Natural Language Processing with Recurrent Neural Networks

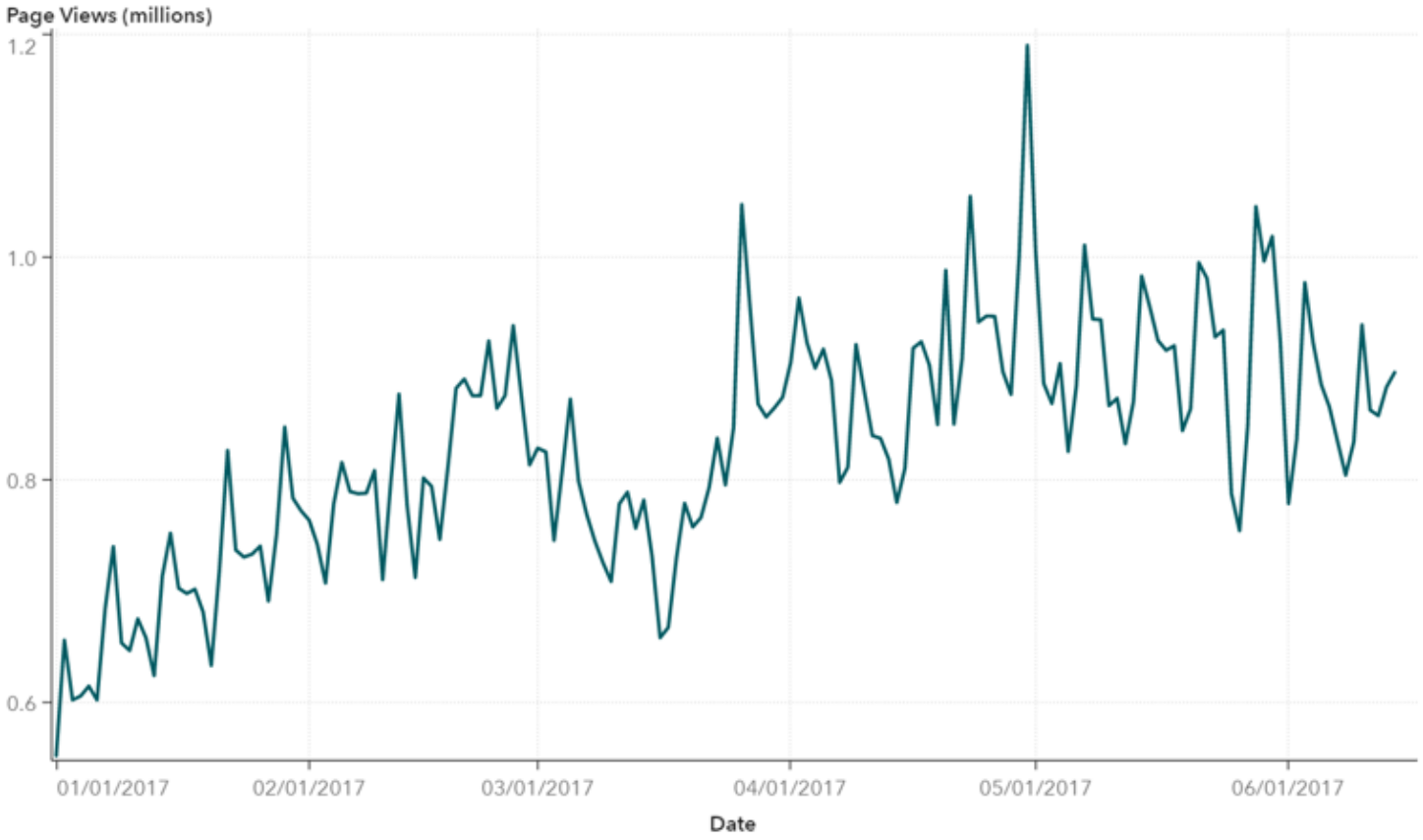
The structure of the lecture:

1. Why RNNs & NLP matter
2. RNNs: An introduction
3. RNNs Under the Hood (and architectural variations)
4. NLP: Why we use RNNs
5. Classifying sentiment: A coded example
6. Further Reading

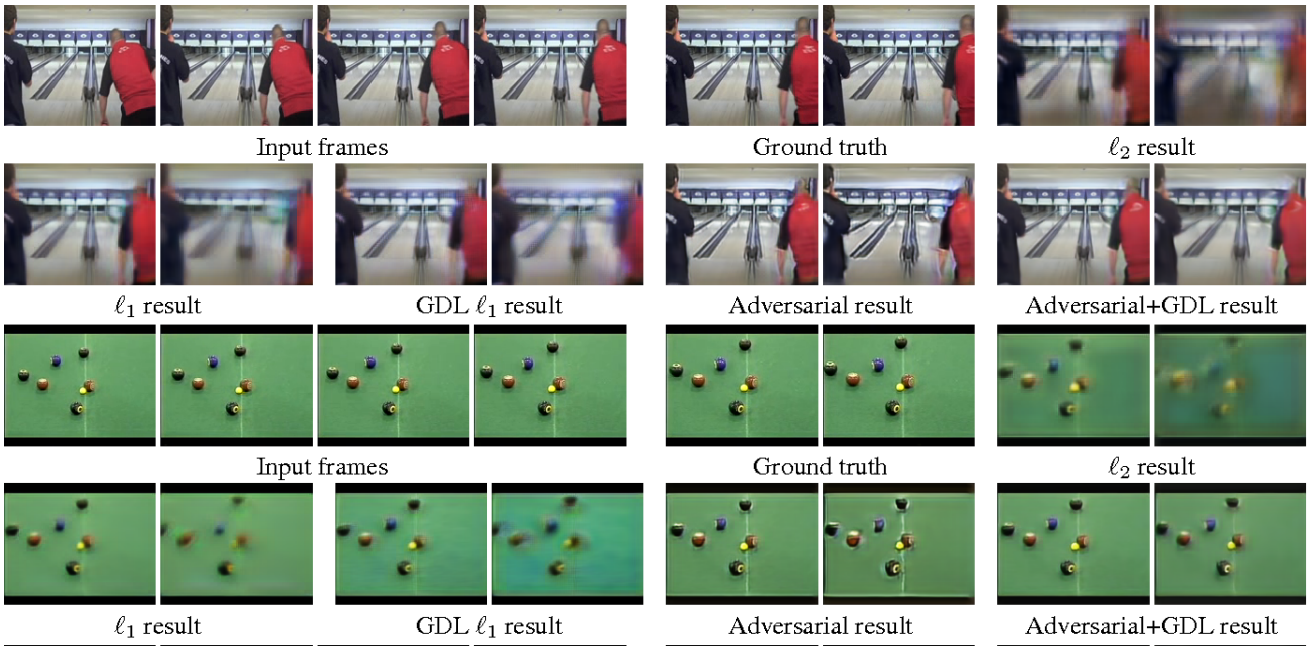
1️⃣ Why RNNs & NLP matter

📈 Recurrent Neural Networks are Neural Networks specifically designed to deal with **sequences** as input data, i.e. **observations repeated throughout time**.

Example 1: Prediction of future stock market values and trends



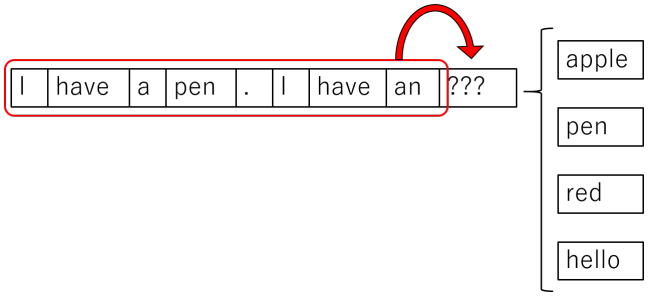
Example 2: Video prediction



🎥 Videos = sequences of images/frames

👉 Why not predicting the next image(s)?

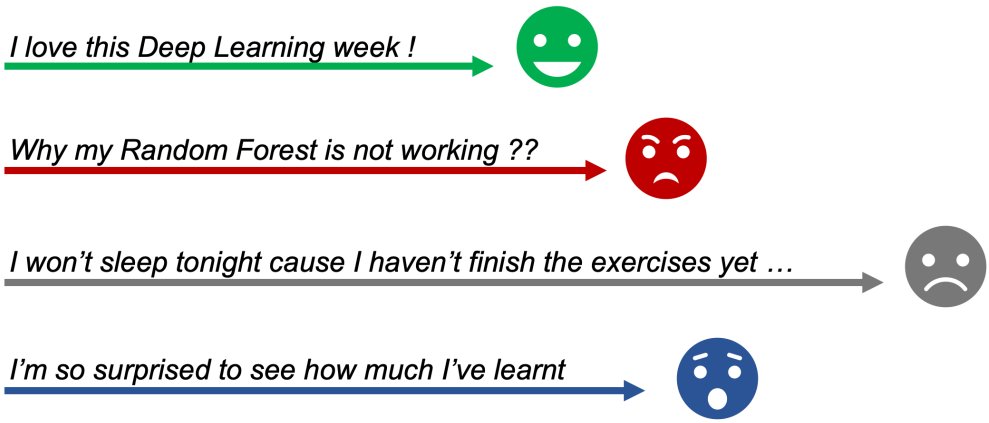
Example 3: Predicting the next word - Natural Language Processing



🗣️ **Recurrent Networks are massively used for text!**

NLP: Text classification, such as sentiment analysis

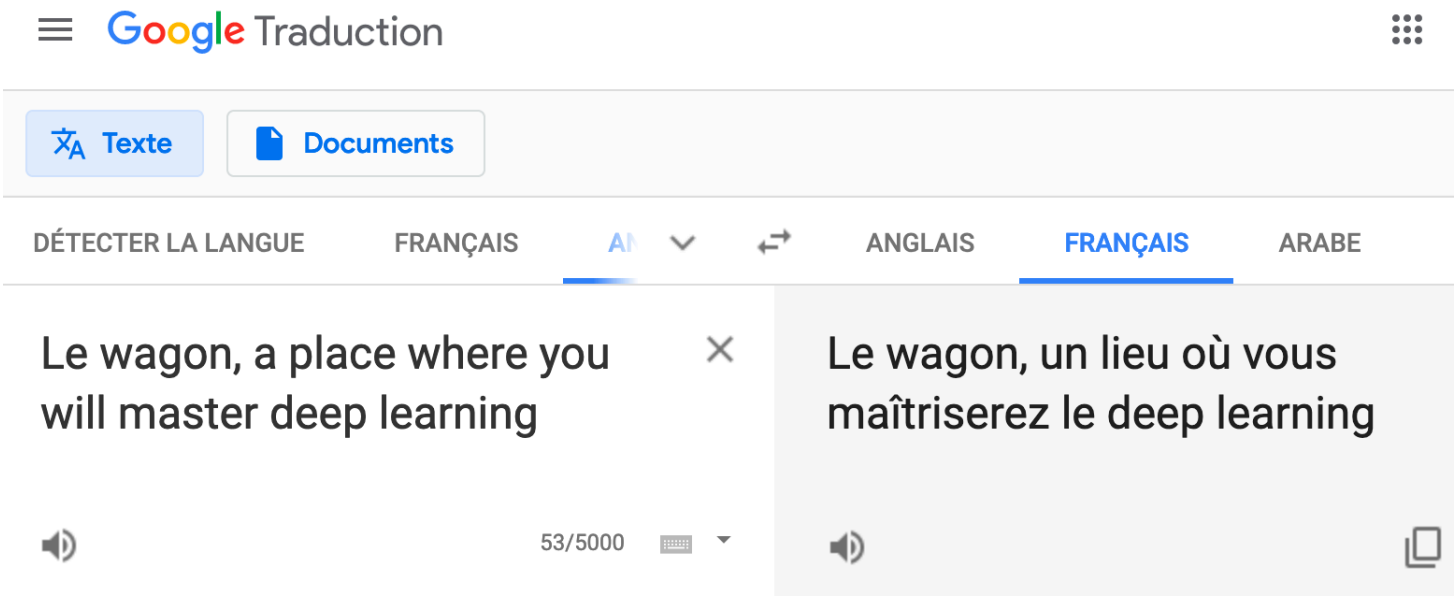
Classification depending on a word, a sentence, a paragraph, ...



The typical setting is **sentiment analysis**: Classify positive or negative sentences (but also happiness, sadness, joy, anger, ...).

Sequence to sequence models

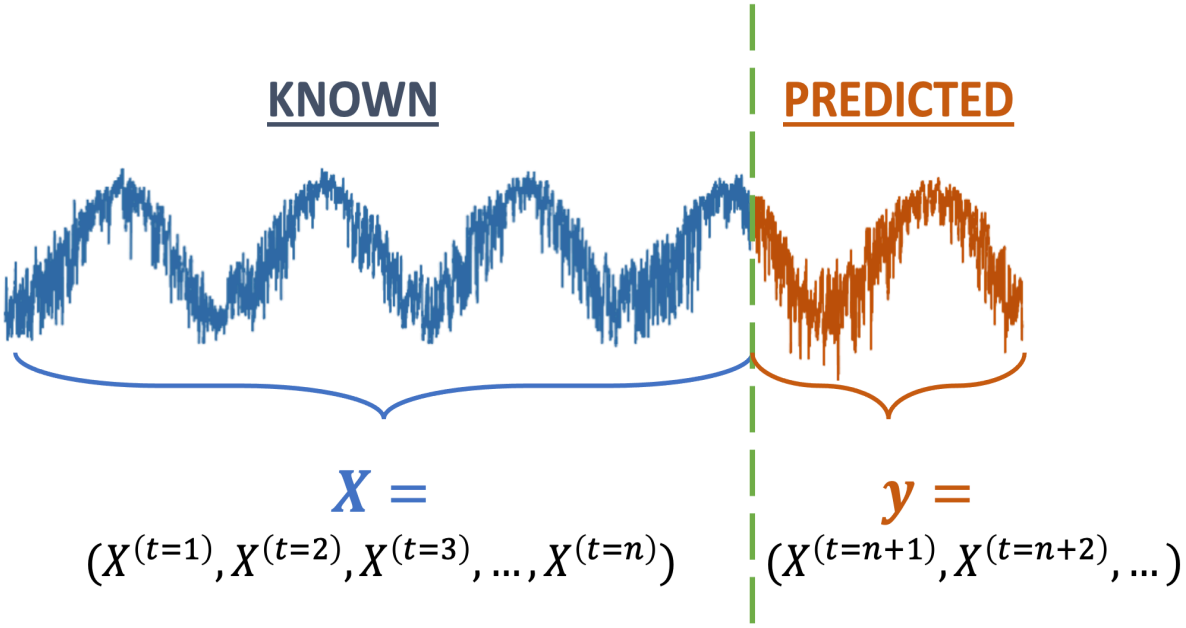
Given an input sequence, produce a corresponding output sequence. Typical application is language translation.



2️⃣ RNNs: An introduction

Inputs & Outputs:

During the Machine Learning lectures, you discovered Time Series as follows:



The algorithm would learn from the past values to predict the future values using temporal features such as the trend, the seasonality...

👉 *Example*: **ARIMA** models **recursively** predict the next data points **one after the other**

What follows is a slighly different approach...

General DL Framework

- ***Training:*** Many pairs of (X_i , y_i) = (observation, target) are used to learn the patterns between the input and the output.

Input		Target
X_1		y_1
X_2		y_2
X_3	Output $y = NN(\text{Input } X)$	y_3
...		...
X_n		y_n

- ***Prediction***: New input X_{new} $\rightarrow$ New prediction/output y_{pred}

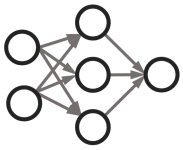
❗ Each individual observation X_i can be of any type: scalar, vector, matrix (an image), ...

👉 `X.shape` = (n_ROWS, n_FEATURES)

What are our inputs for RNNS?

- ***Training***: Each input
  X_i
  consists of a **sequence of repeated observations** through time:

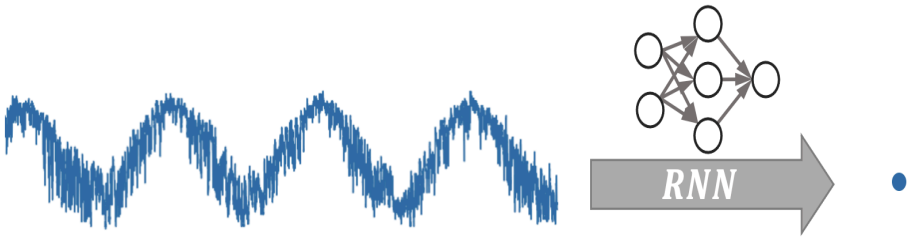
$$X_i = (X_i^{t=1}, X_i^{t=2}, X_i^{t=3}, \ldots, X_i^{t=m})$$

Input		Target
$X_1 = (X_1^{t=1}, X_1^{t=2}, X_1^{t=3}, \ldots)$		y_1
$X_2 = (X_2^{t=1}, X_2^{t=2}, X_2^{t=3}, \ldots)$		y_2
X_3	Output y = *RNN*(Input X)	y_3
...		...
X_n		y_n

👉 `X.shape` = (n_SEQUENCES, n_OBSERVATIONS, n_FEATURES)

❗ The input tensor
X
has a new dimension, the **temporal dimension** which corresponds to the **number of observations through time.**

🔮 Prediction: The next observation

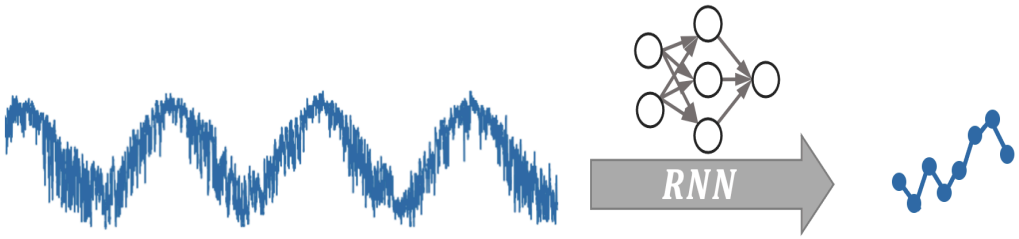


👉 Predict `rain` on `day + 1`

- `X.shape` = (None, 100 days, 1 feature)
- `y.shape` = (None, 1 next day, 1 feature)

ℹ️ None is a placeholder for the number of sequences given, it is equal to batch_size

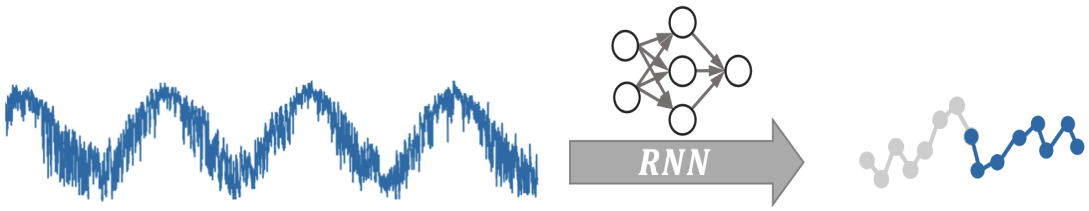
🔮 Prediction: The next observation**S**



👉 Predict `rain` on `[day + 1 ... day + 8]`

- `X.shape` = (None, 100 days, 1 feature)
- `y.shape` = (None, 8 next days, 1 feature)

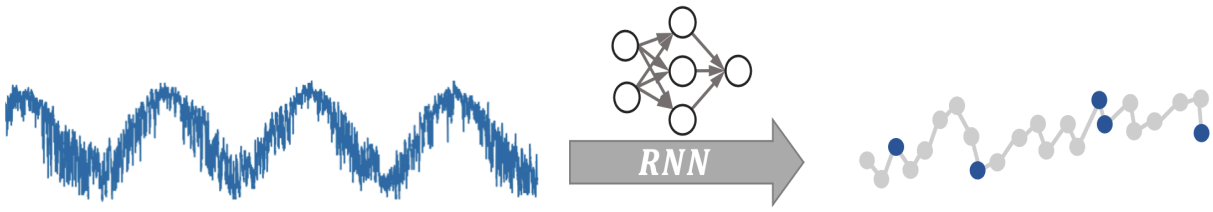
🔮 **Prediction: The next observation<u>S</u>, but not necessarily right after the last seen value**



👉 Predict `rain` on `[day + 8 ... day + 16]`

- `X.shape` = (None, 100 days, 1 feature)
- `y.shape` = (None, 8 consecutive days in the future, 1 feature)

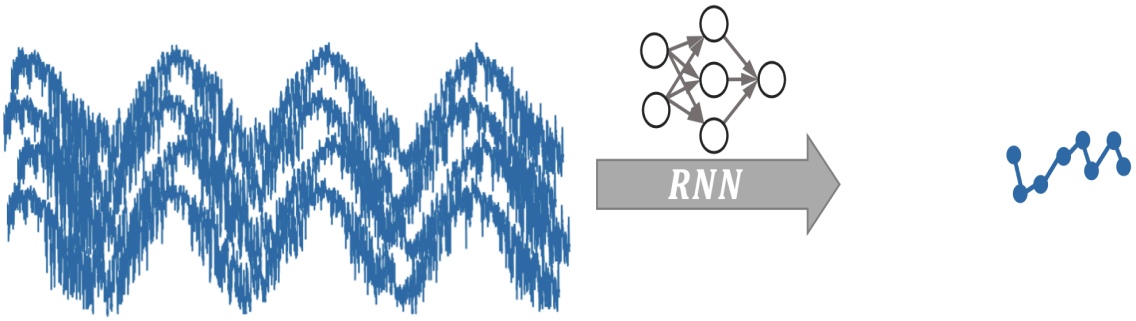
🔮 **Prediction: The next observation<u>S</u>, but not necessarily consecutive**



👉 Predict `rain` for `5 specific days in the future`

- `X.shape` = (None, 100 days, 1 feature)
- `y.shape` = (None, 5 specific days in the future, 1 feature)

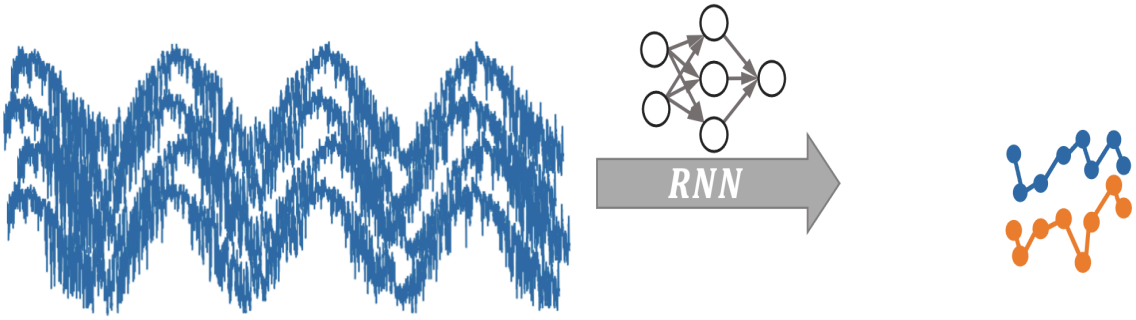
🔮 Prediction with multivariate inputs



👉 Predict `rain` for the `8 next days` based on `multiple features [rain, humidity, temperature, pressure...]`

- `X.shape` = (None, 100 days of observation, 4 features)
- `y.shape` = (None, 8 next days, 1 feature = rain)

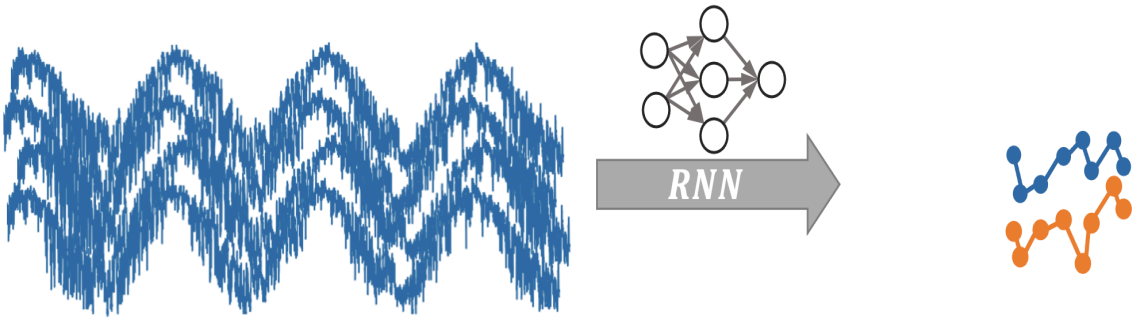
🔮 Predicting multiple outputs



👉 Predict `[rain, temperature]` for the `next 8 days` based on `past [rain, humidity, temperature, pressure...] features`

- `X.shape` = (None, 100 days of observation, 4 features)
- `y.shape` = (None, 8 next days, 2 features = [rain, temperature])

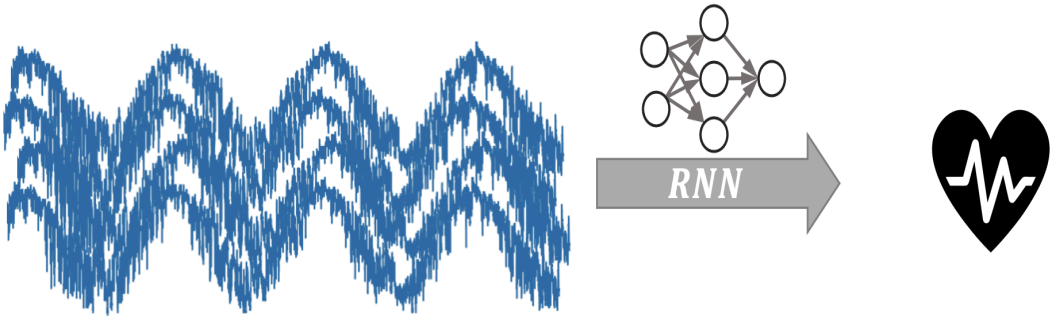
❗ **The output is not necessarily similar to the past observations of X!**



👉 Predict `[avalanches, accidents]` for the `next 8 days` based on past `[rain, temperature, pressure...]` features

- `X.shape` = (None, 100 days of observation, 4 features = [rain,temperature,pressure])
- `y.shape` = (None, 8 next days, 2 features = [avalanche, accidents])

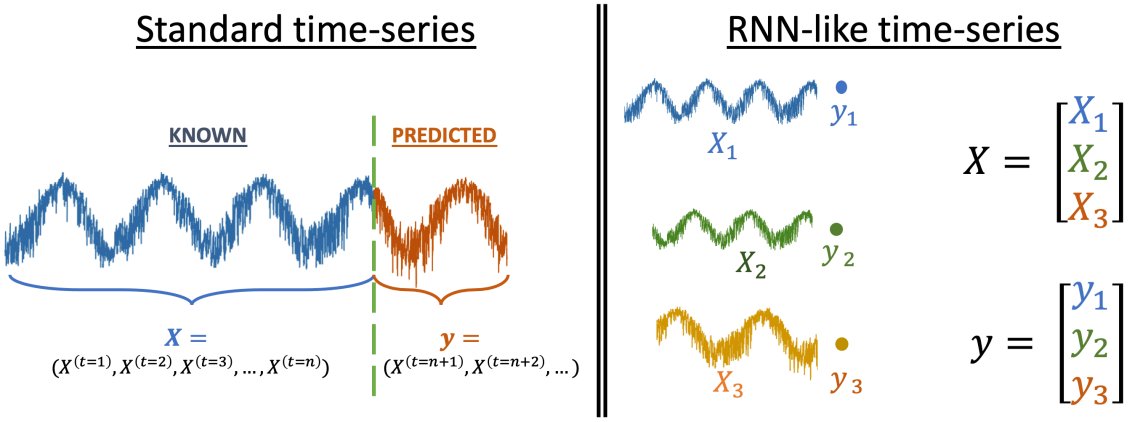
🔮 **Prediction: using Temporal Data for Classification**



👉 Predict the heartbeat category of a patient based on various electrocardiograms (ECG)

- `X.shape` = (None, 100 heartbeats, 4 features)
- `y.shape` = (None, 1 heartbeat, 1 feature = at risk/healthy heart)

🔮 Prediction Recap



In Recurent Neural Networks:

- **`Inputs X`**
  $\rightarrow$
  sequence of **repeated observations** (univariate or multivariate)
  - n_SEQUENCES = number of sequences
  - n_OBSERVATIONS = INPUT_LENGTH = temporal dimension of the input
  - n_FEATURES = features you are observing

✅ A dataset in RNN can have different input lengths

📣 We will discuss how we deal with this with padding later in the lecture!

- **`Outputs y`**
  $\rightarrow$
  is whatever you want to predict,
  - n_SEQUENCES = number of sequences
  - OUTPUT_LENGTH = how many timesteps you want to predict = temporal dimension for the output
  - n_FEATURES = features you want to predict

👀 The feature(s) you want to predict is(are) not naturally similar to the features you are observing.

❌ The shape of what you want to predict must be the same for all the sequences (we'll address this when we talk padding later!)

3.1 Let's code! (Air Pollution Example)

👉 Let's assume that in different cities, you observed weather features every day during four days:

- 🌡️ temperature
- 💨 wind speed
- ☁️ air pollution

You are trying to predict the amount of air pollution the **NEXT DAY**.

In [ ]:
```
import numpy as np

# --- SEQUENCE A (Paris)

day_1 = [10, 25, 60]  # OBSERVATION 1 [temperature, speed, pollution]
day_2 = [13, 10, 80]  # OBSERVATION 2 [temperature, speed, pollution]
day_3 = [ 9,  5, 90]  # OBSERVATION 3 [temperature, speed, pollution]
day_4 = [ 7,  0, 100]  # OBSERVATION 4 [temperature, speed, pollution]

sequence_a = [day_1, day_2, day_3, day_4]

y_a = [110] # Pollution at day 5

# --- SEQUENCE B (Berlin)
sequence_b = [[25, 20, 30], [26, 24, 50], [28, 20, 80], [22, 3, 110]]
y_b = [125]

# --- SEQUENCE C (London)
sequence_c = [[15, 10, 60], [25, 20, 65], [35, 10, 75], [36, 15, 70]]
y_c = [75]

X = np.array([sequence_a, sequence_b, sequence_c]).astype(np.float32)
y = np.expand_dims(np.array([y_a, y_b, y_c]).astype(np.float32), axis=
-1)

print(X.shape)
print(y.shape)
```

```
(3, 4, 3)
(3, 1, 1)
```

In [ ]:
```python
# 0- Imports
from tensorflow.keras import Sequential, Input, layers
from tensorflow.keras.optimizers import Adam

# 1- RNN Architecture
model = Sequential()
model.add(Input(shape=(4, 3)))
model.add(layers.SimpleRNN(units=2, activation='tanh'))
model.add(layers.Dense(1, activation="linear"))

# 2- Compilation
model.compile(loss='mse',
              optimizer=Adam(learning_rate=0.5)) # very high lr so we
can converge with such a small dataset

# 3- Fit
model.fit(X, y, epochs=10000, verbose=0)

# 4- Predict
model.predict(X) # One prediction per city
```

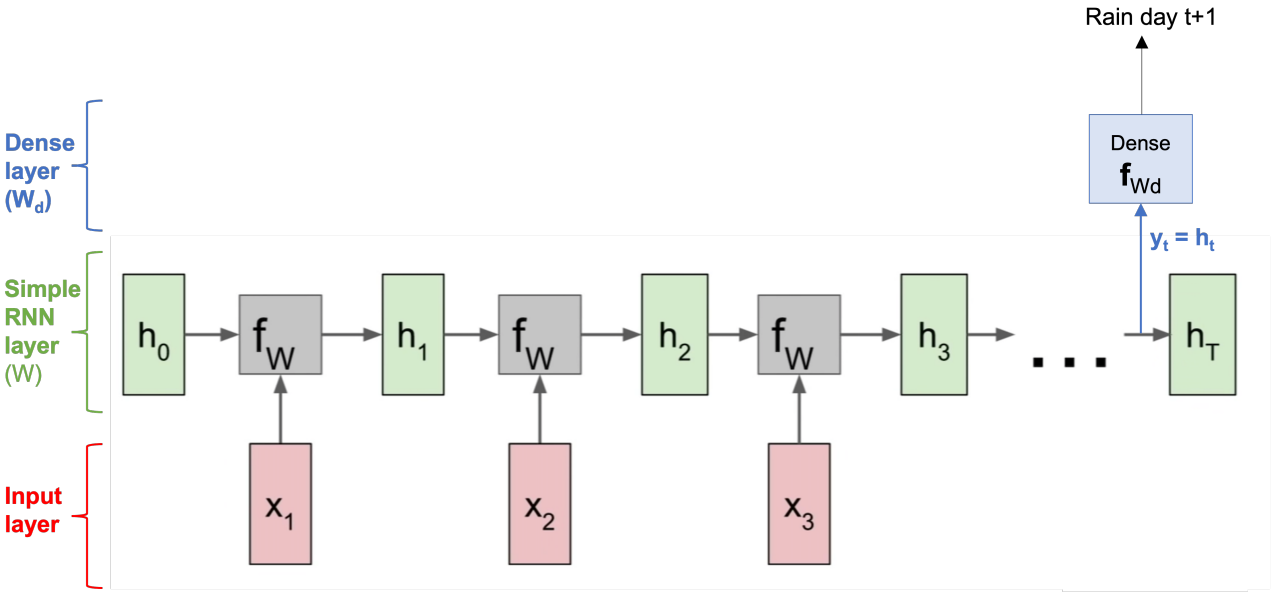
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 89ms/step
```

Out[ ]:
```
array([[117.499985],
       [117.499985],
       [ 75.000015]], dtype=float32)
```

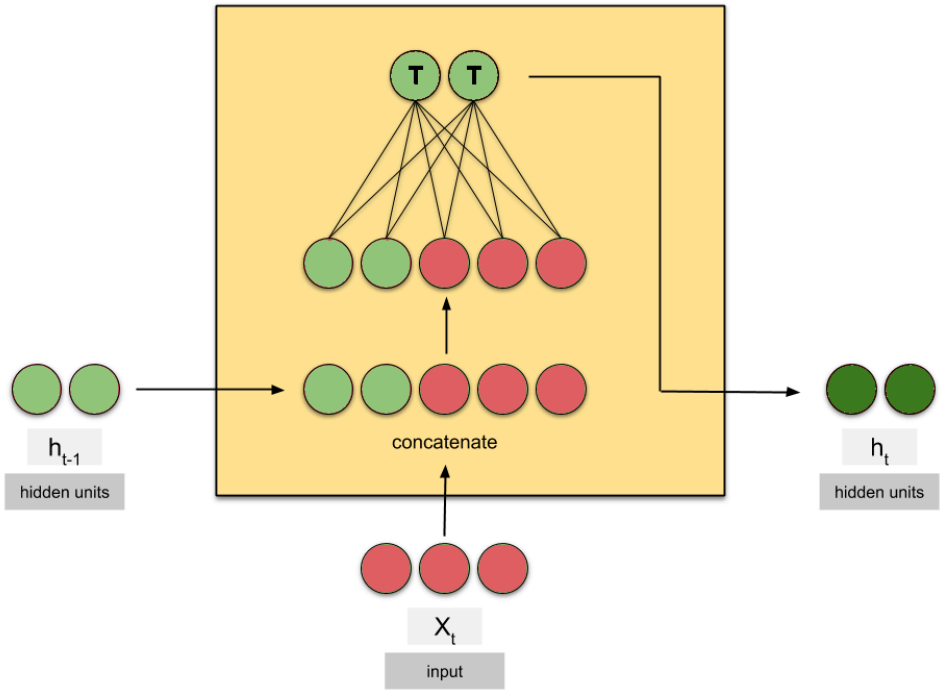
3️⃣ RNNs Under the Hood

🔭 The RNN is fed **one observation**
$x^{(k)}$
at a time (forward in time).

🧠 It maintains an **internal state**
h
that is updated at each time step.


Rain day t+1
Dense
f_{Wd}
$y_t = h_t$
Dense layer (W_d)
Simple RNN layer (W)
Input layer
h_0
f_W
h_1
f_W
h_2
f_W
h_3
h_T
x_1
x_2
x_3

Let's zoom inside this function
f_W
for one time step



- The current observation
  $x^{(t)}$
  and the previous internal state
  $h^{(t-1)}$
  are **concatenated** and fed into a **weights matrix**...
- ...to output the new internal state
  $h^{(t)}$
  !

In [ ]:
```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Pa
simple_rnn (SimpleRNN)	(None, 2)	
dense (Dense)	(None, 1)	

Total params: 47 (192.00 B)

Trainable params: 15 (60.00 B)

Non-trainable params: 0 (0.00 B)

Optimizer params: 32 (132.00 B)

In [ ]:
```
# Recall how we defined our SimpleRNN layer as:
model.add(layers.SimpleRNN(units=2, activation='tanh'))
```

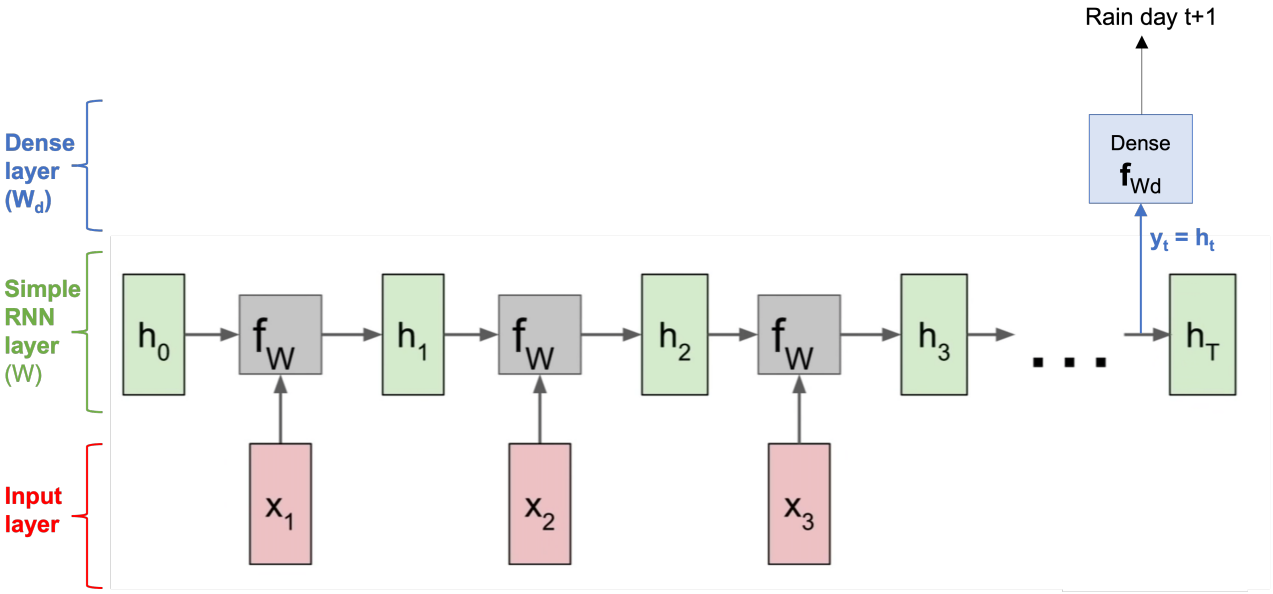
One single RNN layer has a matrix
W
of trainable params, such that

$$h^{(t)} = f_W(h^{(t-1)}, x^{(t)})$$

- The **same** weights
  W
  are used at every time step...
- ...to **recursively** compute its internal state
  h

We can *count* the number of weights of the dense network above as
$n_h(n_h + n_x + 1)$
.

- n_h
  = number of units in our RNN layer
- n_x
  = number of features measured per time-step (e.g. [temperature, humidity, ...])



❗ Two important notes! ❗

1. In a similar way to the number of weight remaining the same in CNNs regardless of image size, the number of weights in a Recurrent Layer has nothing to do with the length of our sequences (4 days in our example).
2. ❌
   $y^{(t)}$
   is **NOT** the prediction/ the target.

   ✅
   $y^{(t)}$
   is a vector of size `RNN_units` (
   $n_h = 2$
   in our example)...

   ✅ ... used as an input to the Dense layer to compute the rain at time
   $t + 1$

How can we improve our model?

Increase the number of RNN units h ?

From `model.add(layers.SimpleRNN(units=2, activation='tanh'))`

to `model.add(layers.SimpleRNN(units=10, activation='tanh'))`

☝️ **Intuition:** With **10** RNN units rather than just 2, this model:

- will try to *capture* **10** interesting *temporal features* from the time-series
  - (maybe: *mean*, *rate of increase*, complex auto-regressive feature, etc...who knows!)
- and combine them into 1 value for our regression task

💡 The number of units can be seen as the **number of memories about features maintained in parallel**.

Add another hidden layer?

From:

```
model.add(layers.SimpleRNN(units=2, activation='tanh'))
model.add(layers.Dense(1, activation="linear"))
```

To:

```
model.add(layers.SimpleRNN(units=2, activation='tanh'))
model.add(layers.Dense(10, activation="linear"))
model.add(layers.Dense(1, activation="linear"))
```

Add another RNN layer?

Much like chaining together Convolutional Layers for CNNs, we can chain together RNN layers 💪

From:

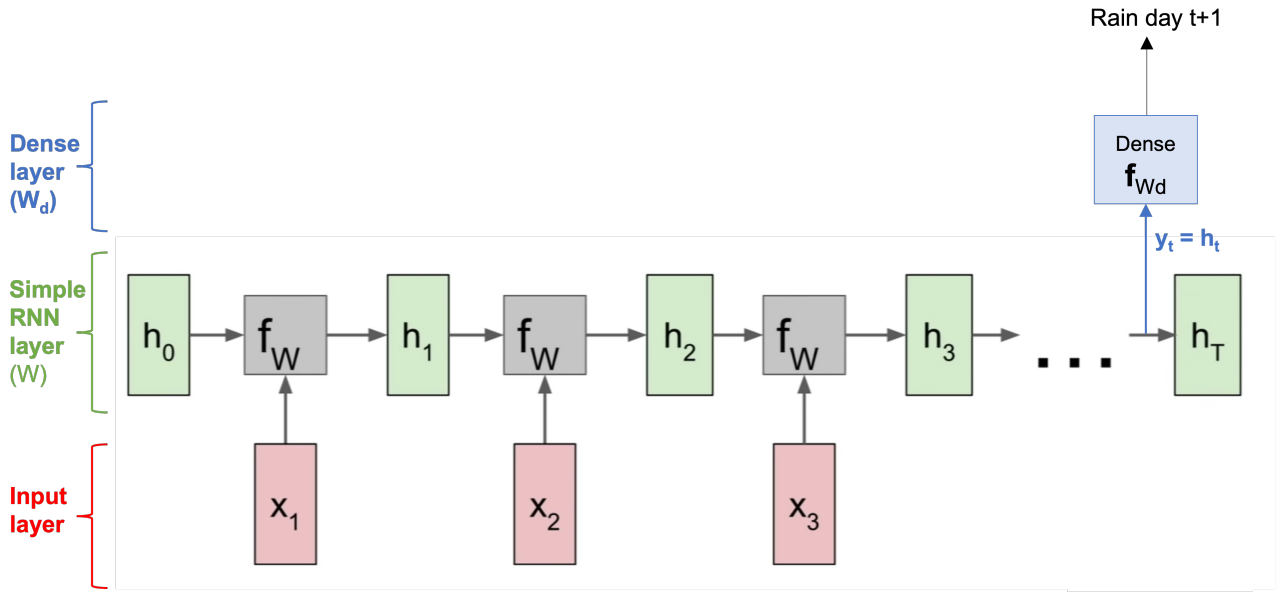
```
model.add(layers.SimpleRNN(units=2, activation='tanh'))
model.add(layers.Dense(1, activation="linear"))
```

To:

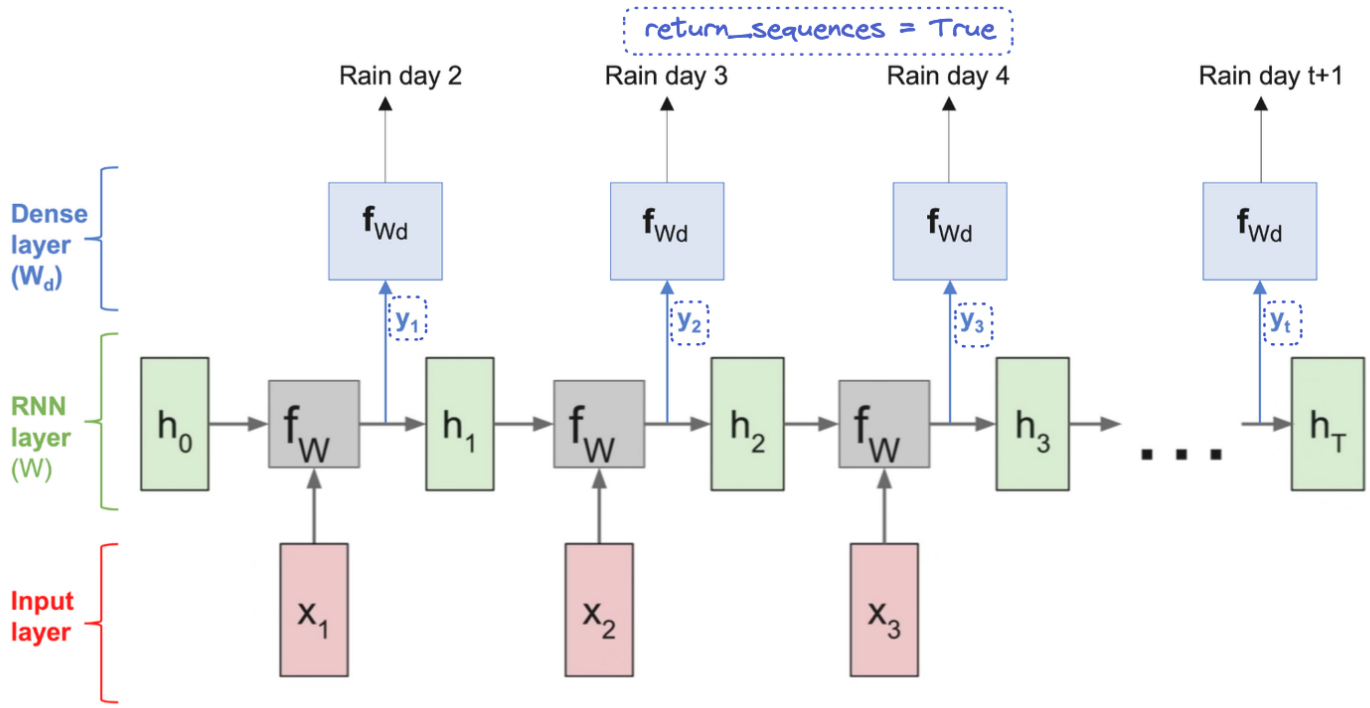
```
model.add(layers.SimpleRNN(units=2, return_sequences=True, activation='tanh'))
model.add(layers.SimpleRNN(units=2, activation='tanh'))
model.add(layers.Dense(1, activation="linear"))
```

❗ Take note! ❗ We've added `return_sequences=True` ; but why?

Our initial approach:



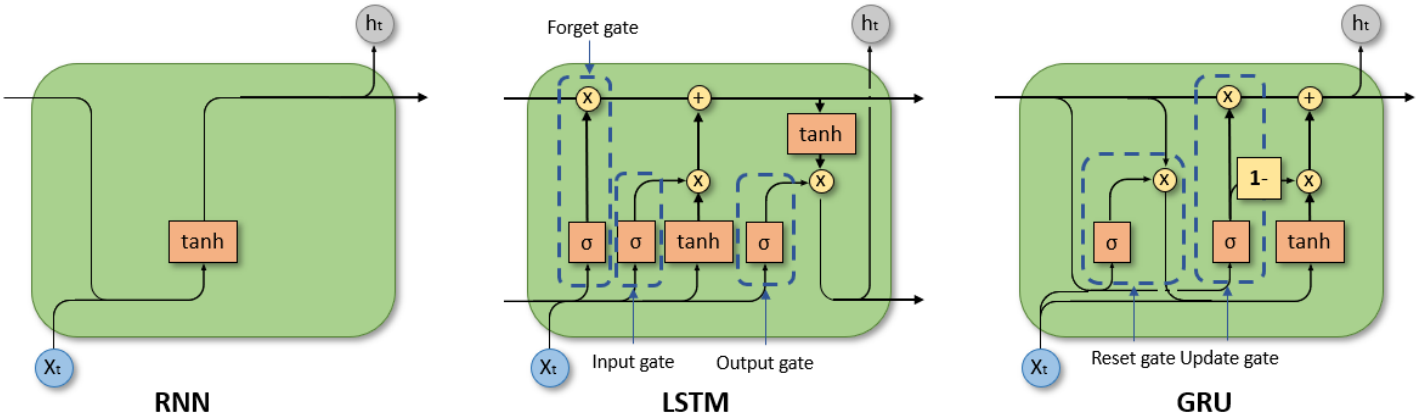
`return_sequences=True` outputs $y^{(t)}$ at each time step.



These can be fed into another RNN layer!

Change the type of RNN we have in our model!

```
model.add(layers.LSTM(units=5, activation='tanh', input_shape=(4,3)))
model.add(layers.Dense(10, activation="linear"))
model.add(layers.Dense(1, activation="linear"))
```



In [ ]:
```
model = Sequential()
model.add(Input(shape=(4, 3)))
model.add(layers.LSTM(units=2, activation='tanh'))
model.add(layers.Dense(1, activation="linear"))
model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Pa
lstm (LSTM)	(None, 2)	
dense_1 (Dense)	(None, 1)	

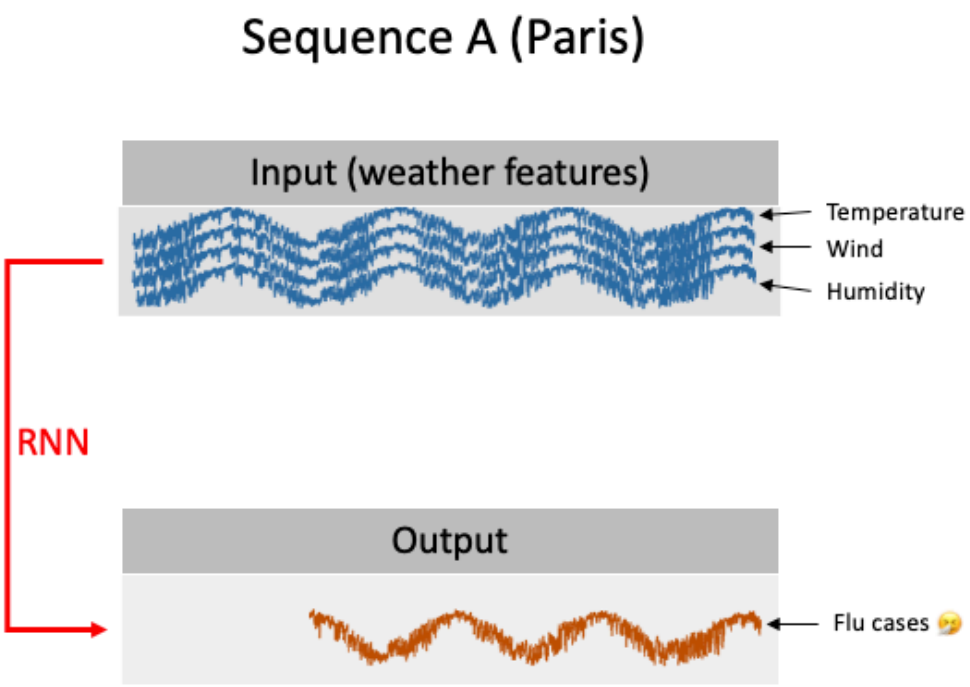
Total params: 51 (204.00 B)

Trainable params: 51 (204.00 B)

Non-trainable params: 0 (0.00 B)

⚠️ N.B. We now have 4 times as many weights in our RNN layer thanks to the additional gates! ⚠️

Predicting an entire *sequence*



For each sequence/city:

$X^{(t)}$
= weather observed at day
t

$y^{(t)}$
= Flu cases reported at day
t

Why not use
y
at previous time steps as a feature?

❗ **To predict a sequence, `y_train` need to be sequences too** ❗

In [ ]:
```
import numpy as np

# --- SEQUENCE A (Paris)

sequence_a = [[10, 25, 50],  # OBS day 1
              [13, 10, 70],  # OBS day 2
              [ 9,  5, 90],  # OBS day 3
              [ 7,  0, 95]]  # OBS day 4

y_a = [70,   # flu cases day 1
       90,   # flu cases day 2
       95,   # flu cases day 3
       110,] # flu cases day 4

# --- SEQUENCE B (Berlin)
sequence_b = [[25, 20, 30], [26, 24, 50], [28, 20, 80], [22, 3, 110]]
y_b = [50, 80, 110, 125]

# --- SEQUENCE C (London)
sequence_c = [[15, 10, 60], [25, 20, 65], [35, 10, 75], [36, 15, 70]]
y_c = [65, 75, 70, 30]

X = np.array([sequence_a, sequence_b, sequence_c]).astype(np.float32)
y = np.expand_dims(np.array([y_a, y_b, y_c]).astype(np.float32),axis=-
1)

print(X.shape)
print(y.shape)
```

```
(3, 4, 3)
(3, 4, 1)
```

❗ Remember the wording ❗

```
X.shape = (n_SEQUENCES, n_OBSERVATIONS, n_FEATURES)
X.shape = (3 cities, 4 days, 3 features)
y.shape = (3 cities, 4 days, 1 feature predicted each day)
```

In [ ]:
```
model_2 = Sequential()

model_2.add(layers.SimpleRNN(2, return_sequences=True))
model_2.add(layers.Dense(1, activation='relu'))

model_2.compile(loss='mse', optimizer='adam')
model_2.fit(X, y, verbose=0)
```

Out[ ]:
```
<keras.src.callbacks.history.History at 0x304fe4740>
```

In [ ]:
```
print("y_pred shape:", model_2.predict(X).shape)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 163ms/step
y_pred shape: (3, 4, 1)
```

4️⃣ NLP: Why we need RNNs

Because order matters!

An obvious example:

> "The teacher inspired the students with a passionate speech about deep learning"
> "The students inspired the teacher with a passionate speech about deep learning"

A more subtle example:

> "The company hired talented employees and achieved great success."
> "The company achieved great success and hired talented employees."

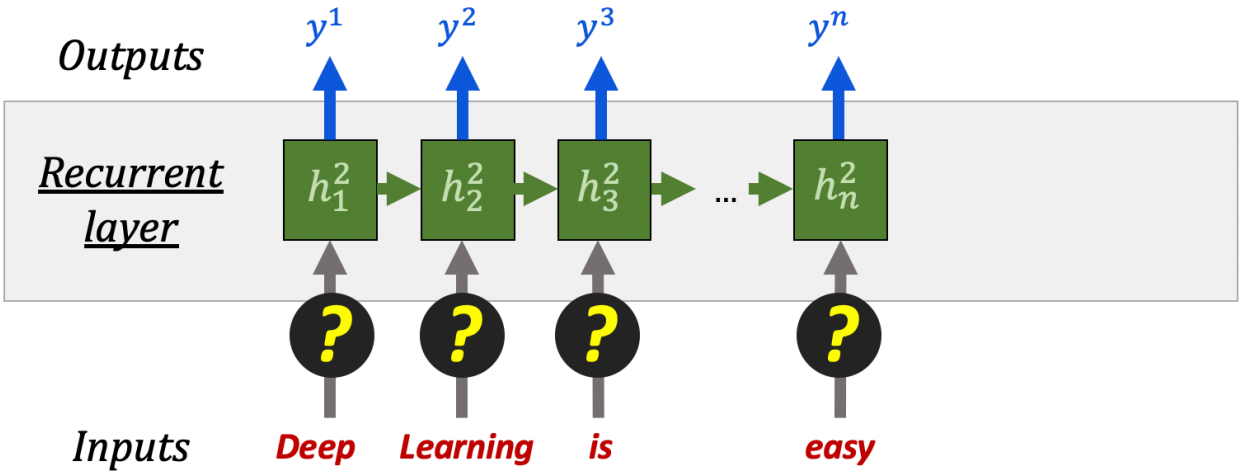
Let's give ourselves a task to solve by the end of the lecture!

Our `X` will be sentences and our `y` will be whether or not those sentences are positive or negative!

How do we go about this?

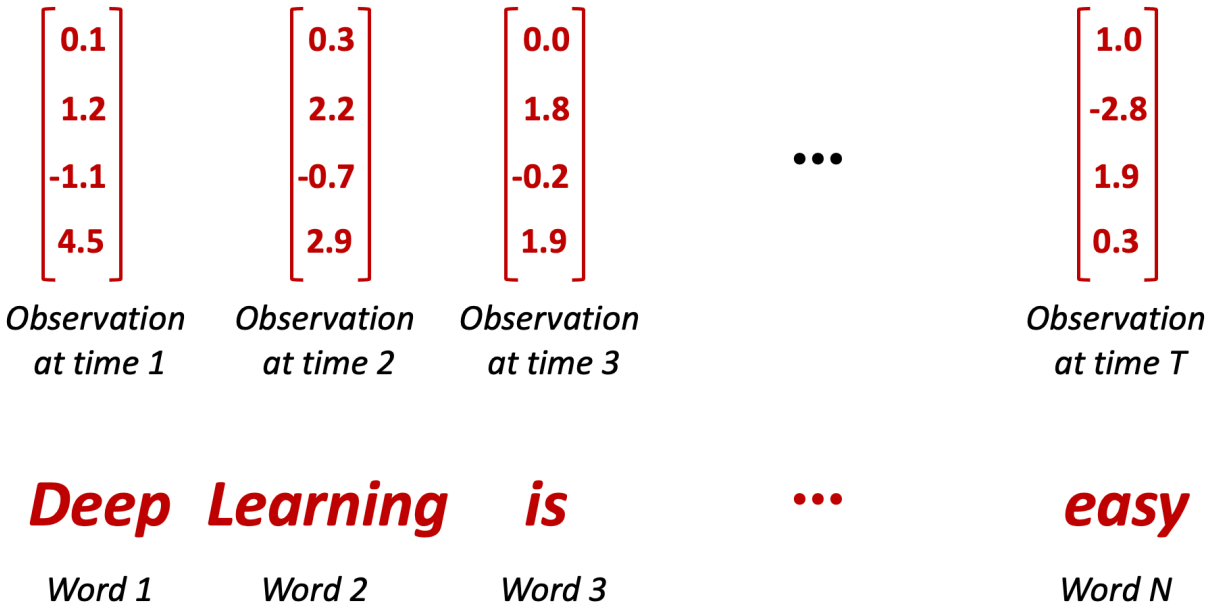
"This movie is the best thing I've seen in my life" -> ✅ "I had to leave this film because I was so bored" -> ❌ "Anything with Nicolas Cage in it is a masterpiece: 5/5" -> ✅

How to feed Recurrent Neural Networks with words?

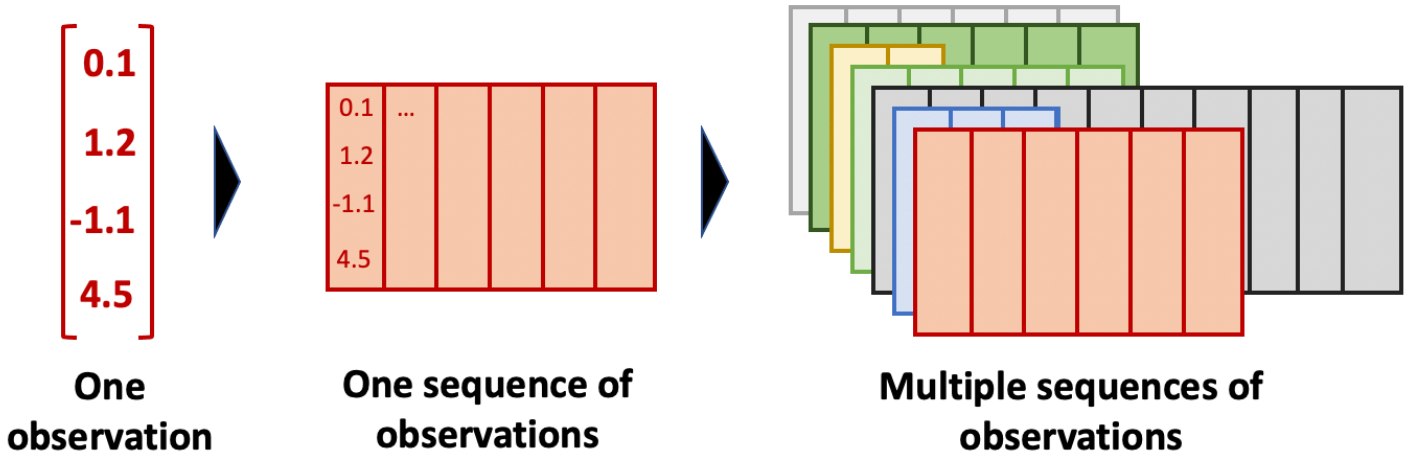


We have to convert words into numbers somehow!

Text is also a form of recurrent data, where a sentence is a sequence of words, each being the *observation*. For now, let's imagine we can express each **word** as **four numbers** (or a point/ vector in 4-D space)



What will our X look like?



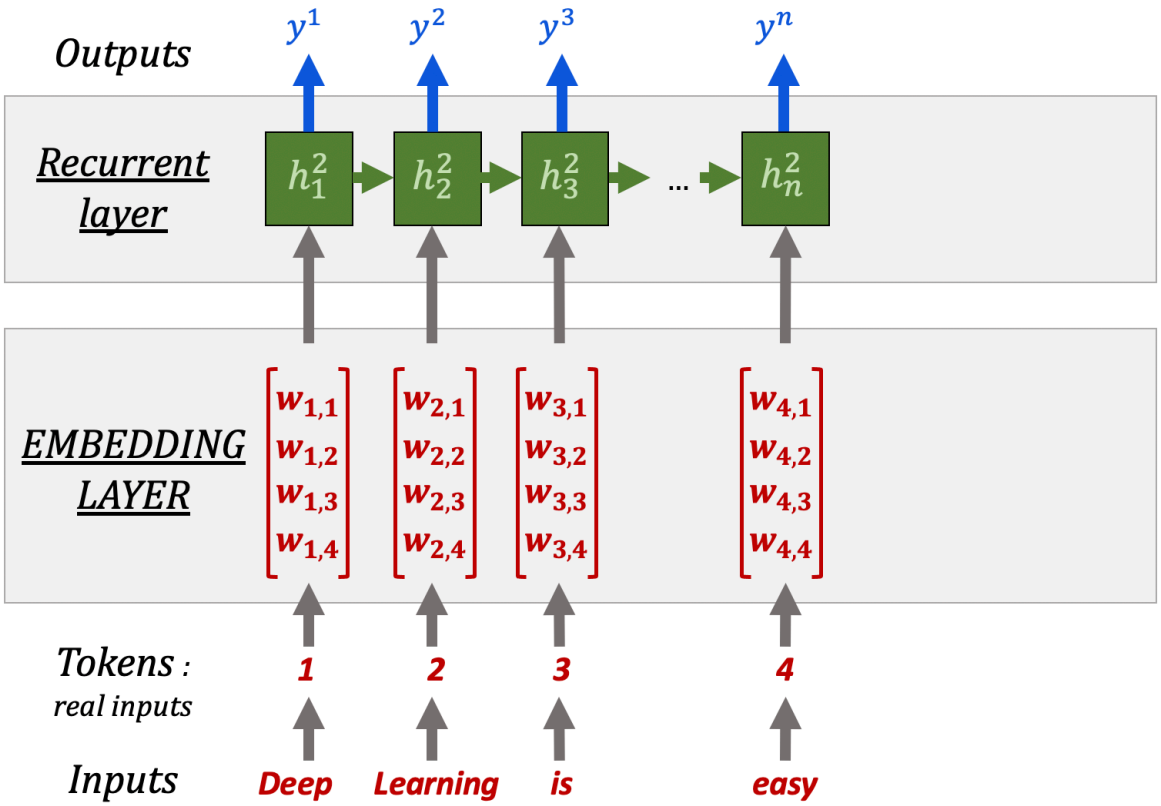
X is just a sequence of observations! Univariate, multivariate, an image, ... The point is that the observation is repeated through time!

```
X.shape = (N_SEQUENCES, N_OBSERVATIONS, N_FEATURES)
```

Warning: The number of observations can vary from one sequence to another! (This is when you pad - we'll get back to that)

First, we'll need tokens

Neural networks' can't work directly on words, so you will still need to provide it with tokens.

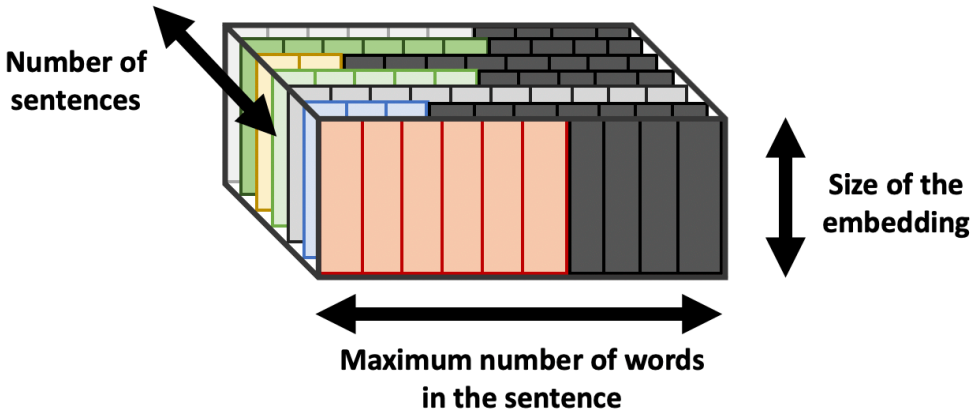


In [ ]:
```
import numpy as np
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer

### Let's create some mock data
def get_mock_up_data():
    sentence_1 = 'This movie was awful'
    sentence_2 = 'I loved every moment of this movie!'
    sentence_3 = 'I want a refund; I was so bored!'

    X = [sentence_1, sentence_2, sentence_3]
    y = np.array([0., 1., 0.])

    ### Let's tokenize the vocabulary
    tk = Tokenizer()
    tk.fit_on_texts(X)
    vocab_size = len(tk.word_index)
    print(f'There are {vocab_size} different words in your corpus')
    X_token = tk.texts_to_sequences(X)

    ### Pad the inputs
    X_pad = pad_sequences(X_token, dtype='float32', padding='pre')

    return X_pad, y, vocab_size

X_pad, y, vocab_size = get_mock_up_data()
print("X_pad.shape", X_pad.shape)
X_pad
```

```
There are 14 different words in your corpus
X_pad.shape (3, 8)
```

Out[ ]:
```
array([[ 0.,  0.,  0.,  0.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  6.,  7.,  8.,  9.,  2.,  3.],
       [ 1., 10., 11., 12.,  1.,  4., 13., 14.]], dtype=float32)
```

What's going on with the `pad_sequences` function?



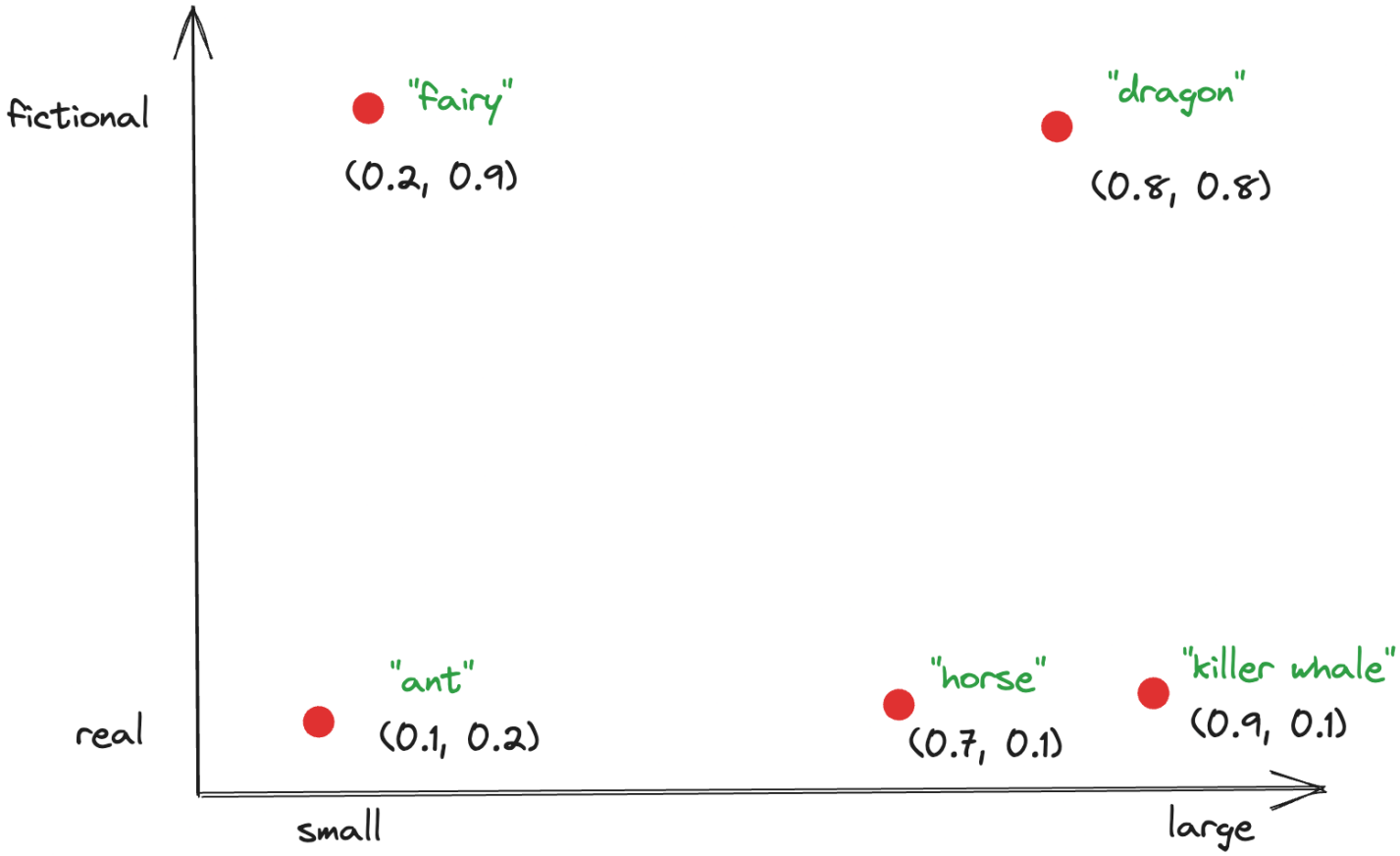
🤔 To"post" or to "pre"? The eternal question: there's no definitive answer and it varies by architecture, but results seem to lean towards using "pre" (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7471694/)

Now we need to make the leap from tokens to vectors

We want each word to be represented by a vector ↗ of *chosen* length. How?

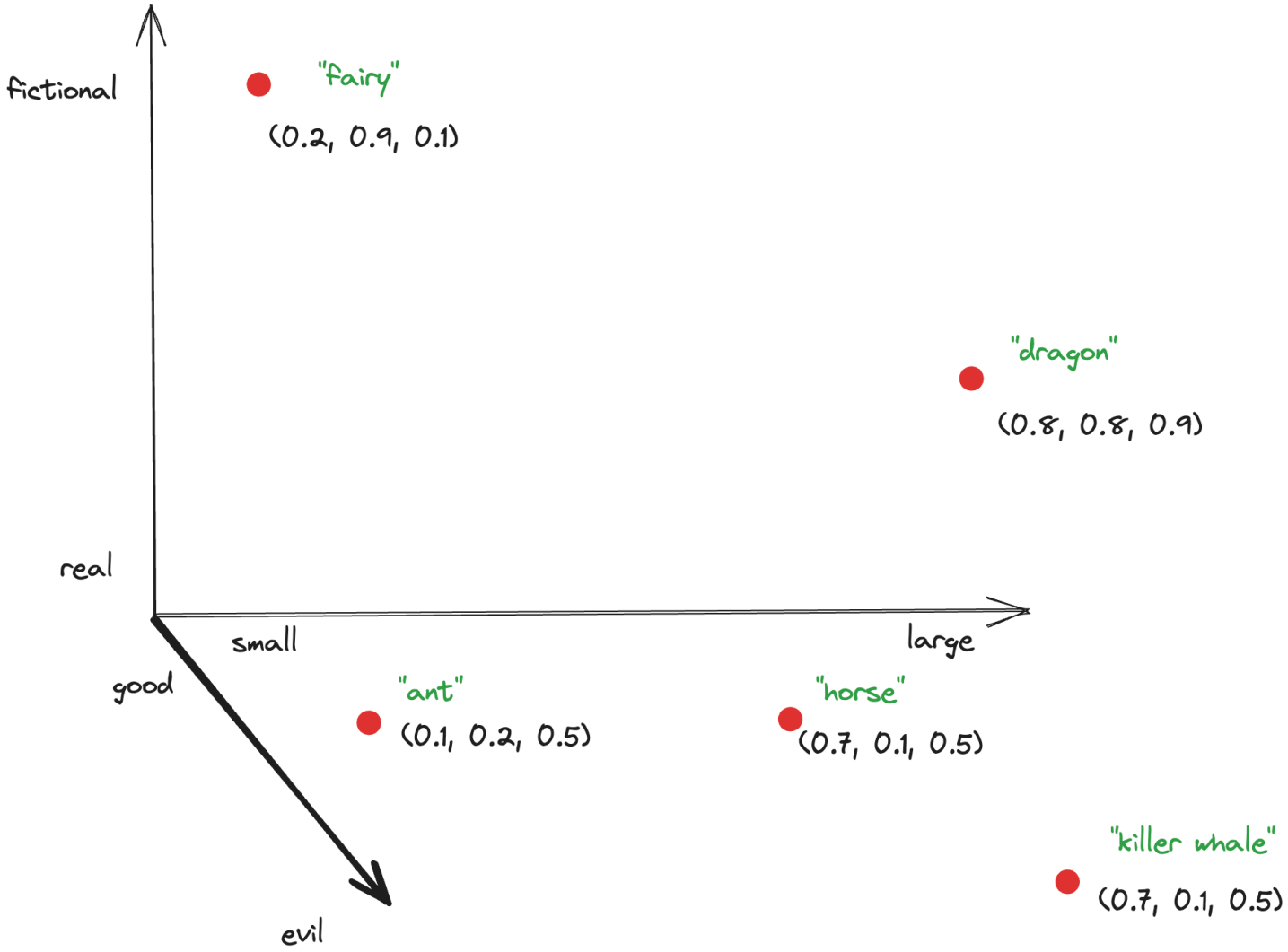
Consider a 2D embedding

To further specify the meaning of each word, we add a dimension: we rank them according to their relative abstraction.



We can't differentiate much between "horse" and "killer whale" 😤

Adding a 3rd dimension

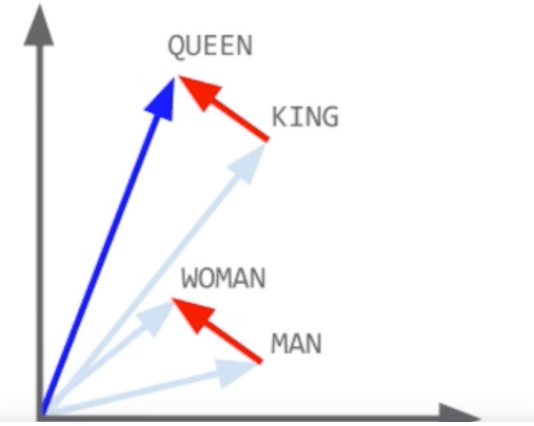


Arithmetic on words?

If we embed correctly, we could dream of performing mathematical operations on the embeddings, which would have meaning in terms of natural language.

E.g. we could express the sentence "Queen is to king as man is to..." as

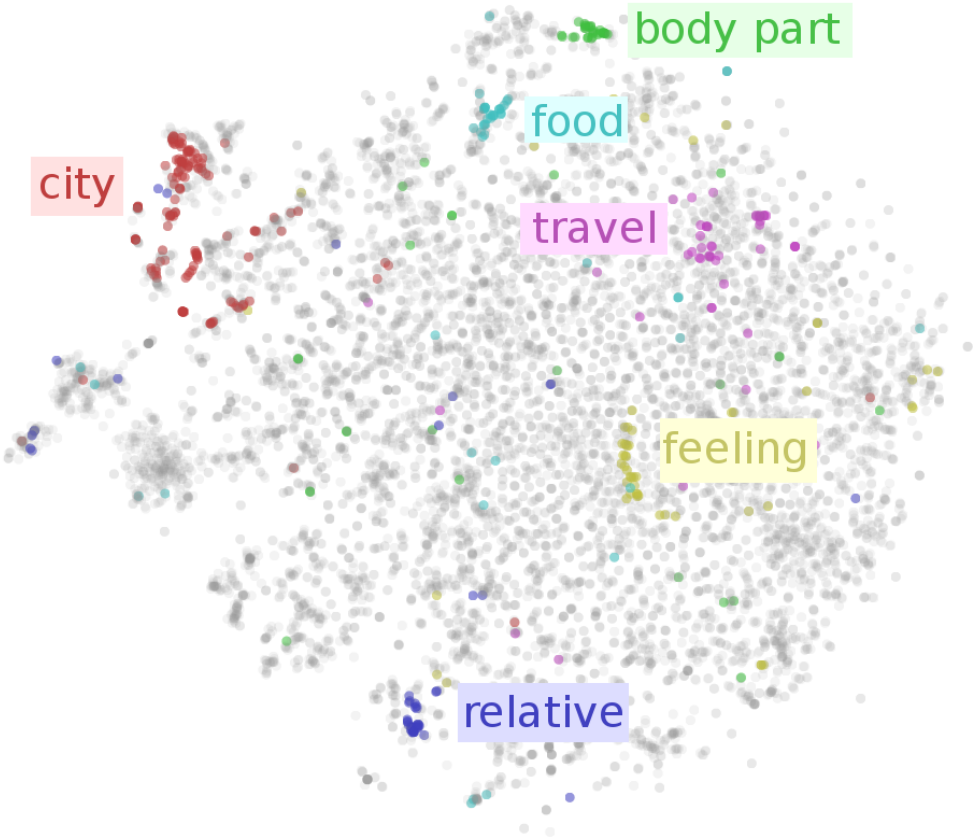
$$V(Queen) - V(King) + V(Man) = ?$$



Here, as the red vectors are similar, we could expect $V(Queen) - V(King) = V(Woman) - V(Man)$
.

So the answer we're looking for might be `Woman` .

What makes a good embedding?

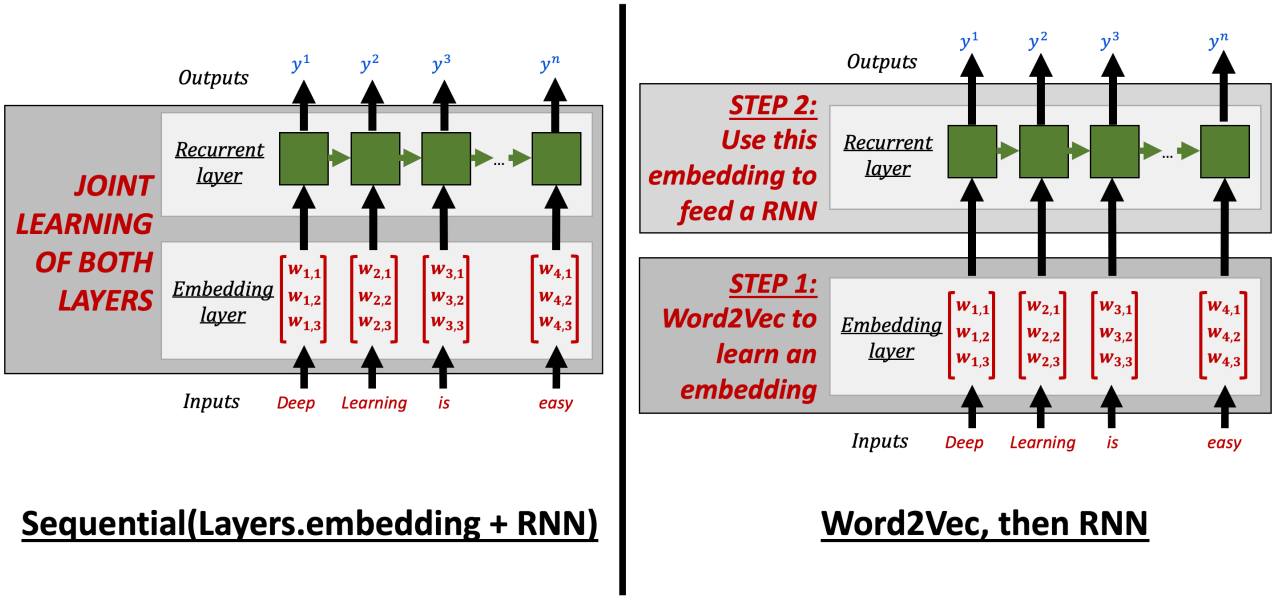


Semantically *close* words are mathematically *close* in this space

❗ Usually, word embedding spaces are from **30** up to **300** dimensions ❗

🤔 But wait ... how do you choose these numbers? Surely we're not scoring all of our words on n difference axes?

Two options: learn with `layers.Embedding` or with Word2Vec



Sequential(Layers.embedding + RNN) | **Word2Vec, then RNN**

👈 The left option allows you to have a representation that is perfectly suited to your task! However, it increases the number of parameters to learn, and thus:

- the time of each epoch (more parameters to optimize during back-propagation)
- the time to converge (because more parameters to find overall)

👉 On the other hand, word2vec is an unsupervised learning method not specifically designed for your task (may be sub-optimal) but training it is very fast! You will also be able to optimize your RNN faster as you'll have less parameters.

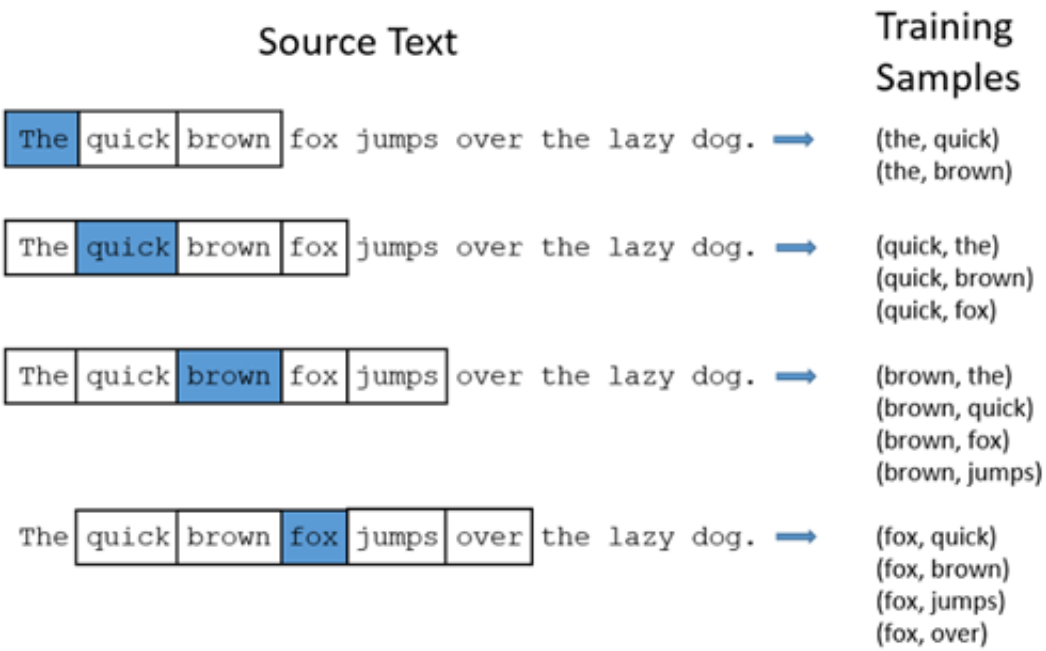
- ❗ Prefer Word2Vec on small corpus (esp. with transfer learning)

First option: `layers.Embedding` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding)

```
Embedding(input_dim=VOCAB_SIZE,
          input_length=MAX_SENTENCE_LENGTH,
          output_dim=EMBED_DIM,
          mask_zero=True)
```

We can insert this layer after our tokenization and all of our words are embedded as the model trains!

Second option: Word2Vec



💡 We use unsupervised learning to look at the words around each chosen word and assume that these will be relevant to the word we're interested in!

Instead of training it on your training set (especially if it is very small), you can directly load a pretrained embedding

In [ ]:
```
import gensim.downloader

print(list(gensim.downloader.info()['models'].keys()))

model_wiki = gensim.downloader.load('glove-wiki-gigaword-50')
```

```
['fasttext-wiki-news-subwords-300', 'conceptnet-numberbatch-17-06-30
0', 'word2vec-ruscorpora-300', 'word2vec-google-news-300', 'glove-wi
ki-gigaword-50', 'glove-wiki-gigaword-100', 'glove-wiki-gigaword-20
0', 'glove-wiki-gigaword-300', 'glove-twitter-25', 'glove-twitter-5
0', 'glove-twitter-100', 'glove-twitter-200', '__testing_word2vec-ma
trix-synopsis']
```

We can even try out our aforementioned example with code!

In [ ]:
```
# N.B. Words not in glove-wiki-gigaword-50 will not have vectors computed
example_1 = model_wiki["queen"] - model_wiki["king"] + model_wiki["man"]
```

In [ ]:
```
model_wiki.most_similar(example_1)
```

Out[ ]:
```
[('woman', 0.8903914093971252),
 ('girl', 0.8453726768493652),
 ('man', 0.8301756381988525),
 ('her', 0.7845831513404846),
 ('boy', 0.7763065099716187),
 ('she', 0.7619765400886536),
 ('herself', 0.7597628831863403),
 ('blind', 0.7296754717826843),
 ('mother', 0.7230340242385864),
 ('blonde', 0.713614284992218)]
```

What about "good is to evil as cold is to..."?

In [ ]:
```
example_2 = model_wiki["good"] - model_wiki["evil"] + model_wiki["cold"]
```

In [ ]:
```
model_wiki.most_similar(example_2)
```

Out[ ]:
```
[('warm', 0.7870427966117859),
 ('dry', 0.7643216848373413),
 ('hot', 0.750431478023529),
 ('cool', 0.7491167187690735),
 ('weather', 0.7476345896720886),
 ('getting', 0.7447267770767212),
 ('good', 0.7442173361778259),
 ('cold', 0.7425146102905273),
 ('low', 0.7231549024581909),
 ('little', 0.7223491668701172)]
```

5️⃣ Coded example: Sentiment analysis

First let's load up our data 💪

In [ ]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense

# Set parameters
max_features = 10000  # Maximum number of words to get out of our imdb data
max_len = 500  # Maximum sequence length
embedding_dim = 50  # Dimensionality of word embeddings

# Load the IMDb dataset
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
17464789/17464789 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
```

Next we need to pad 🗒️

In [ ]:
```
# Pad sequences to a fixed length
x_train = sequence.pad_sequences(x_train, maxlen=max_len)
x_test = sequence.pad_sequences(x_test, maxlen=max_len)
```

Now we'll construct and compile our model 🔨

In [ ]:
```
# Build the model
model = Sequential()
model.add(Embedding(max_features, embedding_dim))
model.add(LSTM(64))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Finally we fit and evaluate 💯

In [ ]:
```
# Train the model
model.fit(x_train, y_train, batch_size=128, epochs=3, validation_data=
(x_test, y_test))

# Evaluate the model
loss, accuracy = model.evaluate(x_test, y_test)
print(f'Test loss: {loss:.4f}')
print(f'Test accuracy: {accuracy:.4f}')
```

```
Epoch 1/3
196/196 ━━━━━━━━━━━━━━━━━━━━ 75s 380ms/step - accuracy: 0.6632 - los
s: 0.5910 - val_accuracy: 0.8569 - val_loss: 0.3400
Epoch 2/3
196/196 ━━━━━━━━━━━━━━━━━━━━ 76s 387ms/step - accuracy: 0.8904 - los
s: 0.2785 - val_accuracy: 0.8722 - val_loss: 0.3052
Epoch 3/3
196/196 ━━━━━━━━━━━━━━━━━━━━ 75s 382ms/step - accuracy: 0.9276 - los
s: 0.2000 - val_accuracy: 0.8694 - val_loss: 0.3089
782/782 ━━━━━━━━━━━━━━━━━━━━ 21s 27ms/step - accuracy: 0.8683 - los
s: 0.3109
Test loss: 0.3089
Test accuracy: 0.8694
```

Done! We can predict sentiment with a very high degree of accuracy and minimal pre-processing!

6️⃣ Further reading

- 📚 Stanford Cheat Sheet (https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks)
- 📚 Medium - Illustrated Guide to RNN (https://medium.com/towards-data-science/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9)
- 📚 Medium - Illustrated Guide to LSTM and GRUs (https://medium.com/towards-data-science/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21)
- 🖥️ RNN explained with 3*3 matrices - 21min (https://www.youtube.com/watch?v=UNmqTiOnRfg)
- 🖥️ *RNN & LSTM* explained without math - 24min (https://www.youtube.com/watch?v=WCUNPb-5EYI)

Deep Learning courses - videos

- CS224N: Natural Language Processing with Deep Learning (https://www.youtube.com/playlist?list=PLoROMvodv4rOhcuXMZkNm7j3fVwBBY42z) by Standford University (recommended)
- 92 step-by-step videos (https://www.youtube.com/watch?v=SGZ6BttHMPw&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH) by Hugo Larochelle.
- Deep Learning course (https://www.coursera.org/specializations/deep-learning), by Andrew Ng.
- MIT Introduction to Deep Learning (https://www.youtube.com/playlist?list=PLtBw6njQRU-rwp5__7C0oIVt26ZgjG9NI), from MIT.
- Deep Learning for Computer Vision (https://www.youtube.com/playlist?list=PL5-TkQAfAZFbzxjBHtzdVCWE0Zbhomg7r), from Michigan University.

Deep Learning courses - books

- *Deep Learning book* (https://www.deeplearningbook.org/), by Ian Goodfellow, Yoshua Bengio, Aaron Courville
- *Deep Learning with Python*, by François Chollet231 Lecture on RNN - 1h20] (https://www.youtube.com/watch?v=6niqTuYFZLQ (https://www.youtube.com/watch?v=6niqTuYFZLQ))