeddings, initial_state=encoder_state)
```

- Next, we need to create the attention layer and pass to it the decoder's states and the encoder's outputs. However, to access the decoder's states at each step we would need to write a custom memory cell. For simplicity, let's use the decoder's outputs instead of its states. In practice this works well too, and it's much easier to code. Then we just pass the attention layer's outputs directly to the output layer, as suggested in the Luong attention paper:

```
attention_layer = tf.keras.layers.Attention()
attention_outputs = attention_layer([decoder_outputs, encoder_outputs])
output_layer = tf.keras.layers.Dense(vocab_size, activation="softmax")
Y_proba = output_layer(attention_outputs)
```

# Training Encoder-Decoder with Attention

- This model should run on a GPU machine or Google Colab.

```
model = tf.keras.Model(inputs=[encoder_inputs, decoder_inputs], outputs=[Y_proba])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", metrics=["accuracy"])
model.fit((X_train, X_train_dec), Y_train, epochs=10,
          validation_data=((X_valid, X_valid_dec), Y_valid))
```

Epoch 1/10

```
3125/3125 [=====] - 96s 28ms/step - loss: 0.3023 - accuracy:
0.5481 - val_loss: 0.2121 - val_accuracy: 0.6424
```

Epoch 2/10

```
3125/3125 [=====] - 84s 27ms/step - loss: 0.1917 - accuracy:
0.6745 - val_loss: 0.1871 - val_accuracy: 0.6814
```

. . . . .

Epoch 10/10

```
3125/3125 [=====] - 82s 26ms/step - loss: 0.0940 - accuracy:
0.8178 - val_loss: 0.1905 - val_accuracy: 0.7031
```

```
<keras.callbacks.History at 0x174159a2f20>
```

- This model now handles much longer sentences. For example:

```
translate("I like soccer and also going to the beach")
```

```
'me gusta el fútbol y también ir a la playa'
```

```
beam_search("I like soccer and also going to the beach", beam_width=3,
             verbose=True)
```

```
'me gusta el fútbol y también ir a la playa'
```

- In short, the attention layer provides a way to focus the attention of the model on part of the inputs.

## Appendix:

# Quiz-like Questions about RNNs



# Input vs Output Dimensions

- **Q1a:** How many dimensions must the inputs of an RNN layer have? What does each dimension represent?
- **Answer 1a.** An RNN layer must have three-dimensional inputs: the first dimension is the batch dimension (its size is the batch size), the second dimension represents the time (its size is the number of time steps), and the third dimension holds the inputs at each time step (its size is the number of input features per time step).  
For example, if you want to process a batch containing 5 time series of 10 time steps each, with 2 values per time step (e.g., the temperature and the wind speed), the shape will be [5, 10, 2].
- **Q1b:** What about its outputs?
- **Answer 1b:** The **outputs** are also three-dimensional, with the same first two dimensions, but the last dimension is equal to the number of neurons. For example, if an RNN layer with 32 neurons processes the batch the output will have a shape of [5, 10, 32].
- **Q2:** If you want to build a deep sequence-to-sequence RNN, which RNN layers should have `return_sequences=True`?
- What about a sequence-to-vector RNN?
- **Answer 2:** To build a deep sequence-to-sequence RNN using Keras, you must set `return_sequences=True` for all RNN layers. To build a sequence-to-vector RNN, you must set `return_sequences=True` for all RNN layers except for the top RNN layer, which must have `return_sequences=False` (or do not set this argument at all, since `False` is the default).

# RNN Architecture

- **Q3:** Suppose you have a daily univariate time series, and you want to forecast the next seven days. Which RNN architecture should you use?
- **Answer 3** The simplest RNN architecture you can use is a stack of RNN layers (all with `return_sequences=True` except for the top RNN layer), using seven neurons in the output RNN layer.

You can then train this model using random windows from the time series (e.g., sequences of 30 consecutive days as the inputs, and a vector containing the values of the next 7 days as the target). This is a sequence-to-vector RNN.

Alternatively, you could set `return_sequences=True` for all RNN layers to create a sequence-to-sequence RNN. You can train this model using random windows from the time series, with sequences of the same length as the inputs as the targets. Each target sequence should have seven values per time step (e.g., for time step  $t$ , the target should be a vector containing the values at time steps  $t + 1$  to  $t + 7$ )

- **Q4** Which neural network architecture could you use to classify videos?
- **Answer 4** To classify videos based on their visual content, one possible architecture could be to take (say) one frame per second, then run every frame through the same convolutional neural network (e.g., a pretrained Xception model, possibly frozen if your dataset is not large), feed the sequence of outputs from the CNN to a sequence-to-vector RNN, and finally run its output through a softmax layer, giving you all the class probabilities.

For training you would use cross entropy as the cost function.

# Difficulties with RNNs

- **Q 5** : What are the main difficulties when training RNNs? How can you handle them?
- **Answer 5** The two main difficulties when training RNNs are unstable gradients (exploding or vanishing) and a very limited short-term memory. These problems both get worse when dealing with long sequences. To alleviate the unstable gradients problem, you can use a smaller learning rate, use a saturating activation function such as the hyperbolic tangent (which is the default), and possibly use gradient clipping, Layer Normalization, or dropout at each time step. To tackle the limited short-term memory problem, you can use LSTM or GRU layers (this also helps with the unstable gradients problem).
- **Q 6**: Why would you want to use 1D convolutional layers in an RNN?
- **Answer 6** An RNN layer is fundamentally sequential. To compute the outputs at time step  $t$ , the RNN has to first compute the outputs at all earlier time steps. This makes it **impossible to parallelize**. On the other hand, a 1D convolutional layer lends itself well to parallelization since it does not hold a state between time steps. In other words, it has no memory: the output at any time step can be computed based only on a small window of values from the inputs without having to know all the past values. Moreover, since a 1D convolutional layer is not recurrent, it suffers less from unstable gradients. One or more 1D convolutional layers can be useful in an RNN to efficiently preprocess the inputs, for example to reduce their temporal resolution (downsampling) and thereby help the RNN layers detect long-term patterns. In fact, it is possible to use only convolutional layers, for example by building a WaveNet architecture.

# Appendix

## Keras API Documentation

# Keras API Documentation, RNN

```
tf.keras.layers.RNN(  
    cell, return_sequences=False, return_state=False, go_backwards=False,  
    stateful=False, unroll=False, time_major=False, **kwargs  
)
```

## Arguments

- **cell**: A RNN cell instance. A RNN cell is a class that has:
  - a `call(input_at_t, states_at_t)` method, returning `(output_at_t, states_at_t_plus_1)`. The `call` method of the cell can also take the optional argument `constants`, see section "Note on passing external constants" below.
  - a `state_size` attribute. This can be a single integer (single state) in which case it is the size of the recurrent state (which should be the same as the size of the cell output). This can also be a list/tuple of integers (one size per state).
  - a `output_size` attribute. This can be a single integer or a `TensorShape`, which represent the shape of the output. For backward compatible reason, if this attribute is not available for the cell, the value will be inferred by the first element of the `state_size`.
  - It is also possible for `cell` to be a list of RNN cell instances, in which cases the cells get stacked on after the other in the RNN, implementing an efficient stacked RNN.
- **return\_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence.
- **return\_state**: Boolean. Whether to return the last state in addition to the output.
- **go\_backwards**: Boolean (default False). If True, process the input sequence backwards and return the reversed sequence.
- **stateful**: Boolean (default False). If True, the last state for each sample at index *i* in a batch will be used as initial state for the sample of index *i* in the following batch.
- **unroll**: Boolean (default False). If True, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.
- **input\_dim**: dimensionality of the input (integer). This argument (or alternatively, the keyword argument `input_shape`) is required when using this layer as the first layer in a model.
- **input\_length**: Length of input sequences, to be specified when it is constant. This argument is required if you are going to connect Flatten then Dense layers upstream (without it, the shape of the dense outputs cannot be computed). Note that if the recurrent layer is not the first layer in your model, you would need to specify the input length at the level of the first layer (e.g. via the `input_shape` argument)
- **Input shape**
- 3D tensor with shape `(batch_size, timesteps, input_dim)`.

# Keras API Documentation, RNN, continued

## Output shape

- *if return\_state*: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each with shape (batch\_size, units). For example, the number of state tensors is 1 (for RNN and GRU) or 2 (for LSTM).
- *if return\_sequences*: 3D tensor with shape (batch\_size, timesteps, units).
- else, 2D tensor with shape (batch\_size, units).

## Masking

- This layer supports masking for input data with a variable number of timesteps. To introduce masks to your data, use an Embedding layer with the mask\_zero parameter set to True.

## Note on using statefulness in RNNs

- You can set RNN layers to be 'stateful', which means that the states computed for the samples in one batch will be reused as initial states for the samples in the next batch. This assumes a one-to-one mapping between samples in different successive batches.
- To enable statefulness: - specify stateful=True in the layer constructor. - specify a fixed batch size for your model, by passing if sequential model: batch\_input\_shape=(...) to the first layer in your model. else for functional model with 1 or more Input layers: batch\_shape=(...) to all the first layers in your model. This is the expected shape of your inputs including the batch size. It should be a tuple of integers, e.g. (32, 10, 100). - specify shuffle=False when calling fit().
- To reset the states of your model, call .reset\_states() on either a specific layer, or on your entire model.

## Note on specifying the initial state of RNNs

- You can specify the initial state of RNN layers symbolically by calling them with the keyword argument initial\_state. The value of initial\_state should be a tensor or list of tensors representing the initial state of the RNN layer.
- You can specify the initial state of RNN layers numerically by calling reset\_states with the keyword argument states. The value of states should be a numpy array or list of numpy arrays representing the initial state of the RNN layer.
- Note on passing external constants to RNNs
- You can pass "external" constants to the cell using the constants keyword argument of RNN.\_\_call\_\_ (as well as RNN.call) method. This requires that the cell.call method accepts the same keyword argument constants. Such constants can be used to condition the cell transformation on additional static inputs (not changing over time), a.k.a. an attention mechanism.

# Keras API Documentation, LSTM

## LSTM

- `keras.layers.LSTM(units, activation='tanh', recurrent_activation='sigmoid', use_bias=True, kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal', bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None, recurrent_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, recurrent_constraint=None, bias_constraint=None, dropout=0.0, recurrent_dropout=0.0, implementation=1, return_sequences=False, return_state=False, go_backwards=False, stateful=False, time_major=False, unroll=False, **kwargs)`
- Long Short-Term Memory layer - Hochreiter 1997.
- Based on available runtime hardware and constraints, this layer will choose different implementations (cuDNN-based or pure-TensorFlow) to maximize the performance. If a GPU is available and all the arguments to the layer meet the requirement of the CuDNN kernel, the layer will use a fast cuDNN implementation.
- The requirements to use the cuDNN implementation are:
  - `activation == tanh`
  - `recurrent_activation == sigmoid`
  - `recurrent_dropout == 0`
  - `unroll` is `False`
  - `use_bias` is `True`
  - Inputs, if use masking, are strictly right-padded.
  - Eager execution is enabled in the outermost context.

# Keras API Documentation, LSTM

## Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use. Default: hyperbolic tangent (`tanh`). If you pass `None`, no activation is applied (i.e., "linear" activation:  $a(x) = x$ ).
- **recurrent\_activation**: Activation function to use for the recurrent step. Default: `sigmoid`. If you pass `None`, no activation is applied (i.e., "linear" activation:  $a(x) = x$ ).
- **use\_bias**: Boolean, whether the layer uses a bias vector. Default: `True`.
- **kernel\_initializer**: Initializer for the kernel weights matrix, used for the linear transformation of the inputs. Default: `glorot_uniform`.
- **recurrent\_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. Default: `orthogonal`.
- **bias\_initializer**: Initializer for the bias vector. Default: `zeros`.
- **unit\_forget\_bias**: Boolean (Default `True`). If `True`, add 1 to the bias of the forget gate at initialization. Setting it to `true` will also force `bias_initializer="zeros"`.
- **kernel\_regularizer**: Regularizer function applied to the `kernel` weights matrix. Default: `None`.
- **recurrent\_regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix. Default: `None`.
- **bias\_regularizer**: Regularizer function applied to the bias vector. Default: `None`.



# Keras API Documentation, LSTM, continued

- **activity\_regularizer**: Regularizer function applied to the output of the layer (its "activation"). Default: `None`.
- **kernel\_constraint**: Constraint function applied to the `kernel` weights matrix. Default: `None`.
- **recurrent\_constraint**: Constraint function applied to the `recurrent_kernel` weights matrix.
- **bias\_constraint**: Constraint function applied to the bias vector (see constraints). Default: `None`.
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Default: 0.
- **recurrent\_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state. Default: 0.
- **return\_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence. Default: `False`.
- **return\_state**: Boolean (Default: `False`) Whether to return the last state in addition to the output. The returned elements of the states list are the hidden state and the cell state, respectively.
- **go\_backwards**: Boolean (default `False`). If `True`, process the input sequence backwards and return the reversed sequence.
- **stateful**: Boolean (default `False`). If `True`, the last state for each sample at index `i` in a batch will be used as initial state for the sample of index `i` in the following batch.
- **time\_major**: The shape format of the inputs and outputs tensors. If `True`, the `inputs` and `outputs` will be in shape `[timesteps, batch, feature]`, whereas in the `False` case, it will be `[batch, timesteps, feature]`. Using `time_major = True` is a bit more efficient because it avoids transposes at the beginning and end of the RNN calculation. However, most TensorFlow data is batch-major, so **by default** this function accepts input and emits output in batch-major form.
- **unroll**: Boolean (default `False`). If `True`, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.

# Keras API Documentation, LSTM

## Call arguments

- `inputs`: A 3D tensor with shape `[batch, timesteps, feature]`.
- `mask`: Binary tensor of shape `[batch, timesteps]` indicating whether a given timestep should be masked (optional, defaults to `None`). An individual `True` entry indicates that the corresponding timestep should be utilized, while a `False` entry indicates that the corresponding timestep should be ignored.
- `training`: Python `boolean` indicating whether the layer should behave in training mode or in inference mode. This argument is passed to the cell when calling it. This is only relevant if `dropout` or `recurrent_dropout` is used (optional, defaults to `None`).
- `initial_state`: List of initial state tensors to be passed to the first call of the cell (optional, defaults to `None` which causes creation of zero-filled initial state tensors).

## Appendix:

# Machine Translation, Historical Perspective

# Statistical Machine Translation

- The statistical approach to machine translation outperformed the classical rule-based methods to become the de-facto standard technique in 1990-s.
- Since the inception at the end of the 1980s, the most popular models for statistical machine translation have been sequence-based. In these models, the basic units of translation are words or sequences of words.
- These kinds of models are simple and effective, and they work well for many language pairs.
- The most widely used techniques were phrase-based and focus on translating sub-sequences of the source text piecewise.
- Statistical Machine Translation (SMT) has been the dominant translation paradigm for decades. Practical implementations of SMT are generally phrase-based systems (PBMT) which translate sequences of words or phrases where the lengths may differ.
- Although effective, statistical machine translation methods suffered from a narrow focus on the phrases being translated, losing the broader nature of the target text.
- The hard focus on data-driven approaches also meant that methods may have ignored important syntax structures, even if they were known to linguists.
- Finally, the statistical approaches required careful tuning of each module in the translation pipeline.

# Machine Translation

- *Machine translation is the task of automatically converting a source text in one language to a text in another language.*
- Automatic or machine translation is one of the most challenging artificial intelligence tasks given the fluidity of human language.
- Several decades ago, we hoped to use rule-based translation systems.
- In the 1990-s rule-based translation systems were replaced with statistical methods.
- We need to understand the challenges of machine translation and the effectiveness of various neural machine translation models.
  - Machine translation is challenging because of the inherent ambiguity and flexibility of human language.
  - **Statistical machine translation** uses a series of models that learn to translate from examples.

# Rules-Based Machine Translation

- In a machine translation task, the input consists of a sequence of symbols in some language, and the computer program must convert this sequence into a sequence of symbols in another language.
- Given a sequence of text in a source language, there is no one single best translation of that text to another language. Natural ambiguity and flexibility of human language make the challenge of automatic machine translation difficult.
- Accurate translation requires background knowledge in order to resolve ambiguity and establish the content of the sentence.
- "Classical" machine translation methods involve rules for converting text in the source language to the target language. The rules are often developed by linguists and may operate at the lexical, syntactic, or semantic level. This focus on rules gives the name to this area of study: *Rule-based Machine Translation*, or RBMT.
- RBMT is characterized with the explicit use and manual creation of linguistically informed rules and representations.
- The key limitations of the classical machine translation approaches are both the expertise required to develop the rules, and the vast number of rules and exceptions required.

# Phase Alignment Approach

- Goal: know which word or phrases in source language would translate to what words or phrases in target language. Some languages, like French and English are, believe it or not, quite close and words and phrases could align well, sometime.
- For example in these two simple sentences, apart from a spurious French *Le*, the alignment is rather close.

	Le
Japan	Japon
shaken	secoué
by	par
two	deux
new	nouveaux
quakes	séismes

- However words and phrases do not always map one-to-one, there could be many-to-one correspondence and many-to-many, as in bottom examples on the left and right respectively.

- In short, classical, even statistical translations, required excessive amounts of knowledge and skills

The	Le
balance	reste
was	appartenait
the	
territory	
of	aux
the	autochtones
aboriginal	
people	

The	Les
poor	pauvres
don't	sont
have	
any	
money	démunis

# Statistical Machine Translation

- Statistical machine translation, or SMT for short, relies on statistical models that learn to translate text from a source language to a target language given a large corpus of examples.
- The task of using a statistical model can be described as follows:
- *Given a sentence  $T$  in the target language, we seek the sentence  $S$  from which the translator produced  $T$ . We know that our chance of error is minimized by choosing that sentence  $S$  that is most probable given  $T$ . Thus, we wish to choose  $S$  so as to maximize conditional probability  $P(S|T)$ .*
- This formal specification makes the maximizing of the probability of the output sequence given the input sequence of text explicit. It also makes the notion of there being a suite of candidate translations explicit and the need for a search process or decoder to select the one most likely translation from the model's output probability distribution.
- Given a text in the source language, what is the most probable translation in the target language? [...] how should one construct a statistical model that assigns high probabilities to "good" translations and low probabilities to "bad" translations?
- The approach is data-driven, requiring a large corpus of examples with both source and target language text. This means that linguists are no longer required to specify the rules of translation.
- This approach does not need a complex ontology of interlingua concepts, nor does it need handcrafted grammars of the source and target languages, nor a hand-labeled treebank. All it needs is data-sample translations from which a translation model can be learned.

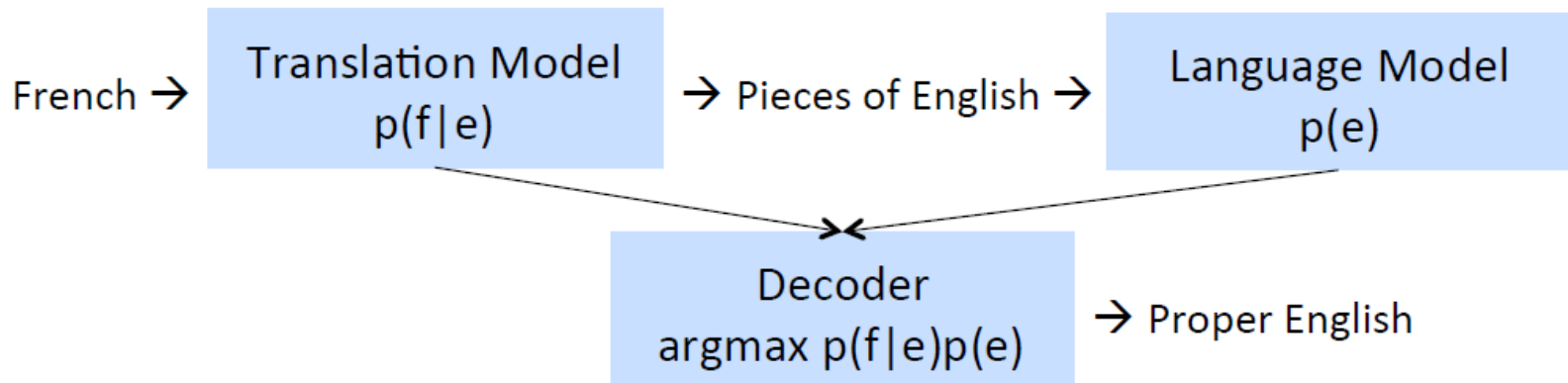


# Formal approach to Statistical Machine Translation

- Source language  $f$ , e.g. French
- Target language  $e$ , e.g. English
- Probabilistic formulation (using Bayes rule)

$$\hat{e} = \operatorname{argmax}_e p(e|f) = \operatorname{argmax}_e p(f|e)p(e)$$

- Translation model  $p(f|e)$  trained on parallel corpus
- Language model  $p(e)$  trained on English only corpus (lots, free!)
- French must have translated all English literature, already. So you have  $p(f|e)$



# Search for the Best Hypothesis

- Since it was moderately easy to collect large language models, with probabilities for single words, bigrams, tri-grams, and so on, we hoped that we could use those language models to construct most probable translations.
- Those searches turn out to be very computationally consuming for sentences of any reasonable length.
- Example below illustrates many possibilities that need to be examined and their probabilities compared even for a short sentence.

