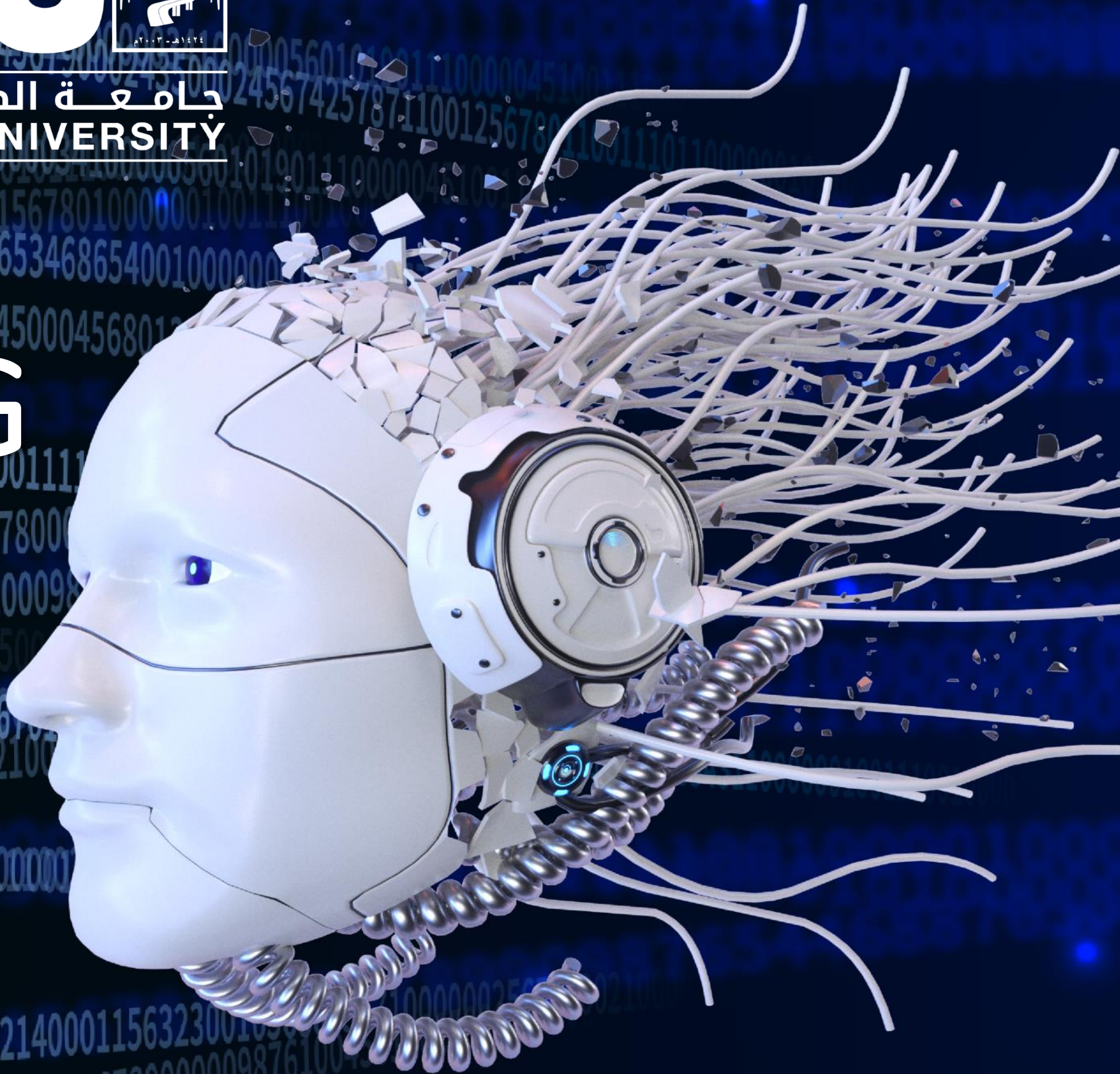
جامعة الطائف
TAIF UNIVERSITY

DEEP LEARNING



Assoc. Prof. Dr. Salha Alzahrani

Lecture Notes for MSc. in Data Science

Introduction

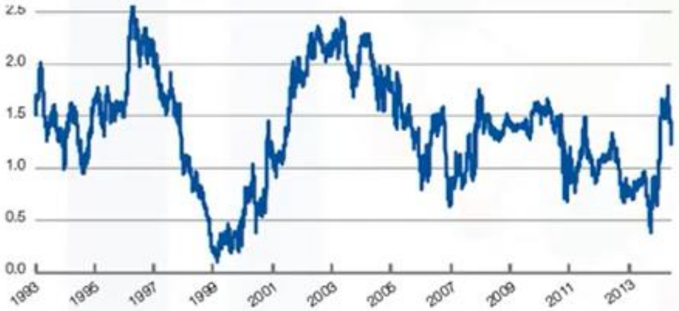
- The neural network architectures we discussed in the previous chapters take in fixed sized input and provide fixed sized output.
- This chapter will lift this constraint by introducing Recurrent Neural Networks (RNNs).
- RNNs help us deal with sequences of variable length by defining a recurrence relation over these sequences (hence the name).
- The ability to process arbitrary sequences of input makes RNNs applicable for natural language processing (NLP) and speech recognition tasks.
- In fact, RNNs can be applied to any problem since it has been proven that they are Turing complete – theoretically, they can simulate any program that a regular computer would not be able to compute. For example, Google's DeepMind has proposed a model called Differentiable Neural Computer, which can learn how to execute simple algorithms, such as sorting.

In this chapter, we will cover the following topics:

- Sequential data
- Recurrent neural network (RNN)
- Long short-term memory (LSTM)
- Gated recurrent unit (GRU)
- Language modeling
- Sequence to sequence learning (Seq2Seq)
- Speech recognition
- Example: Text generation with LSTM
- Example: Sentiment classification using pre-trained language models (transformers)

Sequential data

- Sequential Data – data points with dependencies



- Whenever the points in a dataset are **dependent** on the other points, the data is said to be sequential.
- Common examples of sequential data include **stock price, sensor data**, **sentences, gene sequences, and weather data**.

GUGCAUCUGACUCCUGAGGAGAAG •••

- Not handled well by traditional Neural Networks

Recurrent neural networks

- RNN is a type of neural network, which can process sequential data with variable length.
- Examples of such data include the words of a sentence or the price of a stock in various moments of time.
- By using the word sequential, we imply that the elements of the sequence are related to each other and their order matters. For example, if we take a book and shuffle randomly all the words in it, the text will loose it's meaning, even though we'll still know the individual words.

- RNNs get their name because they apply the same function over a sequence recurrently. We can define an RNN as a recurrence relation:

$$s_t = f(s_{t-1}, x_t)$$

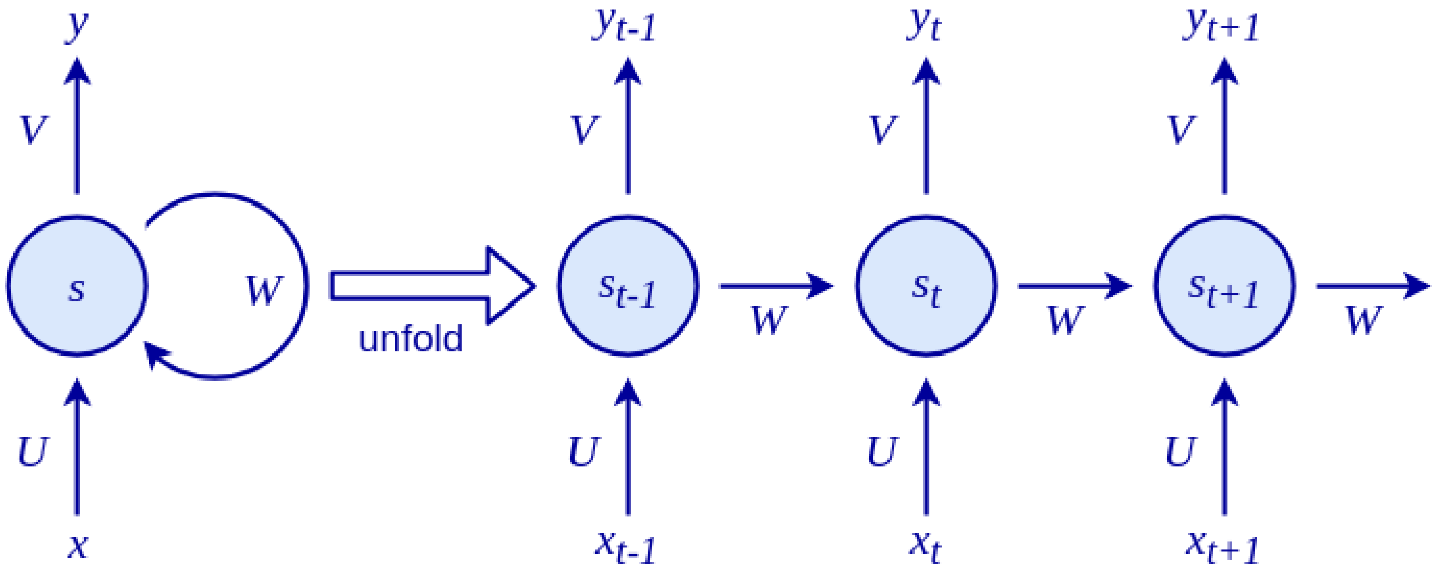
Here, *f* is a differentiable function, s_t is a vector of values called internal network state (at step *t*), and x_t is the network input at step *t*.

Recurrent neural networks

- Unlike regular networks, where the state only depends on the current input (and network weights), here s_t is a function of both the current input, as well as the previous state, s_{t-1}.
- You can think of s_{t-1} as the network's summary of all previous inputs.
- The recurrence relation defines how the state evolves step by step over the sequence via a feedback loop over previous states, as illustrated in the following diagram:

Left: Visual illustration of the RNN recurrence relation: $s_t = s_{t-1} * W + x_t * U$. The final output will be $y_t = s_t * V$.

Right: RNN states recurrently unfolded over the sequence *t-1, t, t+1*. Note that the parameters *U, V, and W* are shared between all the steps



Recurrent neural networks

- The RNN has three sets of parameters (or weights):
  - *U* transforms the input x_t to the state s_t
  - *W* transforms the previous state s_{t-1} to the current state s_t
  - *V* maps the newly computed internal state s_t to the output y_t

- *U*, *V*, and *W* apply linear transformation over their respective inputs. The most basic case of such a transformation is the familiar weighted sum we know and love.
- We can now define the internal state and the network output as follows:

$$s_t = f(s_{t-1} * W + x_t * U)$$

$$y_t = s_t * V$$

Here, *f* is the non-linear activation function (such as tanh, sigmoid, or ReLU).

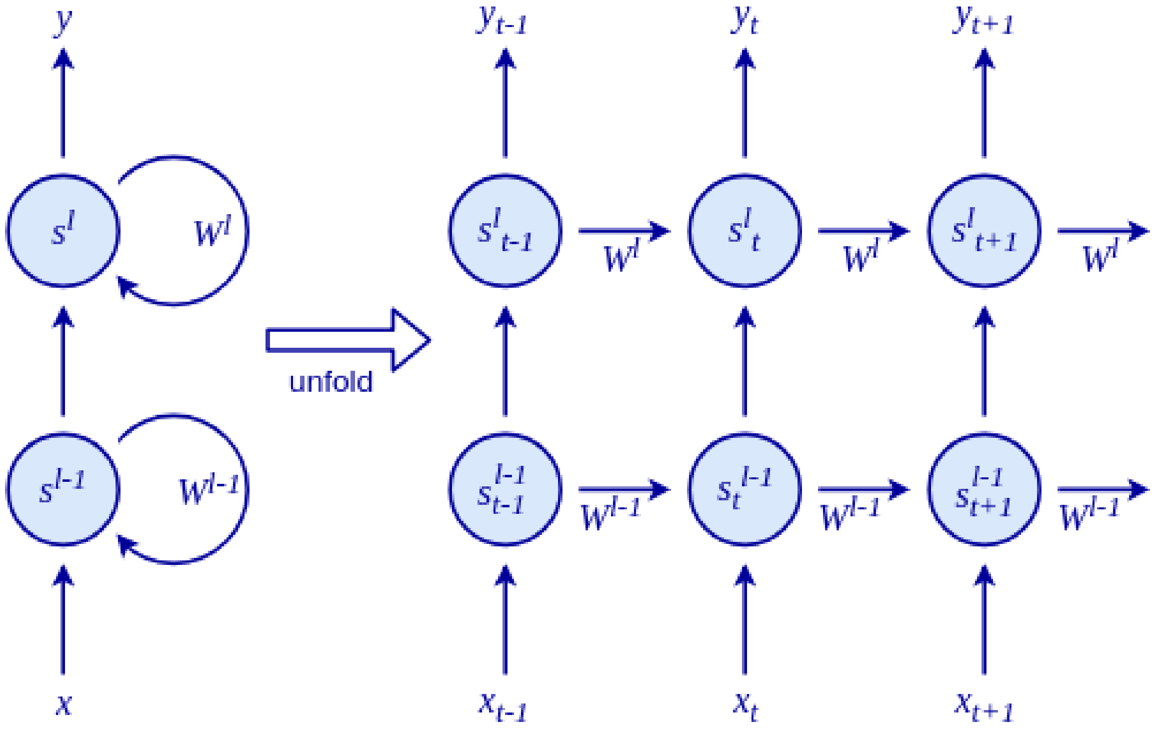
Recurrent neural networks

- For example, in a word-level language model, the input *x* will be a sequence of words encoded in input vectors $(x_1 \dots x_t \dots)$. The state *s* will be a sequence of state vectors $(s_1 \dots s_t \dots)$.
- Finally, the output *y* will be a sequence of probability vectors $(y_1 \dots y_t \dots)$ of the next words in the sequence.
- Note that in a RNN, each state is dependent on all previous computations via this recurrence relation. An important implication of this is that RNNs have memory over time, because the states *s* contains information based on the previous steps.
- In theory, RNNs can remember information for an arbitrarily long period of time, but in practice they are limited to looking back only a few steps.
- The RNN we described is somewhat equivalent to a single layer regular neural network (with an additional recurrence relation). As we now know from Chapter 2, *Neural Networks*, a network with a single layer has some serious limitations.
- As with regular networks, we can stack multiple RNNs to form a **stacked RNN**. The cell state s^l_t of a RNN cell at level *l* at time *t* will take the output y_t^{l-1} of the RNN cell from level *l-1* and previous cell state s^l_{t-1} of the cell at the same level *l* as the input:

$$s_t^l = f(s_{t-1}^l, y_t^{l-1})$$

Recurrent neural networks

In the following diagram, we can see an unfolded, stacked RNN:

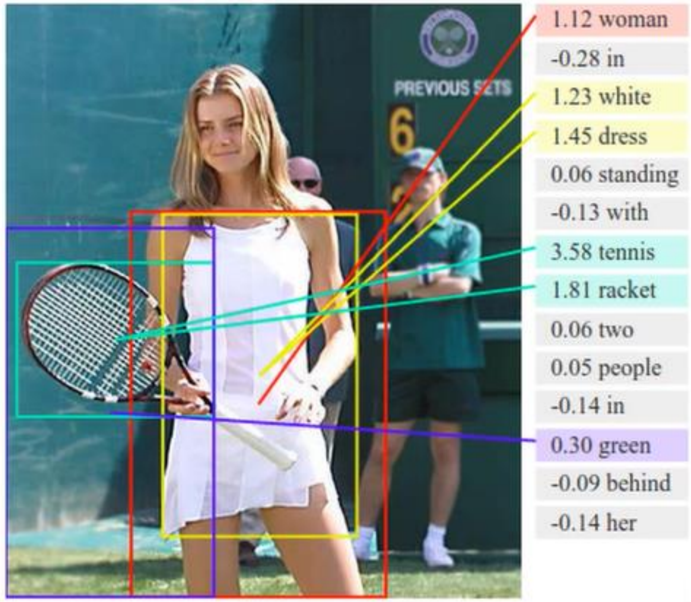


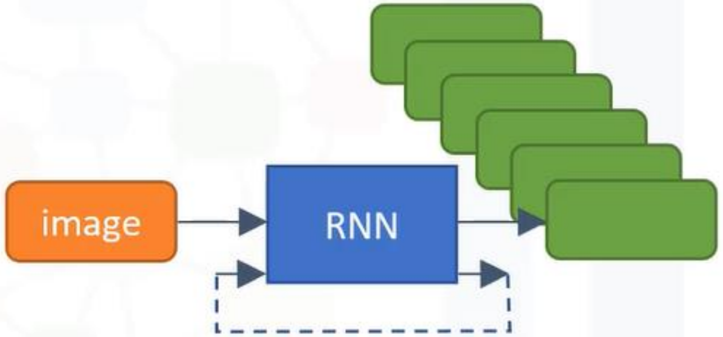
Recurrent neural networks

- Because RNNs are not limited to processing fixed size inputs, they really expand the possibilities of what we can compute with neural networks, such as sequences of different lengths or images of varied sizes. The following are some combinations:
  - ○ **One-to-one**: This is non-sequential processing, such as feedforward neural networks and convolutional neural networks. Note that there isn't much difference between a feedforward network and applying an RNN to a single time step. An example of one-to-one processing is image classification.
  - ○ **One-to-many**: This processing generates a sequence based on a single input, for example, caption generation from an image (https://arxiv.org/abs/1411.4555v2).
  - ○ **Many-to-one**: This processing outputs a single result based on a sequence, for example, sentiment classification from text.
  - ○ **Many-to-many indirect**: A sequence is encoded into a state vector, after which this state vector is decoded into a new sequence, for example, language translation (https://arxiv.org/abs/).
  - ○ **Many-to-many direct:** This outputs a result for each input step, for example, frame phoneme labeling in speech recognition (see the *Speech recognition* section for more details).

Recurrent neural networks

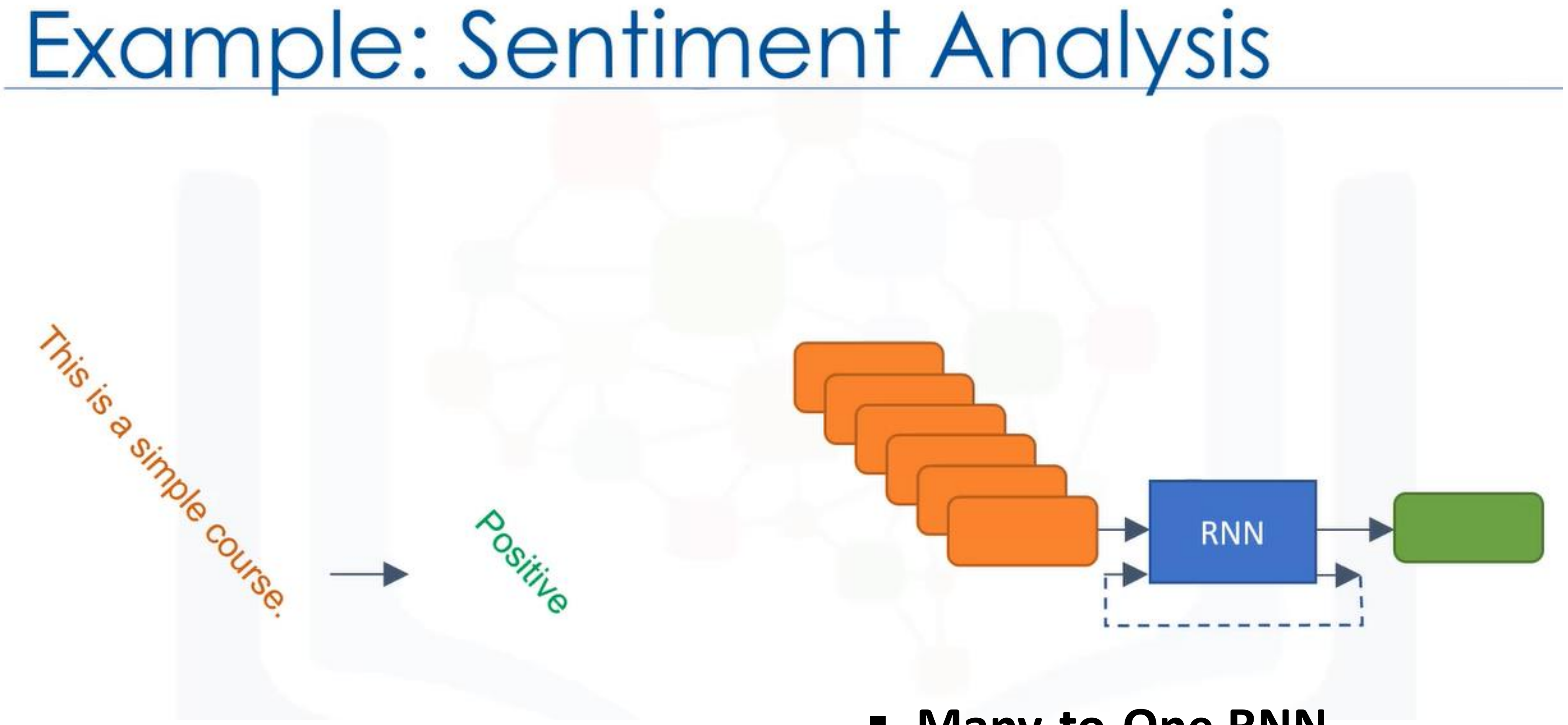
Example: Image Captioning





- **One-to-Many RNN**

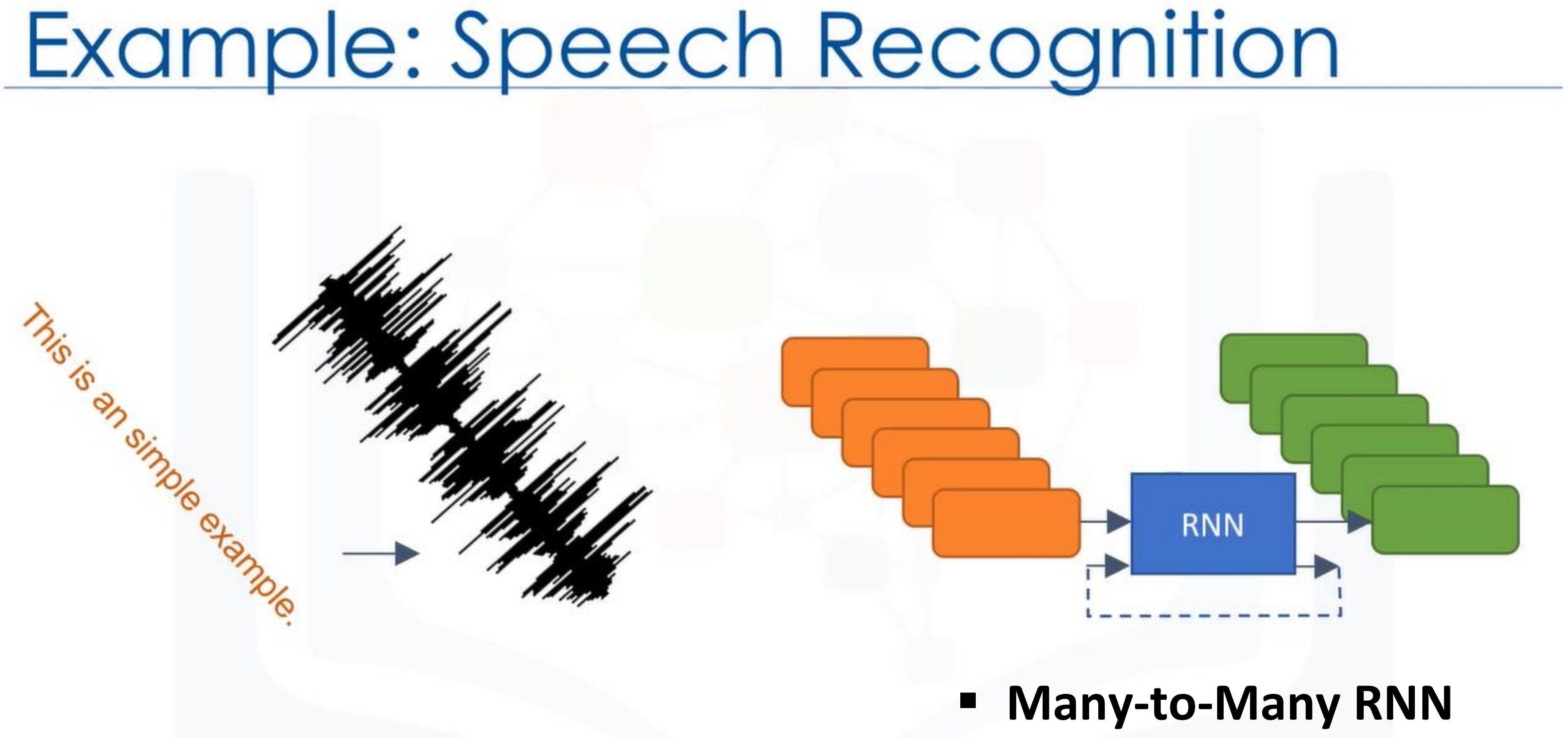
Recurrent neural networks



Example: Sentiment Analysis

- **Many-to-One RNN**

Recurrent neural networks

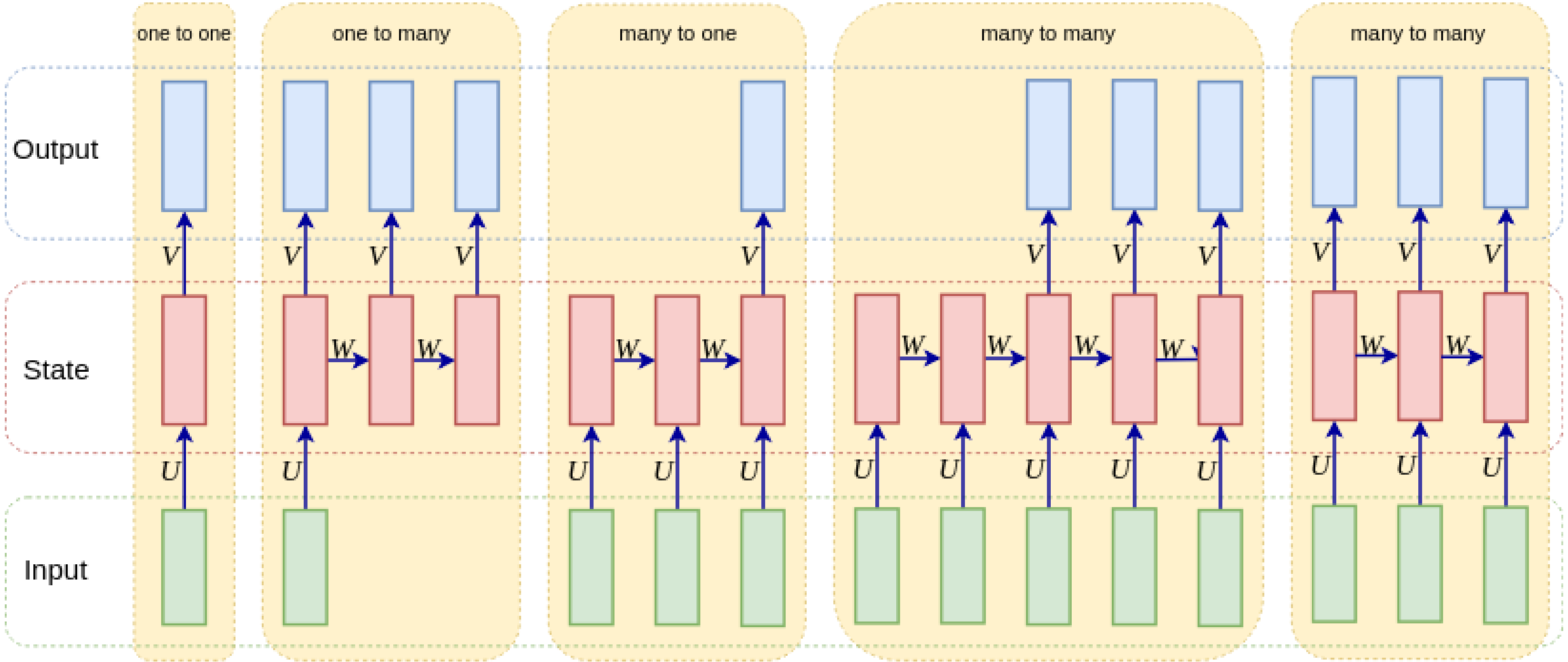


Example: Speech Recognition

- **Many-to-Many RNN**

Recurrent neural networks

The following is a graphical representation of the preceding input-output combinations (idea from http://karpathy.github.io/2015/05/21/rnn-effectiveness/):

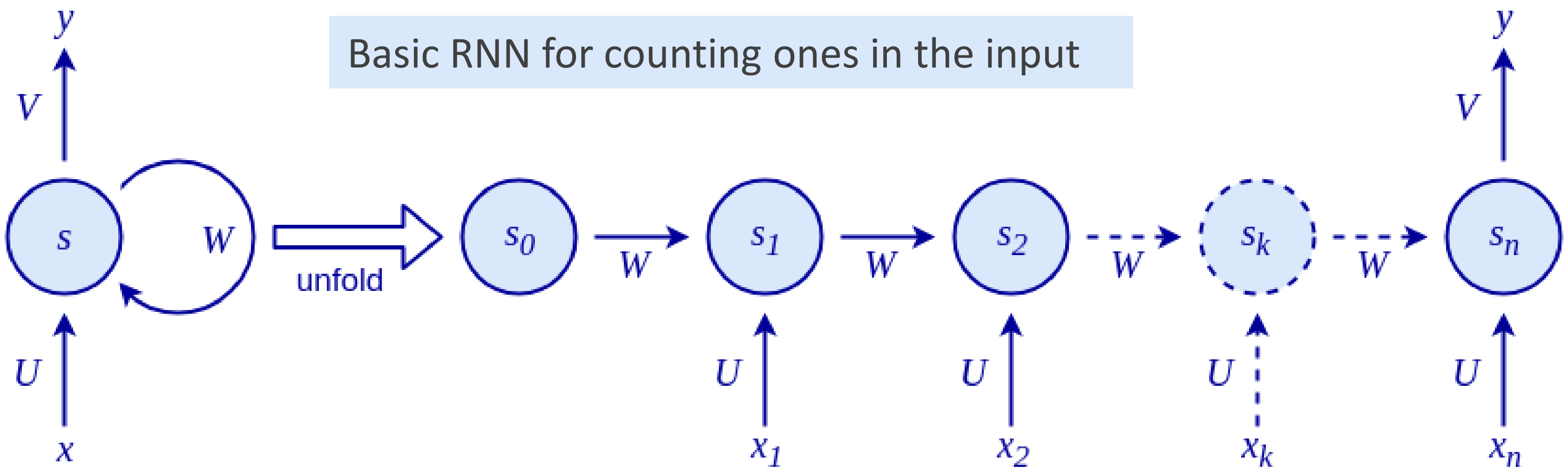


RNN input-output combinations

RNN implementation and training

- In the preceding section, we briefly discussed what RNNs are and what problems they can solve. Let's dive into the details of an RNN and how to train it with a very simple toy example: counting ones in a sequence.
- In this problem, we will teach a basic RNN how to count the number of ones in the input, and then output the result at the end of the sequence. This is an example of a "many-to-one" relationship.

Basic RNN for counting ones in the input



Backpropagation through time

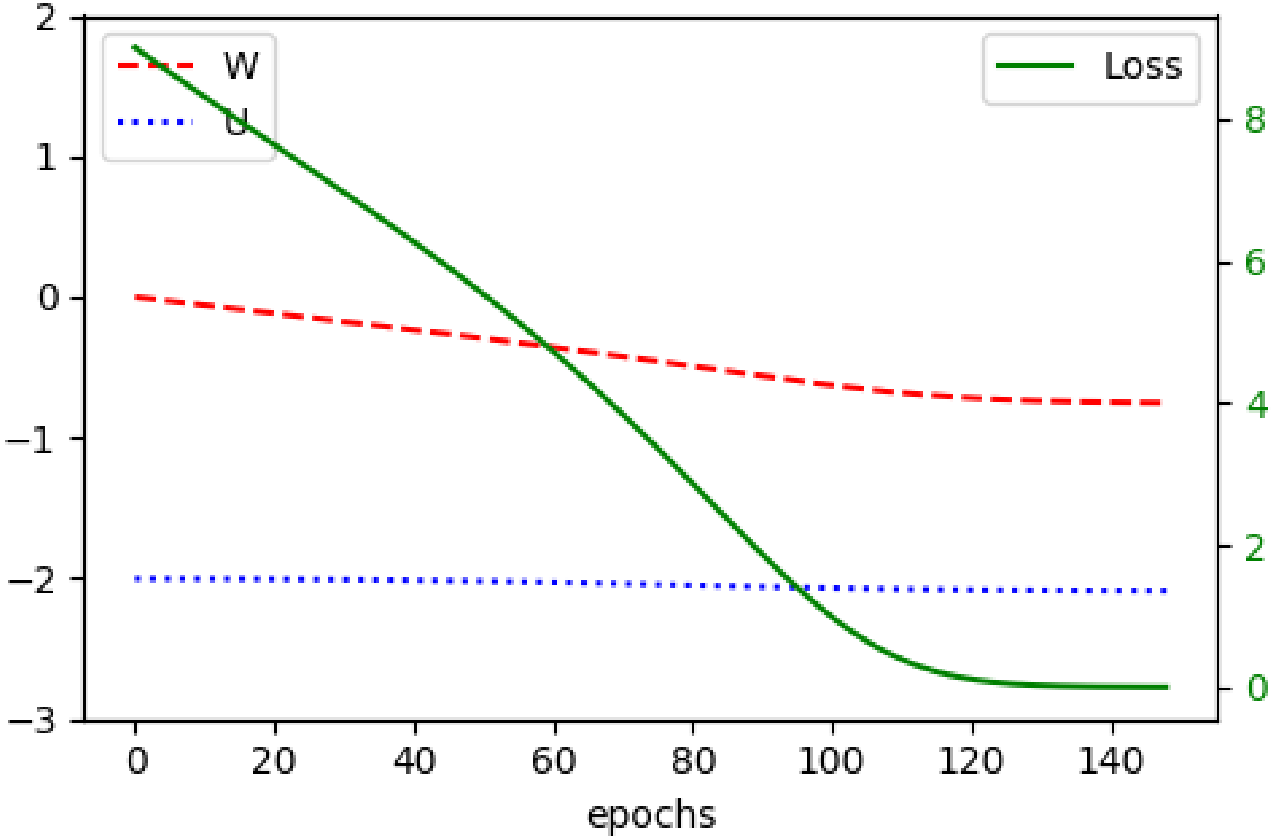
- The main difference between regular backpropagation and backpropagation through time is that the recurrent network is unfolded through time for a certain number of time steps (as illustrated in the preceding diagram).
- Once the unfolding is complete, we end up with a model that is quite similar to a regular multilayer feedforward network.
- That is, one hidden layer of that network represents one step through time. The only differences are that each layer has multiple inputs: the previous state s_{t-1} and the current input x_t.
- The parameters U and W are shared between all hidden layers.

- The forward pass unwraps the RNN along the sequence and builds a stack of states for each step.

- Because the weights W and U are shared across the layers, we'll accumulate the error derivatives for each recurrent step and in the end we'll update the weights with the accumulated value.

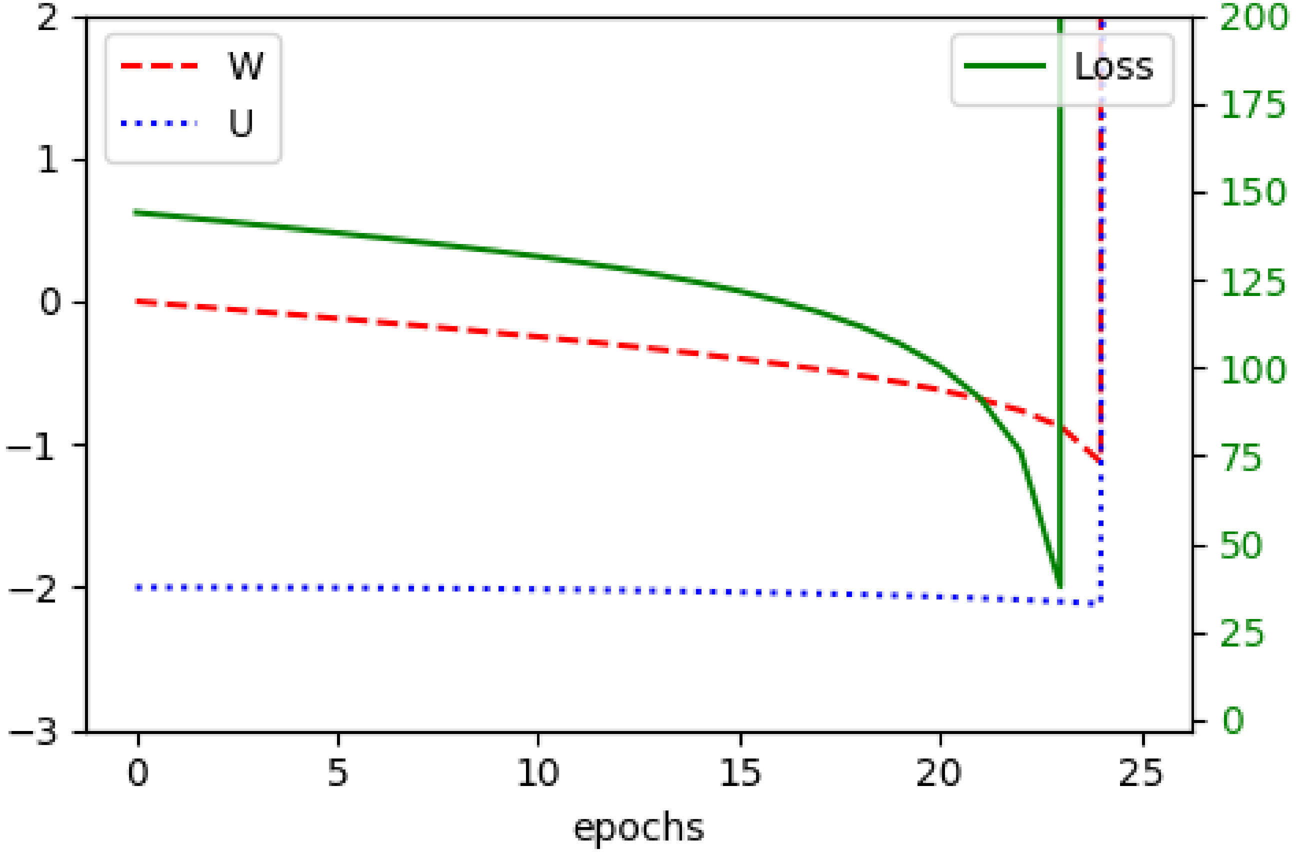
Backpropagation through time

The RNN loss and the uninterrupted line represents the loss, while the dashed lines represent the weights



Vanishing and exploding gradients

- The reason for these warnings is that the final parameters U and W end up as Not a Number (NaN). In the following graph, we can see the weight updates and loss during the training steps:



Vanishing and exploding gradients

- The weights slowly move toward the optimum and the loss decreases until it overshoots at epoch 23 (the exact epoch is unimportant, though). What happens is that the cost surface we are training on is highly unstable. Using small steps, we might move to a stable part of the cost function, where the gradient is low, and suddenly hits upon a jump in cost and a corresponding huge gradient. Because the gradient is so huge, it will have a big effect on our weights via the weight updates – they become NaNs (as illustrated by the jump outside the plot). This problem is known as **exploding gradients**.

- There is also the **vanishing (as opposed to exploding) gradients problem**, which we first mentioned in chapter 3, *Deep Learning Fundamentals*. The gradient decays exponentially over the number of steps to a point where it becomes extremely small in the earlier states. In effect, they are overshadowed by the larger gradients from more recent time steps, and the network's ability to retain the history of these earlier states vanishes. This problem is harder to detect because the training will still work and the network will produce valid outputs (unlike with exploding gradients). It just won't be able to learn long-term dependencies.

Vanishing and exploding gradients

Although vanishing and exploding gradients are present in regular neural networks, they are especially pronounced in RNNs. The reasons for this are as follows:

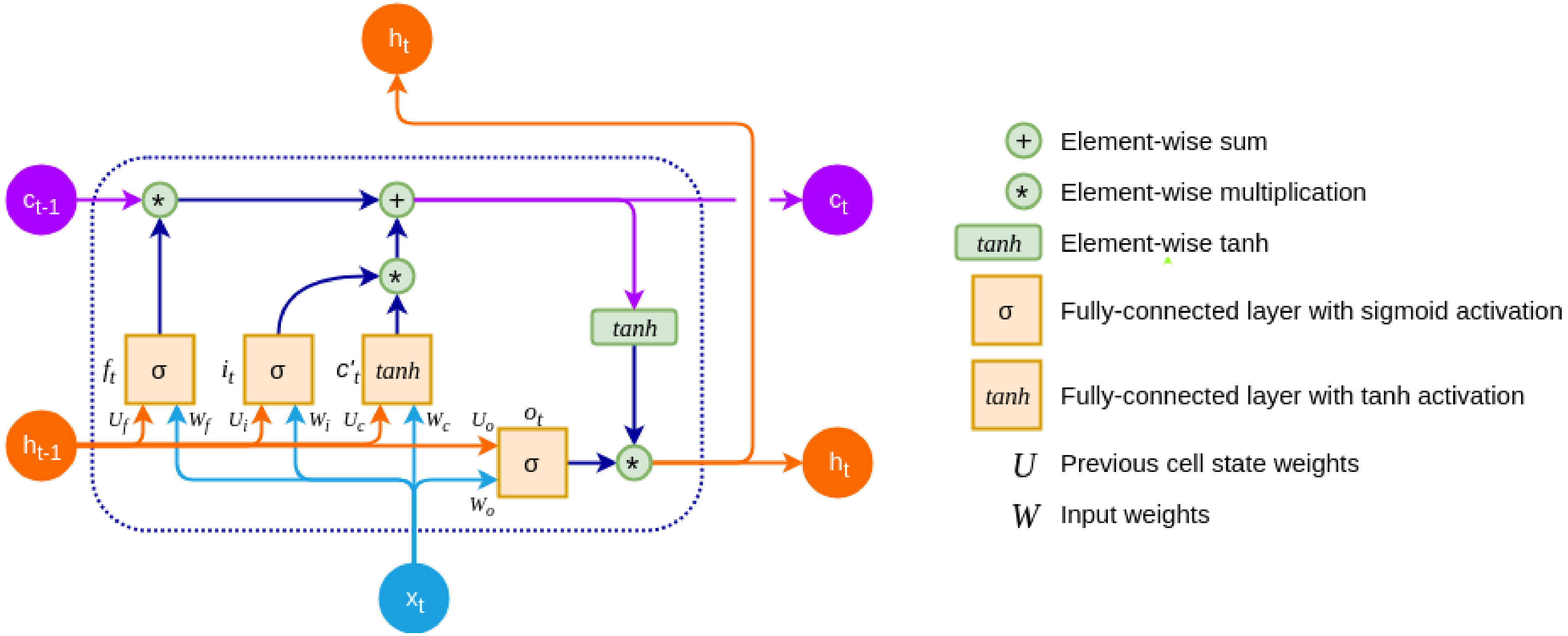
- Depending on the sequence's length, an unfolded RNN can be much deeper compared to a regular network.
- The weights W are shared across all steps. This means that the recurrence relation that propagates the gradient backward through time forms a geometric sequence.

Long short-term memory

- Hochreiter and Schmidhuber studied the problems of vanishing and exploding gradients extensively and came up with a solution called long short-term memory (LSTM) (https://www.bioinf.jku.at/publications/older/2604.pdf).
- LSTMs can handle long-term dependencies due to a specially crafted memory cell. In fact, they work so well that most of the current accomplishments in training RNNs on a variety of problems are due to the use of LSTMs.
- The key idea of LSTM is the cell state (in addition to the hidden RNN state), where the information can only be explicitly written in or removed so that the state stays constant if there is no outside interference. The cell state can only be modified by specific gates, which are a way to let information pass through. These gates are composed of a logistic sigmoid function and element-wise multiplication. Because the logistic function only outputs values between 0 and 1, the multiplication can only reduce the value running through the gate.
- **A typical LSTM is composed of three gates:** a forget gate, an input gate, and an output gate. The cell state, input, and output are all vectors, so the LSTM can hold a combination of different information blocks at each time step.

Long short-term memory

The following is a diagram of a LSTM cell (idea from http://colah.github.io/posts/2015-08-Understanding-LSTMs/):


h_t
c_{t-1}
*
+
c_t
*
tanh
f_t
σ
i_t
σ
c'_t
tanh
h_{t-1}
U_f
W_f
U_i
W_i
U_c
W_c
U_o
o_t
σ
*
h_t
W_o
x_t
+ Element-wise sum
* Element-wise multiplication
tanh Element-wise tanh
σ Fully-connected layer with sigmoid activation
tanh Fully-connected layer with tanh activation
U Previous cell state weights
W Input weights

Long short-term memory

- Before we continue, let's introduce some notations:
  - x_t, c_t, and h_t are the LSTM input, cell memory state, and output (or hidden state) in moment *t*. c'_t is the candidate cell state (more on that later). The input x_t and the previous cell output h_{t-1} are connected to each gate and the candidate cell vector with sets of weights *W* and *U*, respectively.
  - c_t is the cell state in moment *t.*
  - f_t, i_t, and o_t are the forget, input, and output gates of the LSTM cell.

- Let's start with the forget gate. As the name suggests, it decides whether we want to erase the cell state or not. The forget gate bases its decision on the output of the previous cell h_{t-1} and the current input x_t:

$$f_t = \sigma(W_f x_t + U_f h_{t-1})$$

It applies element-wise logistic functions on each element of the previous cell's vector c_{t-1}. Because the operation is element-wise, the values of this vector are squashed in the [0, 1] range. An output of 0 erases a specific c_{t-1} cell block completely and an output of 1 allows the information in that cell block to pass through. This means that the LSTM can get rid of irrelevant information in its cell state vector.

Long short-term memory

- The input gate decides what new information is going to be added to the memory cell. This is done in two parts. The first part decides whether information is going to be added. As in the input gate, it bases it decision on h_{t-1} and x_t. It outputs 0 or 1 through the logistic function for each cell block of the cell's vector. An output of 0 means that no information is added to that cell block's memory. As a result, the LSTM can store specific pieces of information in its cell state vector:

$$i_t = \sigma(W_i x_t + U_i h_{t-1})$$

- The candidate input to be added, c'_t, is based on the previous output h_{t-1} and the current input x_t. It is transformed via a *tanh* function:

$$c'_t = tanh(W_c x_t + U_c h_{t-1})$$

Long short-term memory

- The forget and input gates decide the new cell state by choosing which parts of the new and the old state to include:

$$c_t = f_t * c_{t-1} \oplus i_t * c'_t$$

- The output gate decides what the total cell output is going to be. It takes h_{t-1} and x_t as inputs and outputs 0 or 1 (via the logistic function) for each block of the cell's memory. An output of 0 means that the block doesn't output any information, while an output of 1 means that the block can pass through as a cell's output. The LSTM can thus output specific blocks of information from its cell state vector:

$$o_t = \sigma(W_o x_t + U_o h_{t-1})$$

- Finally, the LSTM cell output is transferred by a *tanh* function:

$$h_t = o_t * tanh(c_t)$$

Long short-term memory

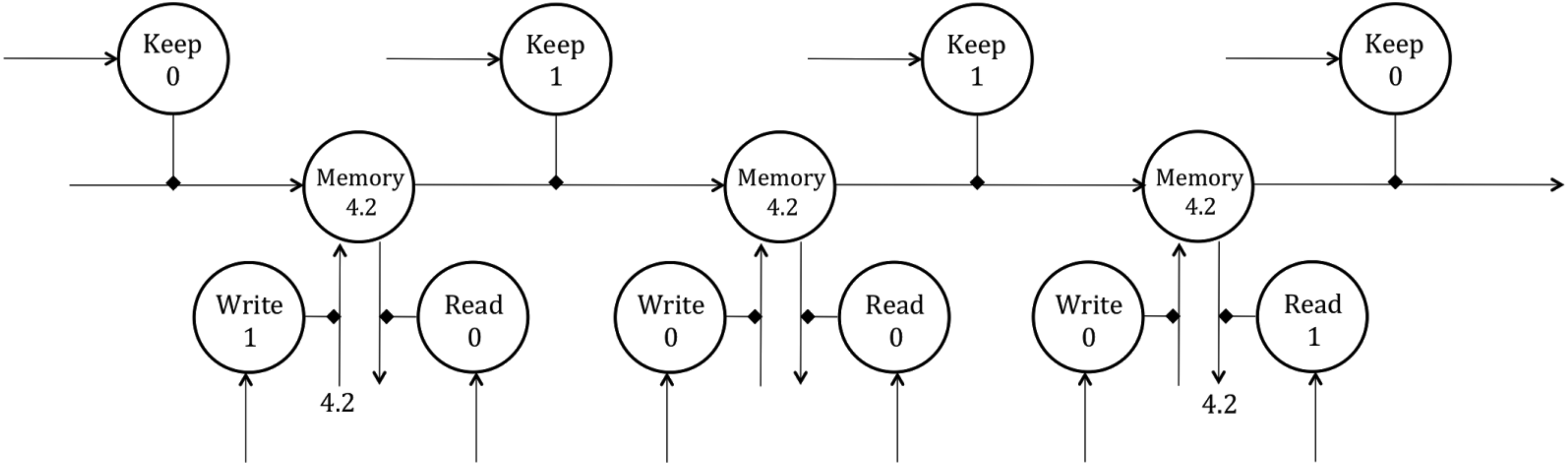
- Because all of these formulas are derivable, we can chain LSTM cells together just like we chain simple RNN states together and train the network via backpropagation through time.

- **But how does the LSTM protect us from vanishing gradients?**
  - Notice that the cell state is copied identically from step to step if the forget gate is 1 and the input gate is 0. Only the forget gate can completely erase the cell's memory. As a result, memory can remain unchanged over a long period of time.
  - Also, notice that the input is a *tanh* activation that's been added to the current cell's memory. This means that the cell memory doesn't blow up and is quite stable.

- Let's use an example to demonstrate how a LSTM is unfolded. We'll start by using the value of 4.2 as network input. The input gate is set to 1 so that the complete value is stored. Then, for the next two time steps, the forget gate is set to 1. In this way, all the information is kept throughout these steps and no new information is being added because the input gates are set to 0. Finally, the output gate is set to 1, and 4.2 is outputted and remains unchanged.

Long short-term memory

The following is an example of a LSTM unfolding through time (source: http://nikhilbuduma.com/2015/01/11/a-deep-dive-into-recurrent-neural-networks/):

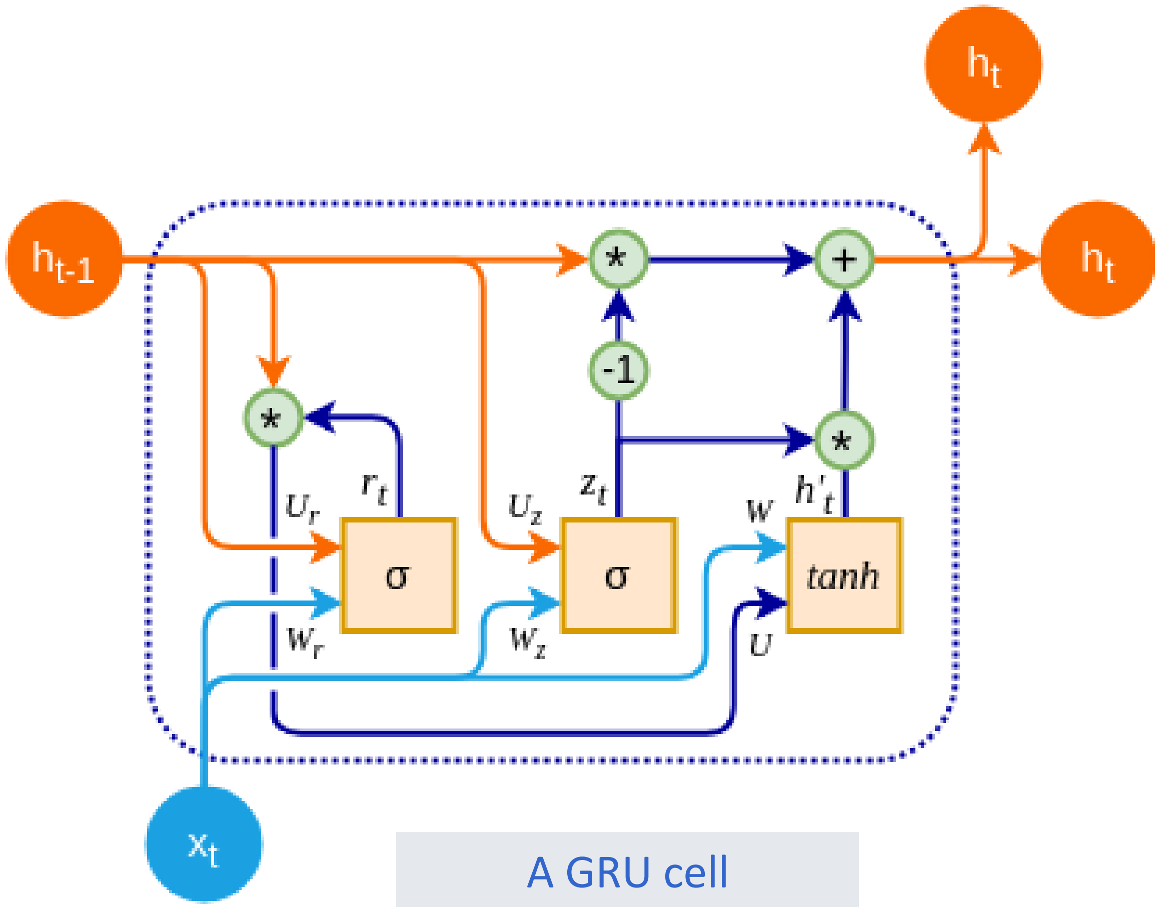


Unrolling a LSTM through time

Gated recurrent units

- A **gated recurrent unit** (GRU**)** is a type of recurrent block that was introduced in 2014 by Kyunghyun Cho et al. (https://arxiv.org/abs/1406.1078, https://arxiv.org/abs/1412.3555), as an improvement over LSTM (see the following diagram).

- A GRU unit usually has similar or better performance to a LSTM, but it does so with fewer parameters and operations:



A GRU cell

Gated recurrent units

- The GRU cell has two gates:
  - An **update gate** ***z**$_t$*, which is a combination of the input and forget LSTM gates. It decides what information to discard and what new information to include in its place, based on the network input x_t and the previous cell hidden state h_{t-1}. By combining the two gates, we can ensure that the cell will forget information, but only when we are going to include new information in its place:

$$z_t = \sigma(W_z x_t + U_z h_{t-1})$$

  - A **reset gate,** ***r**$_t$*, which uses the previous cell state h_{t-1} and the network input x_t to decide how much of the previous state to pass through:

$$r_t = \sigma(W_r x_t + U_r h_{t-1})$$

- Next, we have the candidate state, *h't*:

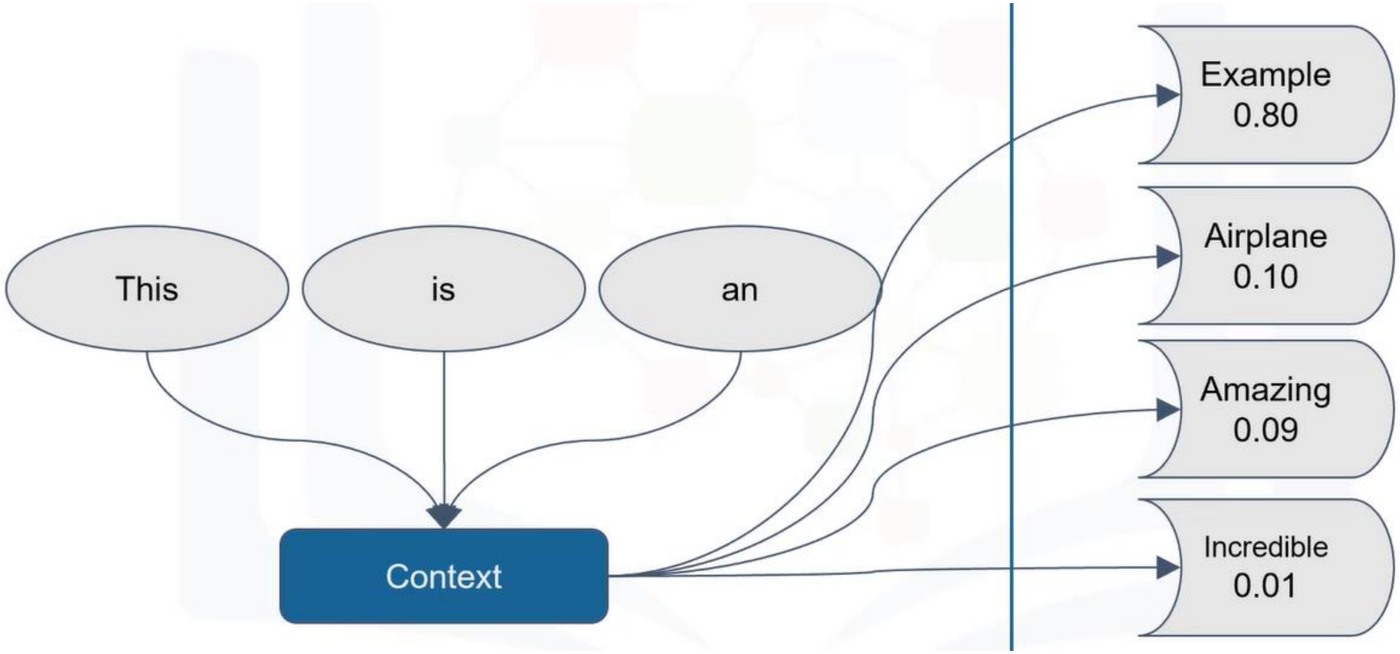
$$h'_t = tanh(W x_t + U(r_t * h_{t-1}))$$

- Finally, the GRU output h_t at time *t* is a linear interpolation between the previous output h_{t-1} and the candidate output h'_t:

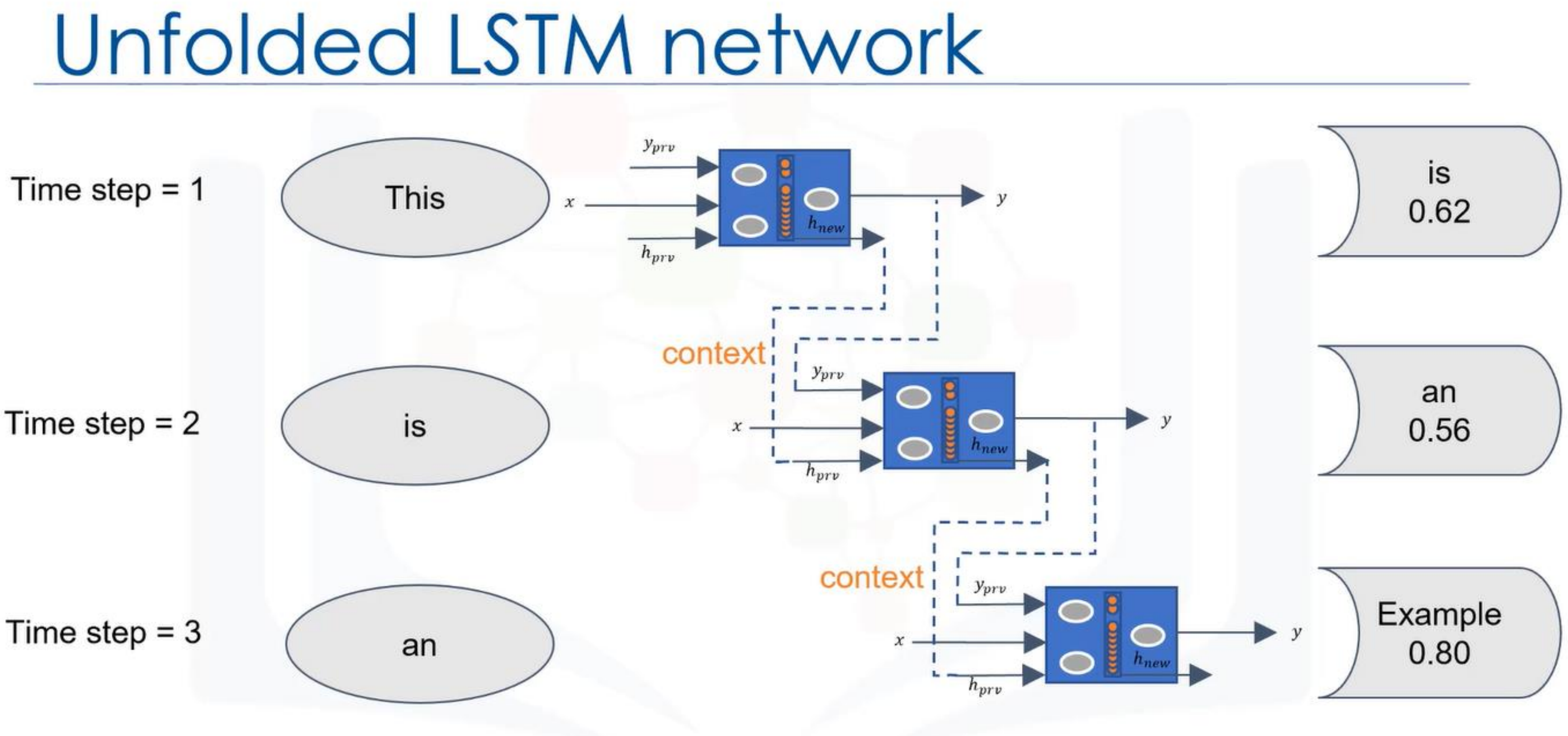
$$h_t = (1 - z_t) * h_{t-1} \oplus z_t * h'_t$$

Language modeling

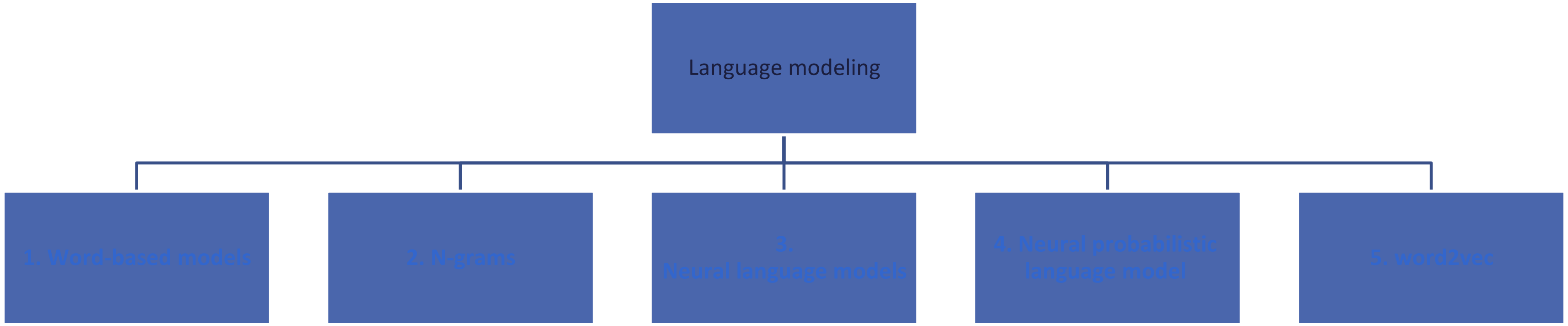
- Language modeling is the task of computing the probability of a sequence of words.
- A good language model can distinguish which phrase is most likely to be correct, based on the context of the conversation.
- This section will provide an overview of word and character-level language models.



Language modeling



Language modeling



Language modeling

1. Word-based models

- A word-based language model defines a probability distribution over sequences of words. Given a sequence of words of length *m*, it assigns a probability $P(w_1, \ldots, w_m)$ to the full sequence of words.

- We can use these probabilities as follows:
  - To estimate the likelihood of different phrases in natural language processing applications.
  - As a generative model to create new text. A word-based language model can compute the likelihood of a given word to follow a sequence of words.

Language modeling

2. N-grams

- The inference of the probability of a long sequence, say $w_1, ..., w_m$, is typically infeasible. Calculating the joint probability of $P(w_1, ... , w_m)$ would be done by applying the following chain rule:

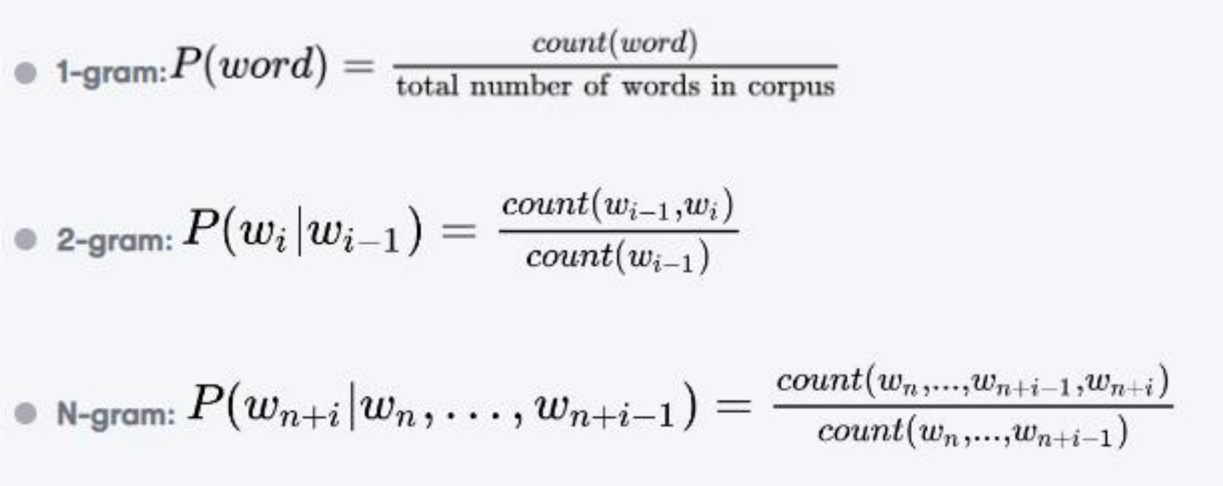
$$P(w_1, \ldots, w_m) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\ldots P(w_m|w_1, \ldots, w_{m-1})$$

- The probability of the later words given the earlier words would be especially difficult to estimate from the data. That's why this joint probability is typically approximated by an independence assumption that the *i*th word is only dependent on the *n-1* previous words. We'll only model the joint probabilities of combinations of *n* sequential words, called n-grams.

- For example, in the phrase *the quick brown fox*, we have the following n-grams:
  - **1-gram**: "The," "quick," "brown," and "fox" (also known as a unigram)
  - **2-grams**: "The quick," "quick brown," and "brown fox" (also known as a bigram)
  - **3-grams**: "The quick brown" and "quick brown fox" (also known as a trigram)
  - **4-grams**: "The quick brown fox"

Language modeling

- The inference of the joint distribution is approximated via n-gram models that split the joint distribution into multiple independent parts.
- The term *n-grams* can be used to refer to other types of sequences of length *n*, such as *n* characters.
- If we have a huge corpus of text, we can find all the n-grams up until a certain *n* (typically 2 to 4) and count the occurrence of each n-gram in that corpus. From these counts, we can estimate the probabilities of the last word of each n-gram, given the previous *n-1* words:



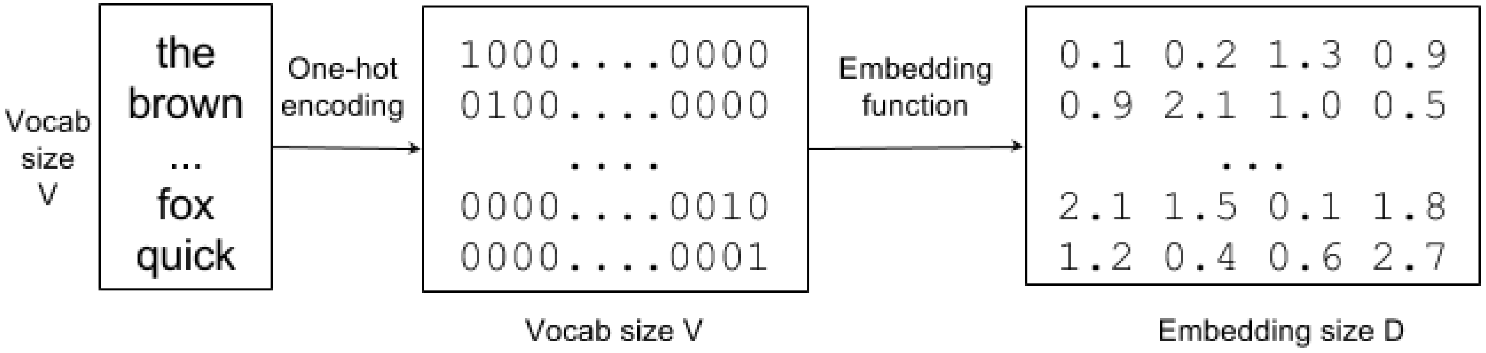
- 1-gram: $P(word) = \frac{count(word)}{\text{total number of words in corpus}}$
- 2-gram: $P(w_i | w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})}$
- N-gram: $P(w_{n+i} | w_n, \ldots, w_{n+i-1}) = \frac{count(w_n, \ldots, w_{n+i-1}, w_{n+i})}{count(w_n, \ldots, w_{n+i-1})}$

Language modeling

3. Neural language models

- One way to overcome the curse of dimensionality is by learning a lower dimensional, distributed representation of the words (http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf).
- This distributed representation is created by learning an embedding function that transforms the space of words into a lower dimensional space of word embeddings, as follows:

Words from the vocabulary with size *V* are transformed into one-hot encoding vectors of size *V* (each word is encoded uniquely). Then, the embedding function transforms this V-dimensional space into a distributed representation of size *D* (here, D=4). The idea is that the embedding function learns semantic information about the words. It associates each word in the vocabulary with a continuous-valued vector representation, that is, the word embedding.

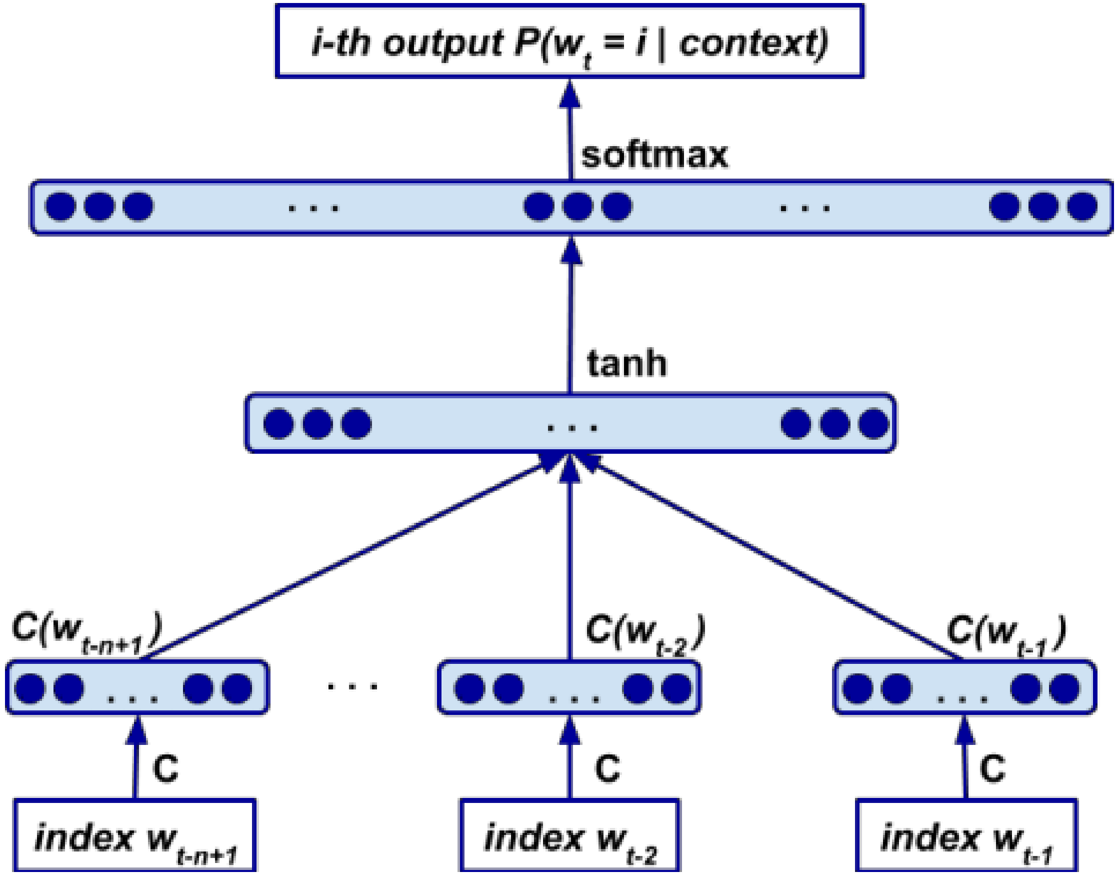


Words -> one-hot encoding -> word embedding vectors

Language modeling

4. Neural probabilistic language model

- It is possible to learn the language model and, implicitly, the embedding function via a feedforward fully-connected network.
- Given a sequence of *n-1* words (w_{t-n+1}, ..., w_{t-1}), it tries to output the probability distribution of the next word w_t
- The following diagram is based on http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf



Language modeling

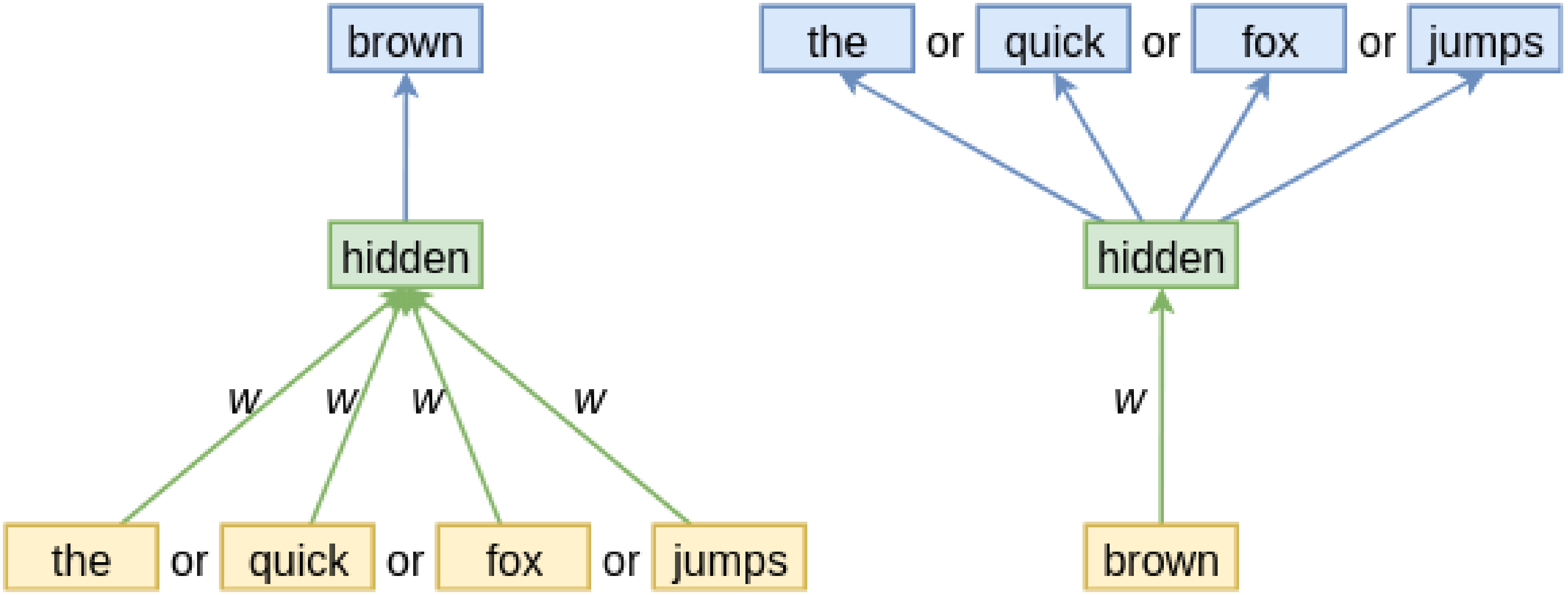
5. word2vec

- A lot of research has gone into creating better word embedding models, in particular by omitting learning the probability function over sequences of words.
- One of the most popular ways to do this is via word2vec (http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf and https://arxiv.org/pdf/1301.3781.pdf).

- To create embedding vectors with a word2vec model, we'll need a simple neural network that has the following:
  - It has an input, hidden, and an output layer
  - The input is a single one-hot encoded word representation
  - The output is a single softmax, which predicts the most likely word to be found in the context (proximity) of the input word

Language modeling

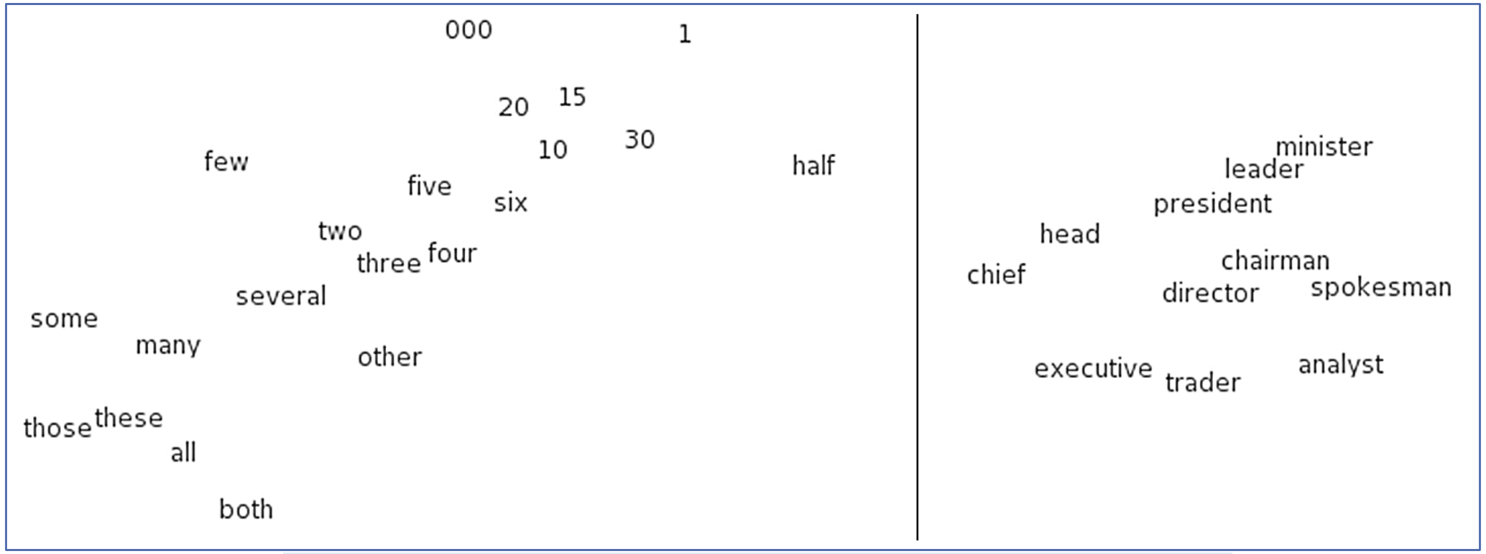
- Depending on the way we train the model, we have two flavors:

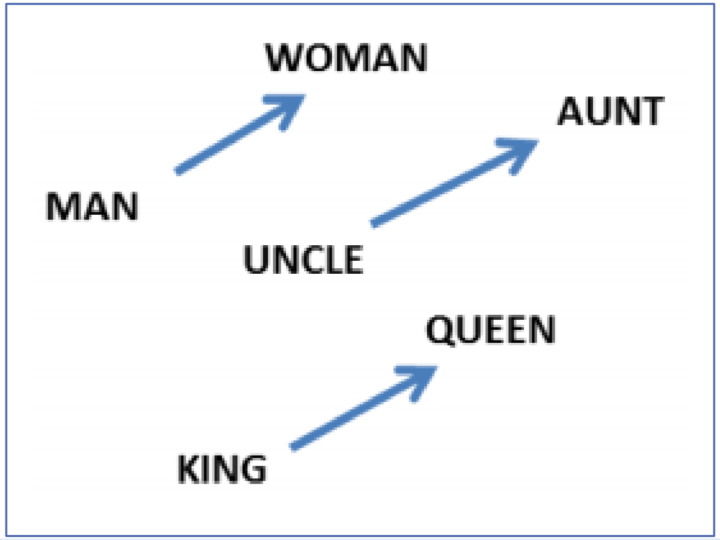


CBOW (left) and skip-gram (right)

Visualizing word embedding vectors

- In the following diagram, we can see a 2D projection of some word embeddings.
- The words, which are semantically close, are also close to each other in the embedding space.
- Word embeddings can capture semantic differences between words.





Related words in a 2D embedding space are close to each other in this space

http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/

Word embeddings can capture semantic differences between words

https://www.aclweb.org/anthology/N/N13/N13-1090.pdf

Sequence to sequence learning

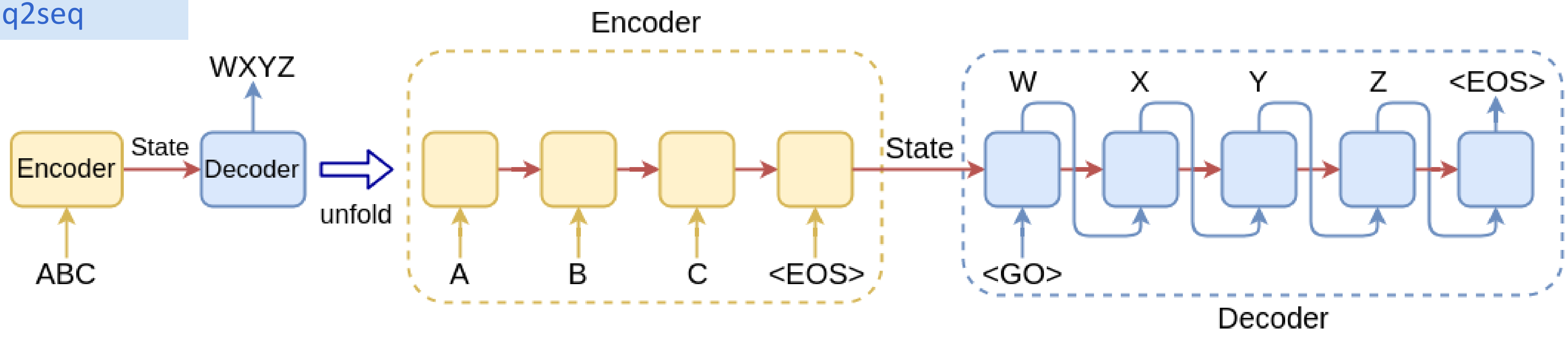
- Many, many NLP problems can be formulated as sequence to sequence tasks. This is a type of task where an input sequence is transformed into another, different output sequence, not necessarily with the same length as the input. To better understand this concept, let's look at some examples:

  - Machine translation is the most popular type of seq2seq task. The input sequences are the words of a sentence in one language and the output sequences are the words of the same sentence, translated into another language. For example, we can translate the English sequence "Tourist attraction" to the German "Touristenattraktion." Not only is the output sentence a different length – there is no direct correspondence between the elements of the input and output sequences. In particular, one output element corresponds to a combination of two input elements. Machine translation that's implemented with a single neural network is called **neural machine translation (NMT)**.
  - Speech recognition, where we take different time frames of an audio input and convert them into text transcript.
  - Question answering chatbots, where the input sequences are the words of a textual question and the output sequence is the answer to that question.
  - Text summarization, where the input is a text document and the output is a short summary of the text's contents.

Sequence to sequence learning

- In 2014, Sutskever et al. (https://arxiv.org/abs/1409.3215) and Cho et al. (https://arxiv.org/abs/1406.1078) introduced a method called sequence to sequence (seq2seq, or encoder-decoder) learning.
- It's an example of an "indirect many-to-many" relationship.
- A seq2seq model consists of two parts:
  - The encoder is an RNN. The original paper uses LSTM, but GRU or other types would also work.
  - The decoder generate the output sequence. The encoder is also a RNN (LSTM or GRU). The link between the encoder and the decoder is the most recent encoder state vector h_t.

Seq2seq

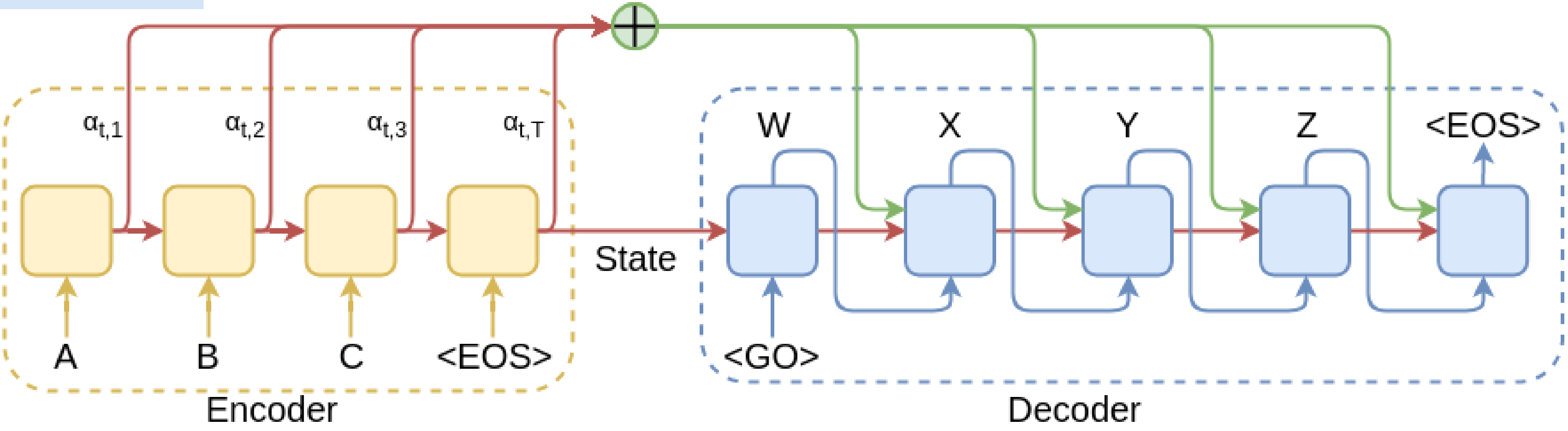


Sequence to sequence learning

Sequence to sequence with attention

- Bahdanau et al. (https://arxiv.org/abs/1409.0473) proposed a seq2seq extension called an attention mechanism, which provides a way for the decoder to work with all encoder hidden states, and not just the last one.

Seq2seq with attention



Speech Recognition

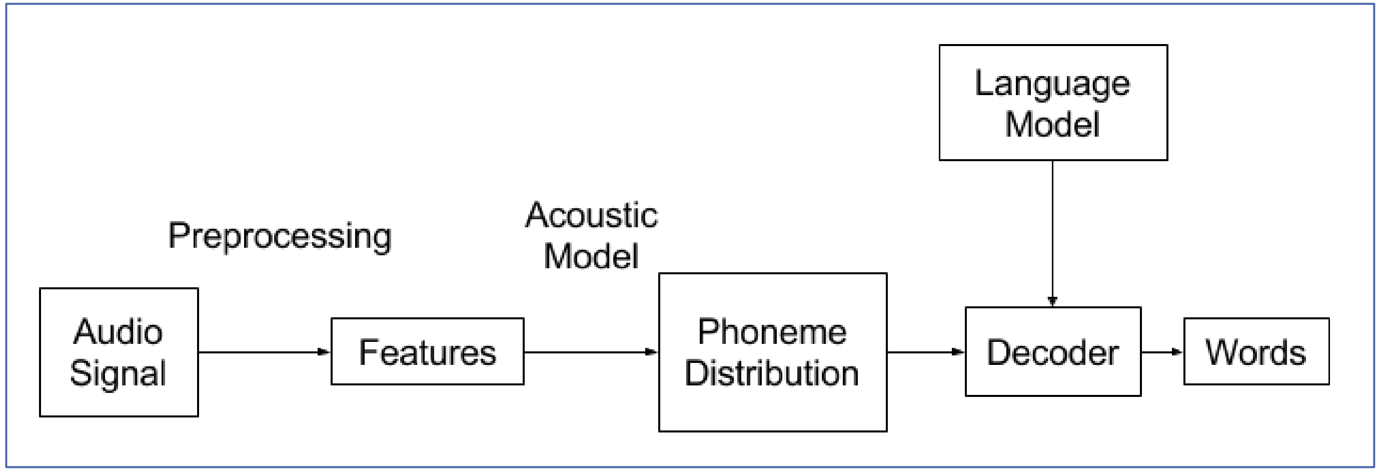
- Speech recognition tries to find a transcription of the most probable word sequence considering the acoustic observations provided:

transcription = argmax(P(words | audio features))

- This probability function is typically modeled in different parts (note that the normalizing term P (audio features) is usually ignored):

*P (words | audio features) = P (audio features | words) * P (words)*

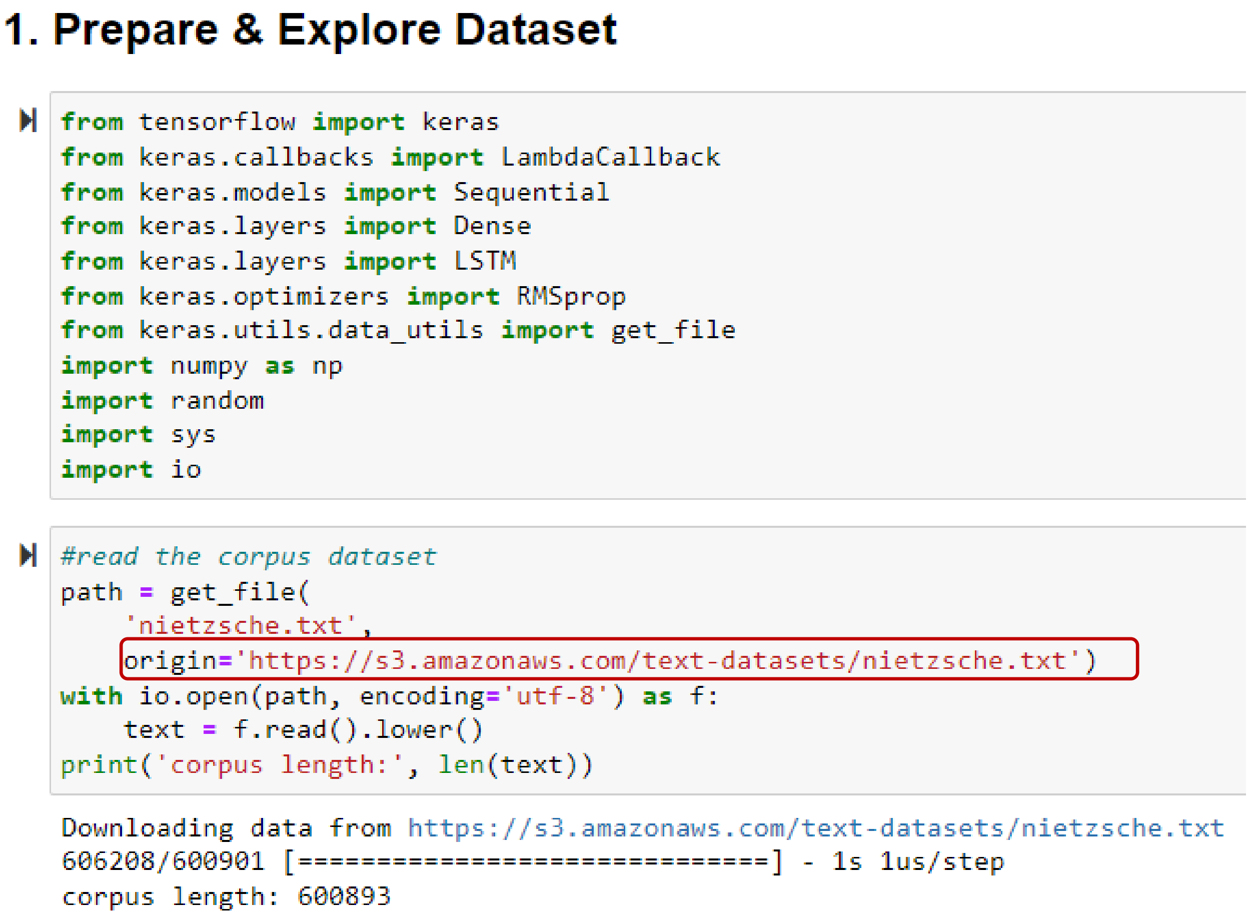
*= P (audio features | phonemes) * P (phonemes | words) * P (words)*



Speech recognition pipeline

Let’s have fun ☺

LSTM application to generate text from Nietzsche's writings

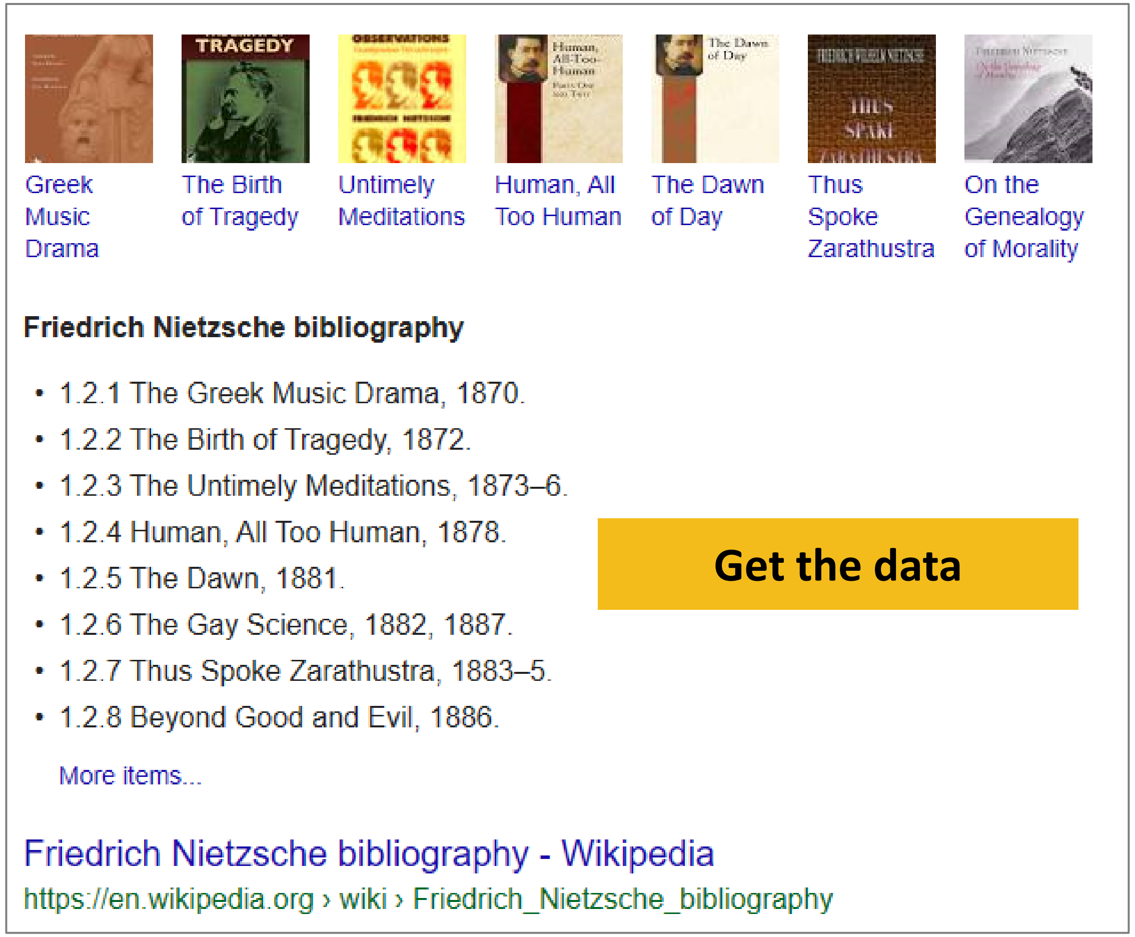


1. Prepare & Explore Dataset

```
from tensorflow import keras
from keras.callbacks import LambdaCallback
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.optimizers import RMSprop
from keras.utils.data_utils import get_file
import numpy as np
import random
import sys
import io
```

```
#read the corpus dataset
path = get_file(
    'nietzsche.txt',
    origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')
with io.open(path, encoding='utf-8') as f:
    text = f.read().lower()
print('corpus length:', len(text))
```

```
Downloading data from https://s3.amazonaws.com/text-datasets/nietzsche.txt
606208/600901 [==============================] - 1s 1us/step
corpus length: 600893
```



Greek Music Drama

The Birth of Tragedy

Untimely Meditations

Human, All Too Human

The Dawn of Day

Thus Spoke Zarathustra

On the Genealogy of Morality

Friedrich Nietzsche bibliography

- 1.2.1 The Greek Music Drama, 1870.
- 1.2.2 The Birth of Tragedy, 1872.
- 1.2.3 The Untimely Meditations, 1873–6.
- 1.2.4 Human, All Too Human, 1878.
- 1.2.5 The Dawn, 1881.
- 1.2.6 The Gay Science, 1882, 1887.
- 1.2.7 Thus Spoke Zarathustra, 1883–5.
- 1.2.8 Beyond Good and Evil, 1886.

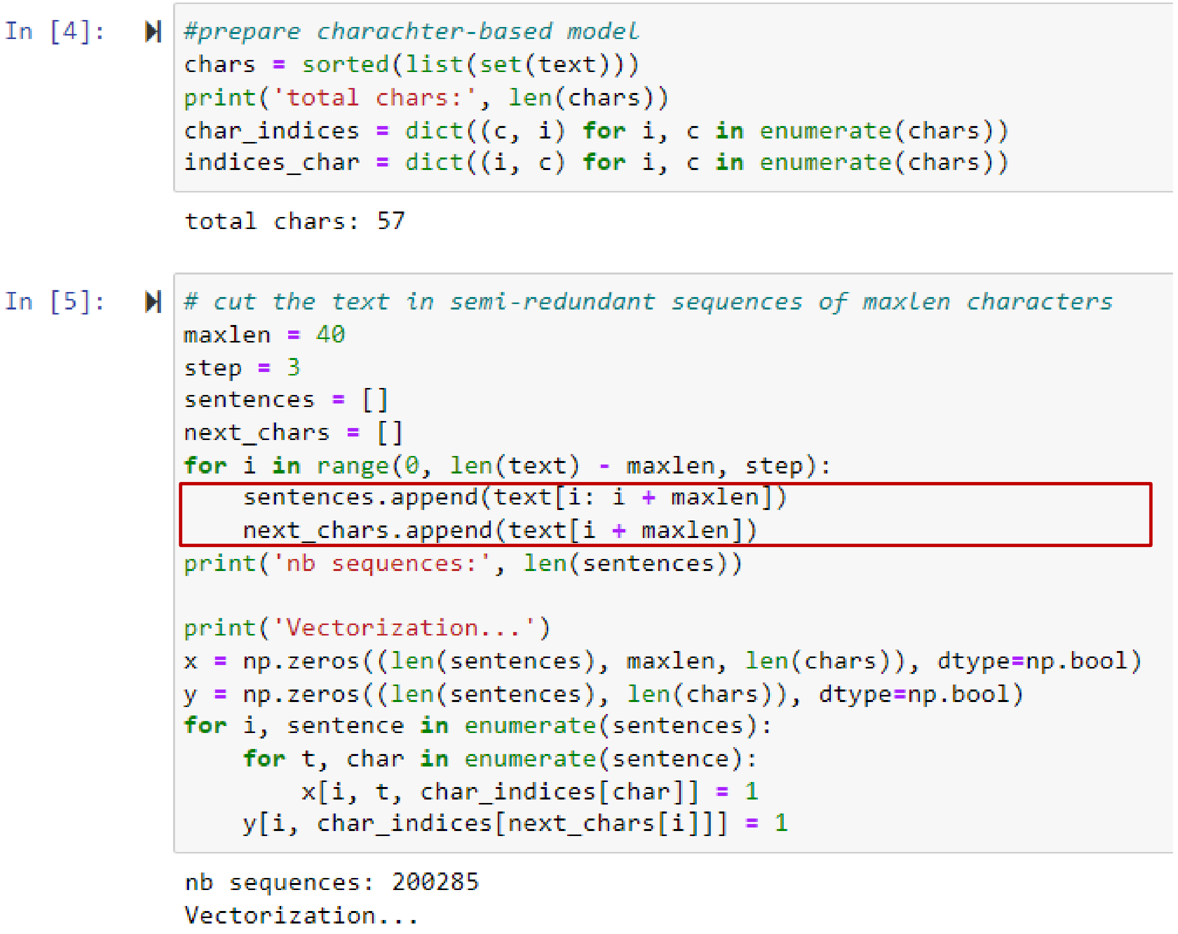
More items...

Friedrich Nietzsche bibliography - Wikipedia

https://en.wikipedia.org › wiki › Friedrich_Nietzsche_bibliography

Get the data

LSTM application to generate text from Nietzsche's writings



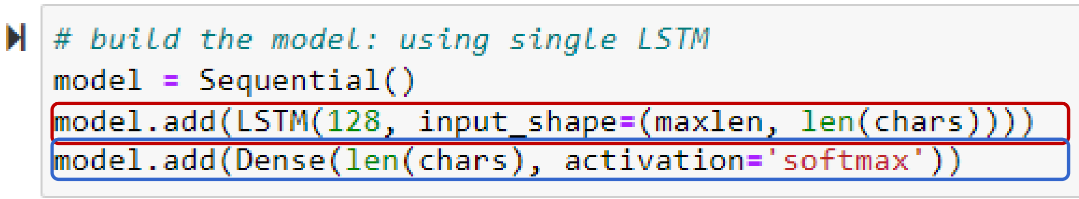
```
In [4]:  #prepare charachter-based model
         chars = sorted(list(set(text)))
         print('total chars:', len(chars))
         char_indices = dict((c, i) for i, c in enumerate(chars))
         indices_char = dict((i, c) for i, c in enumerate(chars))

         total chars: 57
```

```
In [5]:  # cut the text in semi-redundant sequences of maxlen characters
         maxlen = 40
         step = 3
         sentences = []
         next_chars = []
         for i in range(0, len(text) - maxlen, step):
             sentences.append(text[i: i + maxlen])
             next_chars.append(text[i + maxlen])
         print('nb sequences:', len(sentences))

         print('Vectorization...')
         x = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool)
         y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
         for i, sentence in enumerate(sentences):
             for t, char in enumerate(sentence):
                 x[i, t, char_indices[char]] = 1
             y[i, char_indices[next_chars[i]]] = 1

         nb sequences: 200285
         Vectorization...
```

Build the language model

LSTM application to generate text from Nietzsche's writings

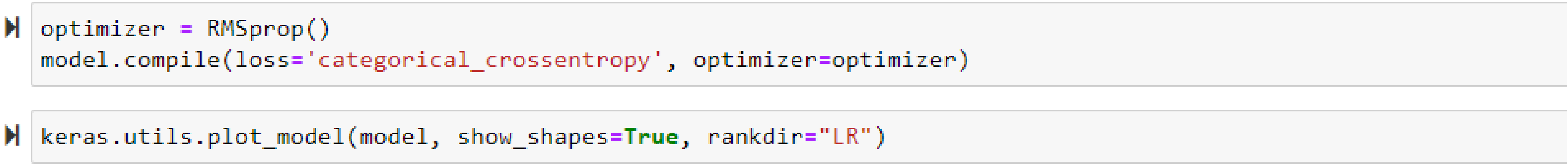
2. Define the neural network architecture



```
# build the model: using single LSTM
model = Sequential()
model.add(LSTM(128, input_shape=(maxlen, len(chars))))
model.add(Dense(len(chars), activation='softmax'))
```

- We feed the input into the first LSTM unit.
- Then, we use a softmax activation function to calculate the probability of the output words.

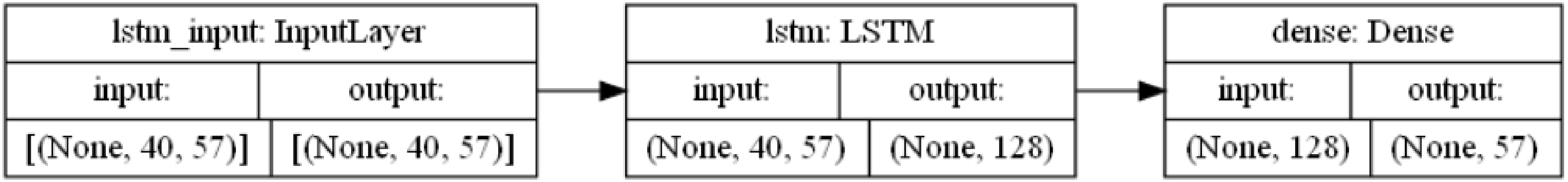
3. Compile the neural net



```
optimizer = RMSprop()
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

```
keras.utils.plot_model(model, show_shapes=True, rankdir="LR")
```

8]:

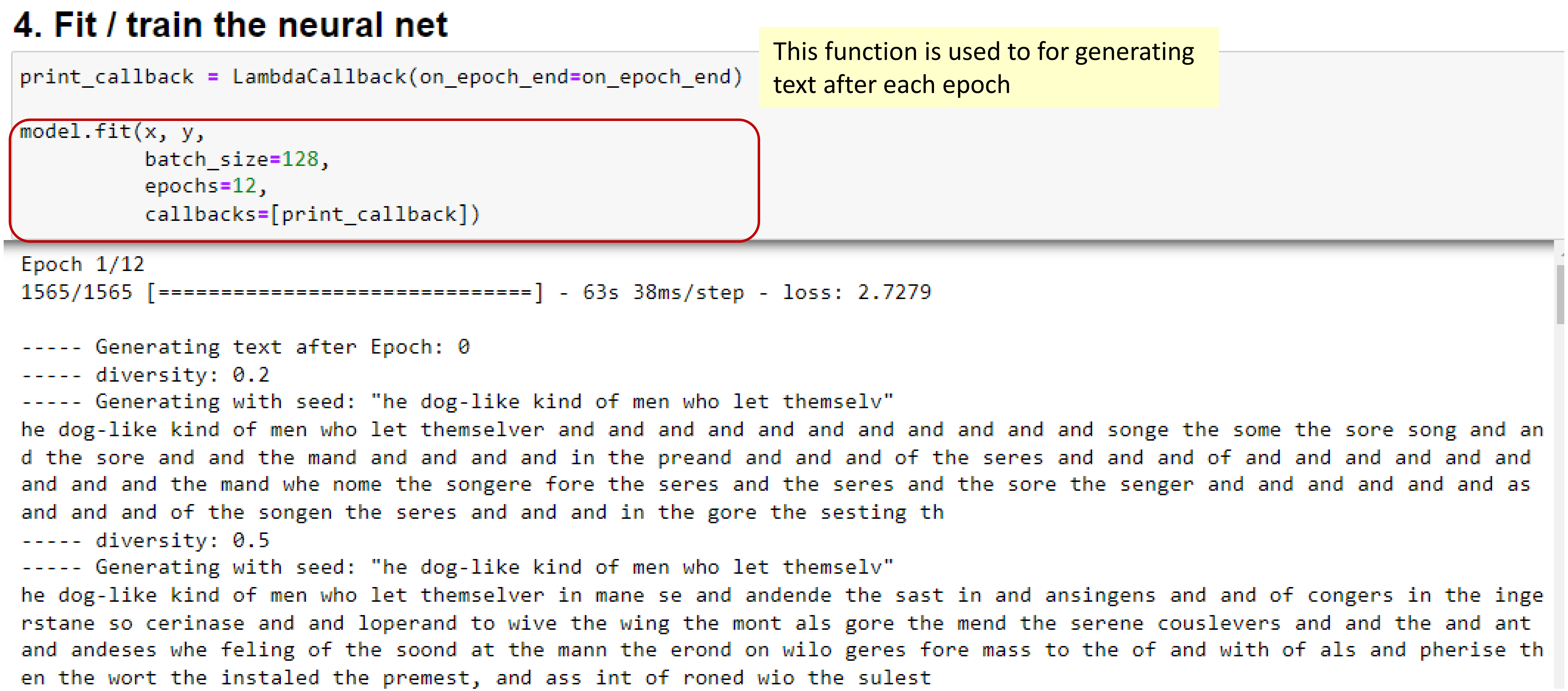


lstm_input: InputLayer | input: [(None, 40, 57)] | output: [(None, 40, 57)]

lstm: LSTM | input: (None, 40, 57) | output: (None, 128)

dense: Dense | input: (None, 128) | output: (None, 57)

LSTM application to generate text from Nietzsche's writings



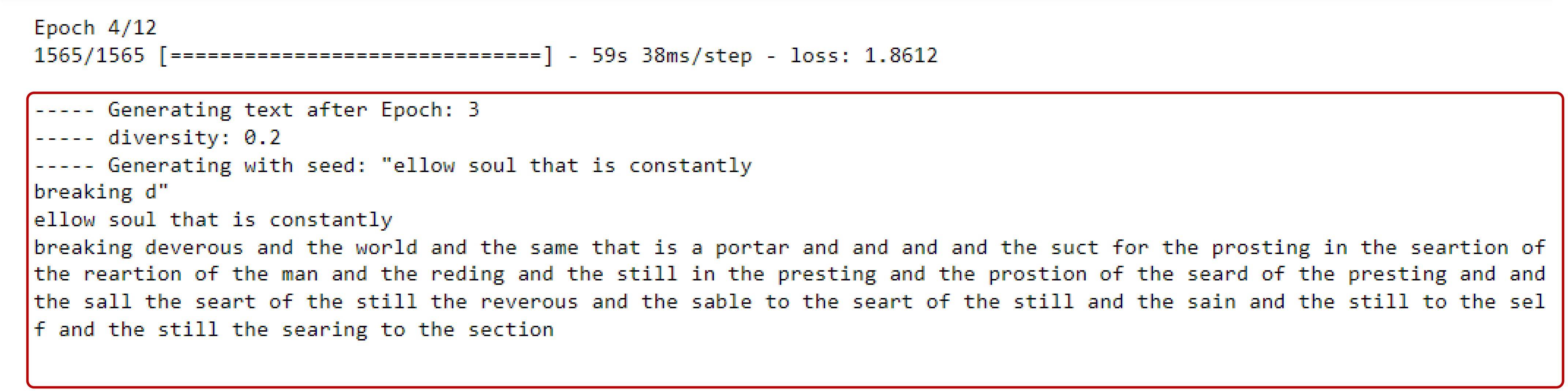
4. Fit / train the neural net

```
print_callback = LambdaCallback(on_epoch_end=on_epoch_end)

model.fit(x, y,
          batch_size=128,
          epochs=12,
          callbacks=[print_callback])
```

This function is used to for generating text after each epoch

```
Epoch 1/12
1565/1565 [==============================] - 63s 38ms/step - loss: 2.7279

----- Generating text after Epoch: 0
----- diversity: 0.2
----- Generating with seed: "he dog-like kind of men who let themselv"
he dog-like kind of men who let themselver and and and and and and and and and and and songe the some the sore song and an
d the sore and and the mand and and and and in the preand and and and of the seres and and and of and and and and and and
and and and the mand whe nome the songere fore the seres and the seres and the sore the senger and and and and and and as
and and and of the songen the seres and and and in the gore the sesting th
----- diversity: 0.5
----- Generating with seed: "he dog-like kind of men who let themselv"
he dog-like kind of men who let themselver in mane se and andende the sast in and ansingens and and of congers in the inge
rstane so cerinase and and loperand to wive the wing the mont als gore the mend the serene couslevers and and the and ant
and andeses whe feling of the soond at the mann the erond on wilo geres fore mass to the of and with of als and pherise th
en the wort the instaled the premest, and ass int of roned wio the sulest
```

LSTM application to generate text from Nietzsche's writings



```
Epoch 4/12
1565/1565 [==============================] - 59s 38ms/step - loss: 1.8612

----- Generating text after Epoch: 3
----- diversity: 0.2
----- Generating with seed: "ellow soul that is constantly
breaking d"
ellow soul that is constantly
breaking deverous and the world and the same that is a portar and and and and the suct for the prosting in the seartion of
the reartion of the man and the reding and the still in the presting and the prostion of the seard of the presting and and
the sall the seart of the still the reverous and the sable to the seart of the still and the sain and the still to the sel
f and the still the searing to the section
```

- At least 20-60 epochs are required before the generated text starts sounding coherent.
- It is recommended to run this script on GPU, as recurrent networks are quite computationally intensive.

Sentiment classification using pre-trained language models (transformers)

🤗 Transformers

State-of-the-art Machine Learning which provide APIs and tools to easily download and train state-of-the-art pretrained models. Using pretrained models can reduce your compute costs, carbon footprint, and save you the time and resources required to train a model from scratch. These models support common tasks :

- 📝 **Natural Language Processing**: text classification, named entity recognition, question answering, language modeling, summarization, translation, multiple choice, and text generation.
- 🖼️ **Computer Vision**: image classification, object detection, and segmentation.
- 🗣️ **Audio**: automatic speech recognition and audio classification.
- 🐙 **Multimodal**: table question answering, optical character recognition, information extraction from scanned documents, video classification, and visual question answering.

https://huggingface.co/

Sentiment classification using pre-trained language models (transformers)

Supported models

1. ALBERT (from Google Research and the Toyota Technological Institute at Chicago) released with the paper ALBERT: A Lite BERT for Self-supervised Learning of Language Representations, by Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, Radu Soricut.
2. BART (from Facebook) released with the paper BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension by Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov and Luke Zettlemoyer.
3. BARThez (from École polytechnique) released with the paper BARThez: a Skilled Pretrained French Sequence-to-Sequence Model by Moussa Kamal Eddine, Antoine J.-P. Tixier, Michalis Vazirgiannis.
4. BARTpho (from VinAI Research) released with the paper BARTpho: Pre-trained Sequence-to-Sequence Models for Vietnamese by Nguyen Luong Tran, Duong Minh Le and Dat Quoc Nguyen.
5. BEiT (from Microsoft) released with the paper BEiT: BERT Pre-Training of Image Transformers by Hangbo Bao, Li Dong, Furu Wei.
6. BERT (from Google) released with the paper BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding by Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova.
7. BERT For Sequence Generation (from Google) released with the paper Leveraging Pre-trained Checkpoints for Sequence Generation Tasks by Sascha Rothe, Shashi Narayan, Aliaksei Severyn.
8. BERTweet (from VinAI Research) released with the paper BERTweet: A pre-trained language model for English Tweets by Dat Quoc Nguyen, Thanh Vu and Anh Tuan Nguyen.

Sentiment classification using pre-trained language models (transformers)

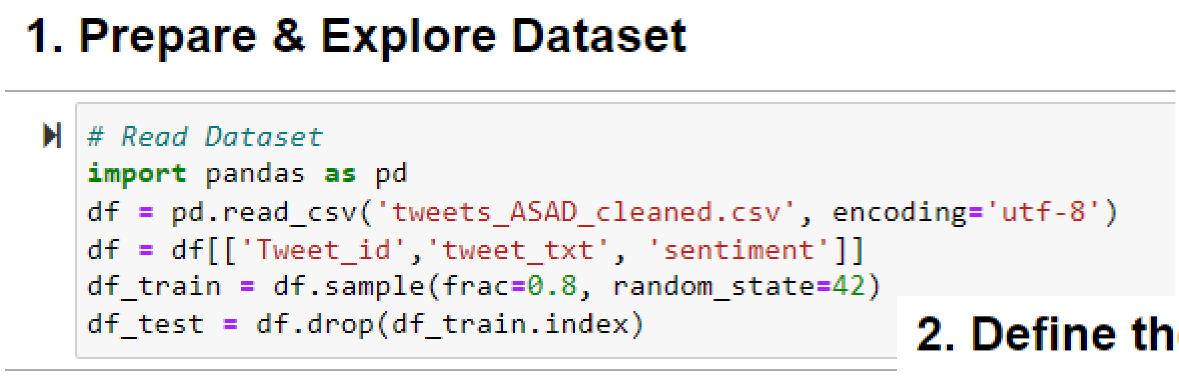
جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

University
Library

ASAD: A Twitter-based Benchmark Arabic Sentiment Analysis Dataset

https://repository.kaust.edu.sa/handle/10754/665873?show=full

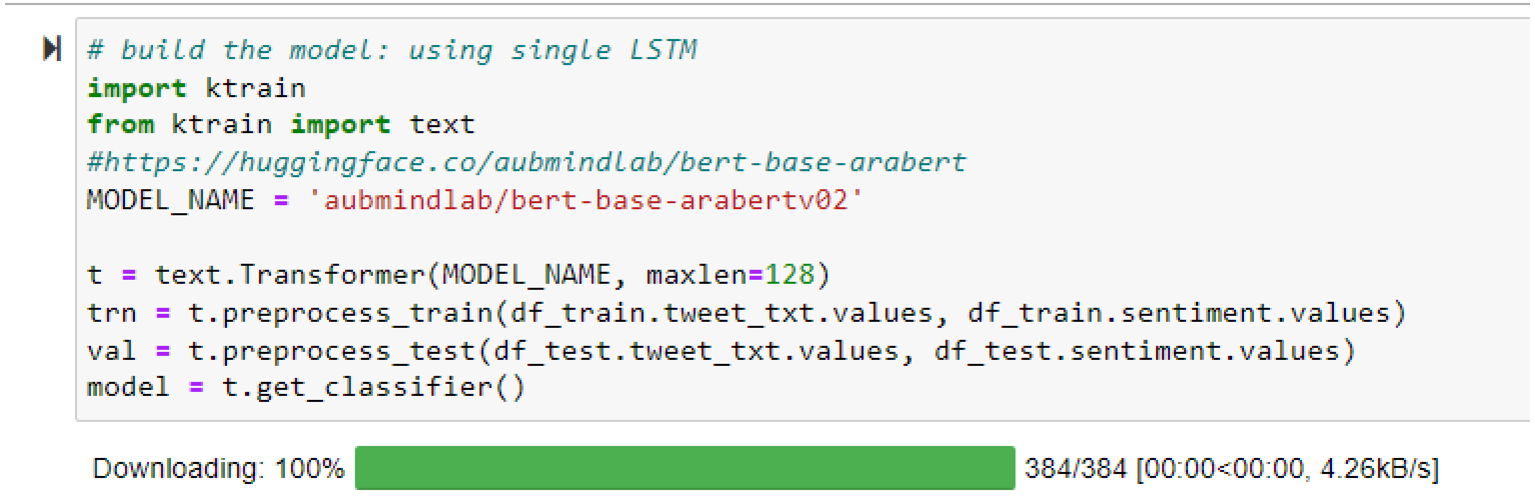
Sentiment classification using pre-trained language models (transformers)



1. Prepare & Explore Dataset

```
# Read Dataset
import pandas as pd
df = pd.read_csv('tweets_ASAD_cleaned.csv', encoding='utf-8')
df = df[['Tweet_id','tweet_txt', 'sentiment']]
df_train = df.sample(frac=0.8, random_state=42)
df_test = df.drop(df_train.index)
```

2. Define the neural network architecture



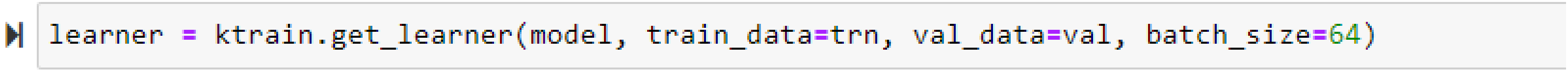
```
# build the model: using single LSTM
import ktrain
from ktrain import text
#https://huggingface.co/aubmindlab/bert-base-arabert
MODEL_NAME = 'aubmindlab/bert-base-arabertv02'

t = text.Transformer(MODEL_NAME, maxlen=128)
trn = t.preprocess_train(df_train.tweet_txt.values, df_train.sentiment.values)
val = t.preprocess_test(df_test.tweet_txt.values, df_test.sentiment.values)
model = t.get_classifier()
```

```
Downloading: 100% 384/384 [00:00<00:00, 4.26kB/s]
```

Module (6) Deep_Learning_Module_6_SentimentClassification.ipynb

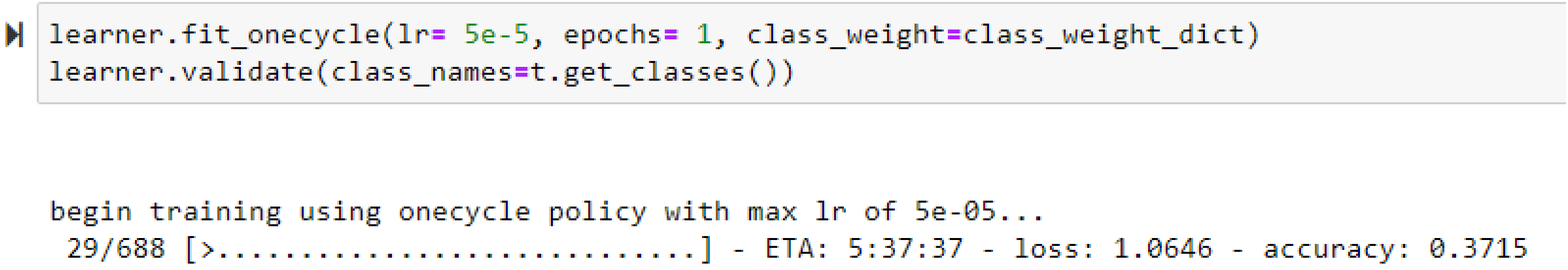
Sentiment classification using pre-trained language models (transformers)

3. Compile the neural net



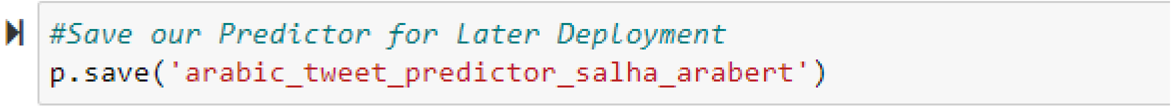
```
learner = ktrain.get_learner(model, train_data=trn, val_data=val, batch_size=64)
```

4. Fit / train the neural net



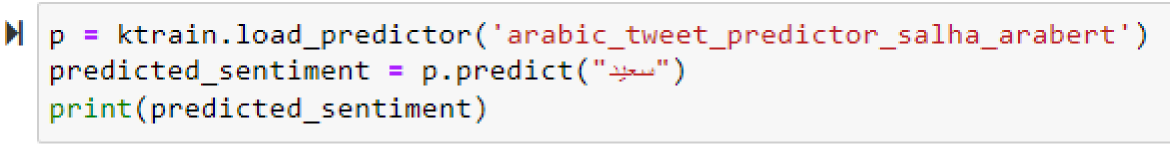
```
learner.fit_onecycle(lr= 5e-5, epochs= 1, class_weight=class_weight_dict)
learner.validate(class_names=t.get_classes())
```

```
begin training using onecycle policy with max lr of 5e-05...
 29/688 [>.............................] - ETA: 5:37:37 - loss: 1.0646 - accuracy: 0.3715
```



```
#Save our Predictor for Later Deployment
p.save('arabic_tweet_predictor_salha_arabert')
```

5. Test our predictor



```
p = ktrain.load_predictor('arabic_tweet_predictor_salha_arabert')
predicted_sentiment = p.predict("سعيد")
print(predicted_sentiment)
```

My Notes

Recap!

- ☑ Sequential data
- ☑ Recurrent neural network (RNN)
- ☑ Long short-term memory (LSTM)
- ☑ Gated recurrent unit (GRU)
- ☑ Language modeling
- ☑ Sequence to sequence learning (Seq2Seq)
- ☑ Speech recognition
- ☑ Example: Text generation with LSTM
- ☑ Example: Sentiment classification using pre-trained language models (transformers)

Success is liking yourself, liking what you do, and liking how you do it.

Maya Angelou

@SalhaAlzahrani