Deep learning: processing sequence data

Examples from the book “Deep Learning with Python”, F. Chollet
https://github.com/fchollet/deep-learning-with-python-notebooks
(especialmente capitulo 11)

Processing sequence data

One area of success of DL is the ability to process sequential data, in particular textual and audio, as well as time series, with good performance

One of the most commonly used topologies in these applications are recurrent neural networks with various configurations

Applications include:

- classification of documents; identification of relevant documents
- machine translation
- sentiment analysis
- named entity recognition (e.g. given names, cities, ...) and the relationships between them
- time series forecasting
- prediction of protein function, structure, and properties
- QSAR/ predicting activities of chemical compounds

Some tasks in Natural Language Processing (NLP)

- “What’s the topic of this text?” - **text classification**
- “Does this text contain abuse?” - **content filtering**
- “Does this text sound positive or negative?” - **sentiment analysis**
- “What should be the next word in this incomplete sentence?” - **language modeling**
- “How would you say this in German?” - **translation**
- “How would you summarize this article in one paragraph?” - **summarization**

Processing texts

DL models only process numerical data, and thus do not accept raw text – need to pre-process texts – ***vectorization*** – to convert into numerical vectors

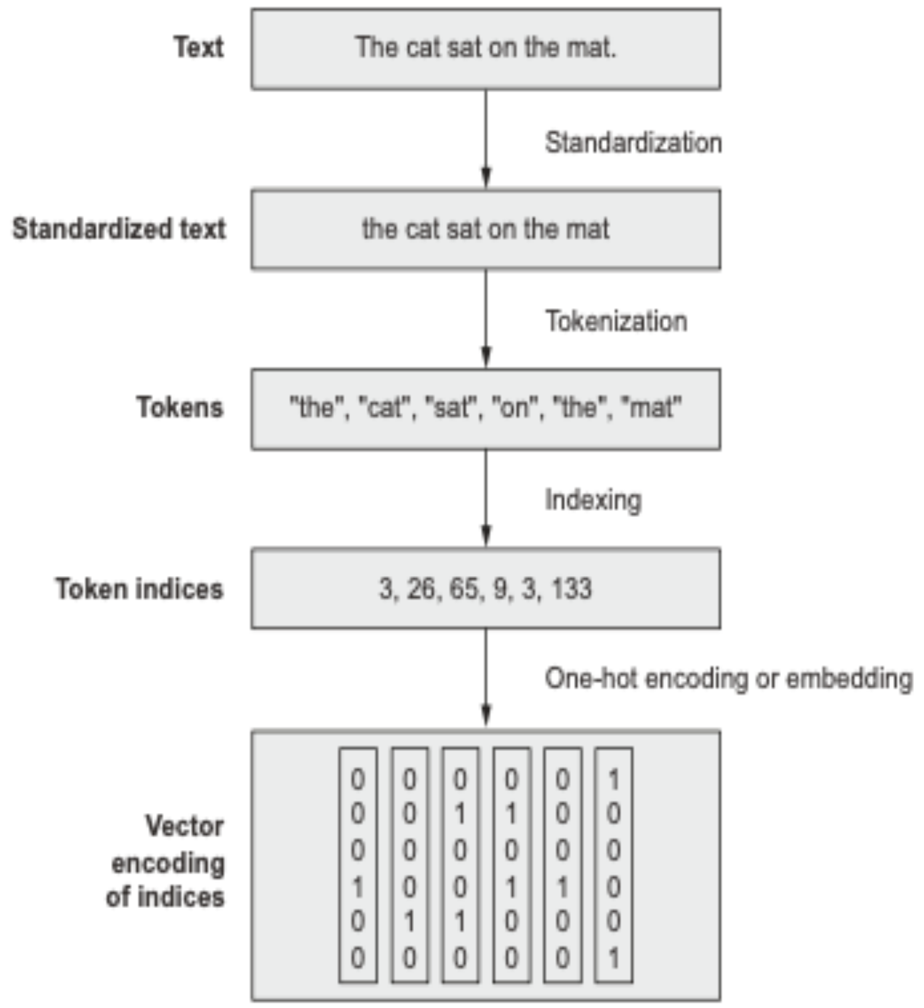
Several steps are usually taken:

- ***Standardization*** – removing punctuation; upper case to lower case; stemming; etc
- ***Tokenization*** - splitting the text into units (tokens), that may be words, groups of words, characters, groups of characters
- ***Indexing*** the tokens – converting into a numerical vector (e.g. one-hot encoding)

Processing texts

Typically, we see texts as word sequences

Texts are converted to numeric vectors, where words (or other types of tokens) are encoded as indexes of a vocabulary that typically has a predefined maximum size



Tokenization

- **Word-level tokenization**— tokens are space-separated (or punctuation-separated) substrings. A variant is to split words into subwords when applicable—for instance, treating “staring” as “star+ing” or “called” as “call+ed.”
- **N-gram tokenization**— tokens are groups of N consecutive words. For instance, “the cat” or “he was” would be 2-gram tokens (also called bigrams).
- **Character-level tokenization**— each character is its own token. In practice, this scheme is rarely used, only in specialized contexts, like text generation or speech recognition.

One-hot encoding: example

```
vocabulary = {}
for text in dataset:
       text = standardize(text)
       tokens = tokenize(text)
       for token in tokens:
              if token not in vocabulary:
                     vocabulary[token] = len(vocabulary)
```

```
def one_hot_encode_token(token):
       vector = np.zeros((len(vocabulary),))
       token_index = vocabulary[token]
       vector[token_index] = 1
       return vector
```

For a more complete example:
https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter11_part01_introduction.ipynb

Most common representation: ***One-hot encoding*** where each word (token) has an index; representation of a word (token) is a vector with a value of 1 in that index and zeros in the remaining

Usually, index 0 is for "mask" used for instance in padding
Index 1 is used for words not in vocabulary

F. Chollet

One-hot encoding: example

V = [a, aaron, …, zulu, <UNK>] **Vocabulary**

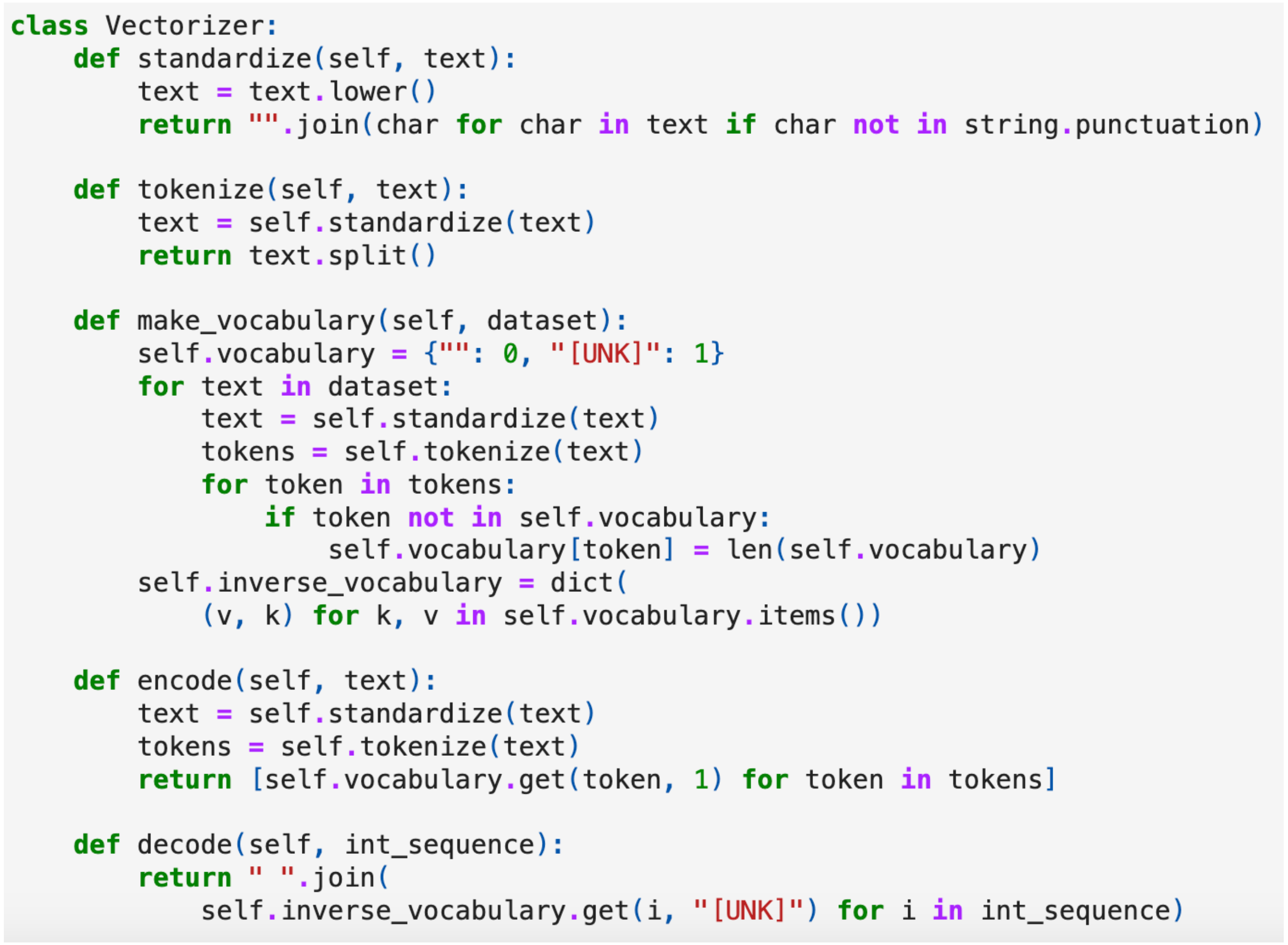
Man (5391) Woman (9853) King (4914) Queen (7157) Apple (456) Orange (6257)

$$
\begin{array}{r}\\ \\ \\ \\ \\ 5391\\ \\ \\ \\ \end{array}\begin{bmatrix}0\\0\\0\\0\\\vdots\\1\\\vdots\\0\\0\end{bmatrix}
\quad
\begin{array}{r}\\ \\ \\ \\ \\ \\ 9853\\ \\ \\ \end{array}\begin{bmatrix}0\\0\\0\\0\\0\\\vdots\\1\\\vdots\\0\end{bmatrix}
\quad
\begin{array}{r}\\ \\ \\ \\ 4914\\ \\ \\ \\ \\ \end{array}\begin{bmatrix}0\\0\\0\\\vdots\\1\\\vdots\\0\\0\\0\end{bmatrix}
\quad
\begin{array}{r}\\ \\ \\ \\ \\ \\ 7157\\ \\ \\ \end{array}\begin{bmatrix}0\\0\\0\\0\\0\\\vdots\\1\\\vdots\\0\end{bmatrix}
\quad
\begin{array}{r}\\ \\ 456\\ \\ \\ \\ \\ \\ \\ \end{array}\begin{bmatrix}0\\\vdots\\1\\\vdots\\0\\0\\0\\0\\0\end{bmatrix}
\quad
\begin{array}{r}\\ \\ \\ \\ \\ \\ 6257\\ \\ \\ \end{array}\begin{bmatrix}0\\0\\0\\0\\0\\\vdots\\1\\\vdots\\0\end{bmatrix}
$$

$|V| = 10000$

Andrew Ng

Code example in Python



```python
class Vectorizer:
    def standardize(self, text):
        text = text.lower()
        return "".join(char for char in text if char not in string.punctuation)

    def tokenize(self, text):
        text = self.standardize(text)
        return text.split()

    def make_vocabulary(self, dataset):
        self.vocabulary = {"": 0, "[UNK]": 1}
        for text in dataset:
            text = self.standardize(text)
            tokens = self.tokenize(text)
            for token in tokens:
                if token not in self.vocabulary:
                    self.vocabulary[token] = len(self.vocabulary)
        self.inverse_vocabulary = dict(
            (v, k) for k, v in self.vocabulary.items())

    def encode(self, text):
        text = self.standardize(text)
        tokens = self.tokenize(text)
        return [self.vocabulary.get(token, 1) for token in tokens]

    def decode(self, int_sequence):
        return " ".join(
            self.inverse_vocabulary.get(i, "[UNK]") for i in int_sequence)
```

In practice, we can use the **TextVectorization** class from Keras !

See example in the notebook

Full code:

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter11_part01_introduction.ipynb

Representing groups of words in ML

- Counts of (or presence) of words or tokens (also known as **bags of words**) can can be used as features in Machine Learning approaches (tabular datasets)
- In this case, the order of the words is neglected, so we treat each example as a **set** of words or n-grams
- Main advantage: represent each example text as a vector, where each entry is a presence indicator/ count for a given word/ n-gram. The vector has as many dimensions as words in the vocabulary (as we did in the last session)
- Main disadvantage: we do not consider the order of the words

Check example for the IMDB dataset in : https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter11_part01_introduction.ipynb

F. Chollet

Representing order of words in ML

- By considering N-grams we can recover local order information
- To avoid combinatorial explosion, **bigrams** (pairs of words/ tokens) are the mostly used
- Therefore, the presence or counts of bigrams will be the considered features in this approach, still keeping a tabular dataset but increasing significantly the number of features
- As before we can use presence/ absence (binary) or counts

Check example for the IMDB dataset in : https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter11_part01_introduction.ipynb

F. Chollet

TF-IDF normalization

- Objective of normalization is to take counts of words or N-grams considering the number of times a word typically occurs in the language
- TF-IDF “term frequency, inverse document frequency” is the most used method for normalization
- It divides term frequency (how many times the term appears in the current document) by “document frequency” which estimates how often the term comes up across the dataset
- So, a term will be important if it occurs much more frequently in the given document than in the overall dataset
- TF-IDF may be used to create normalized features from word/N-gram counts

Check example for the IMDB dataset in : https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter11_part01_introduction.ipynb

F. Chollet

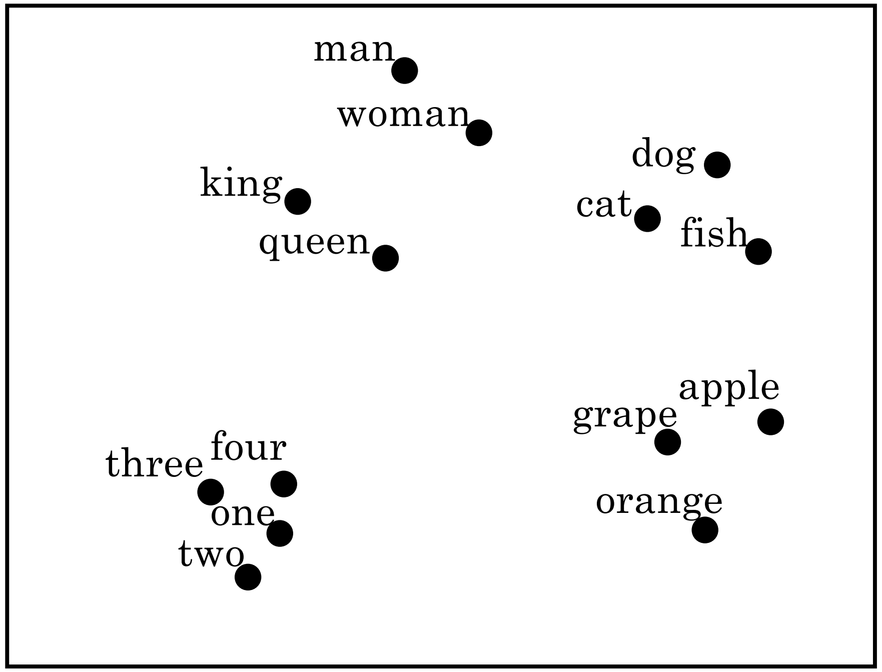
Word embeddings

Each word is represented by a relatively small "dense" numeric vector (e.g. size 256-1024)

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1.0	1.0	-0.95	0.97	0.00	0.01
Royal	0.02	0.01	0.93	0.95	-0.01	0.00
Age	0.41	0.38	0.7	0.69	0.03	-0.02
Food	0.01	0.01	0.02	0.01	0.95	0.97

Andrew Ng

Visualizing word embeddings/ analogies



$$e_{man} - e_{woman} \approx e_{king} - e_{?}$$

MAX:

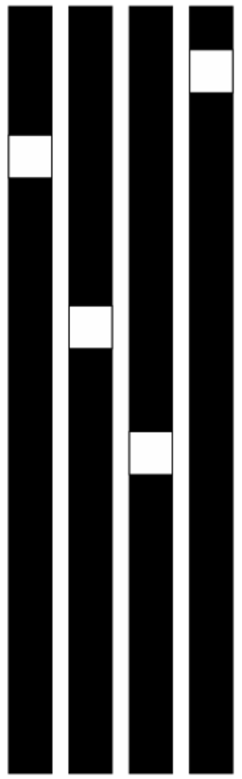
$$sim(e_w, e_{king} - e_{man} + e_{woman})$$

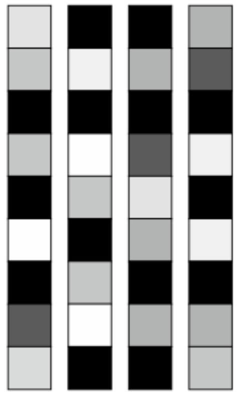
t-SNE

Allows to represent 2D embeddings keeping distances

Andrew Ng

Word embeddings: characteristics


One-hot encoding


Embeddings

Word embeddings are
- Dense representations (vs sparse representations in one-hot encoding)
- Lower dimension
- Continuous values
- **Learned from data**

F. Chollet

Learning embeddings

Embedding – matrix E:

- rows: dimensions of the embedding
- columns: words of the vocabulary

Note that by multiplying the representation of a word as *one-hot encoding* by E we have the column that corresponds to the embedding vector of that same word

Values of this matrix E can be learned as weights in any neural network by gradient descent, being matrix E the first layer, having as inputs one or more words and can have any type of task in subsequent layers

Learning embeddings: example

I	want	a	glass	of	orange	______.
4343	9665	1	3852	6163	6257	

$$\begin{tikzcd}
\text{I} & o_{4343} \arrow[r] & E \arrow[r] & e_{4343} \arrow[ddr] & & \\
\text{want} & o_{9665} \arrow[r] & E \arrow[r] & e_{9665} \arrow[dr] & & \\
\text{a} & o_{1} \arrow[r] & E \arrow[r] & e_{1} \arrow[r] & \textit{softmax layer} \ (10\text{k}) & \text{What's the next word ?} \\
\text{glass} & o_{3852} \arrow[r] & E \arrow[r] & e_{3852} \arrow[ur] & & \\
\text{of} & o_{6163} \arrow[r] & E \arrow[r] & e_{6163} \arrow[uur] & & \\
\text{orange} & o_{6257} \arrow[r] & E \arrow[r] & e_{6257} \arrow[uuur] & &
\end{tikzcd}$$

[Bengio et. al., 2003, A neural probabilistic language model]

Example Embedding - IMDB

```
from tensorflow.keras.datasets import imdb
from tensorflow.keras import preprocessing

max_features = 10000
maxlen = 500
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)

print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)
print(x_train[0])
print(y_train[0])
```

Jupyter Notebook:
ex_imdb_embedding.ipynb

Example Embedding - IMDB

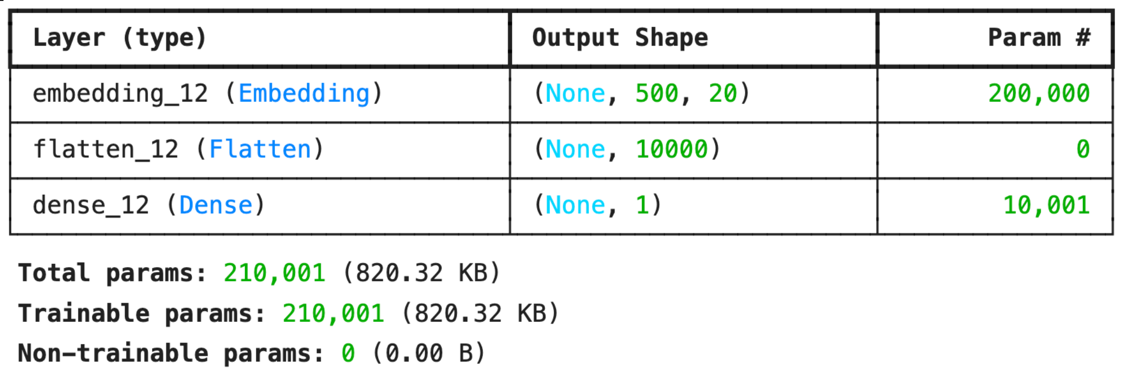
```
dim_embed = 20
model = Sequential()
model.add(Input((maxlen,)))
model.add(Embedding(max_features, dim_embed))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
model.summary()
history = model.fit(x_train, y_train, epochs=5, batch_size=32, validation_split=0.2)
```

Matrix E:

- Vocabulary: 10 k
- Dim of embedding: 20

Layer *embedding*:

- Input: samples x maxlen
- Output: samples x maxlen x 20



```
┃ Layer (type)                ┃ Output Shape          ┃       Param # ┃
│ embedding_12 (Embedding)    │ (None, 500, 20)       │       200,000 │
│ flatten_12 (Flatten)        │ (None, 10000)         │             0 │
│ dense_12 (Dense)            │ (None, 1)             │        10,001 │

Total params: 210,001 (820.32 KB)
Trainable params: 210,001 (820.32 KB)
Non-trainable params: 0 (0.00 B)
```

Example Embedding - IMDB

```
history = model.fit(x_train, y_train, epochs=10,
                    batch_size=32, validation_split=0.2)

results = model.evaluate(x_test, y_test)
print(results)
```

Play with the parameters: max features (number of words), maxlen (length of the review to consider), dimension of the embedding and check how the results change

Embeddings*:* transfer learning

Embeddings can be learned from scratch for each problem (if there is enough data)

In many cases, it is preferable to re-use pre-trained embeddings in extended datasets with large vocabularies – *transfer learning*

If we have enough data, we can always refine embeddings with the available data

Embeddings most used: Word2Vec, Glove

Example with Glove for the iMDB dataset:
https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter11_part02_sequence-models.ipynb

GLOVE embedding available in: http://nlp.stanford.edu/data/glove.6B.zip

Recurrent neural networks

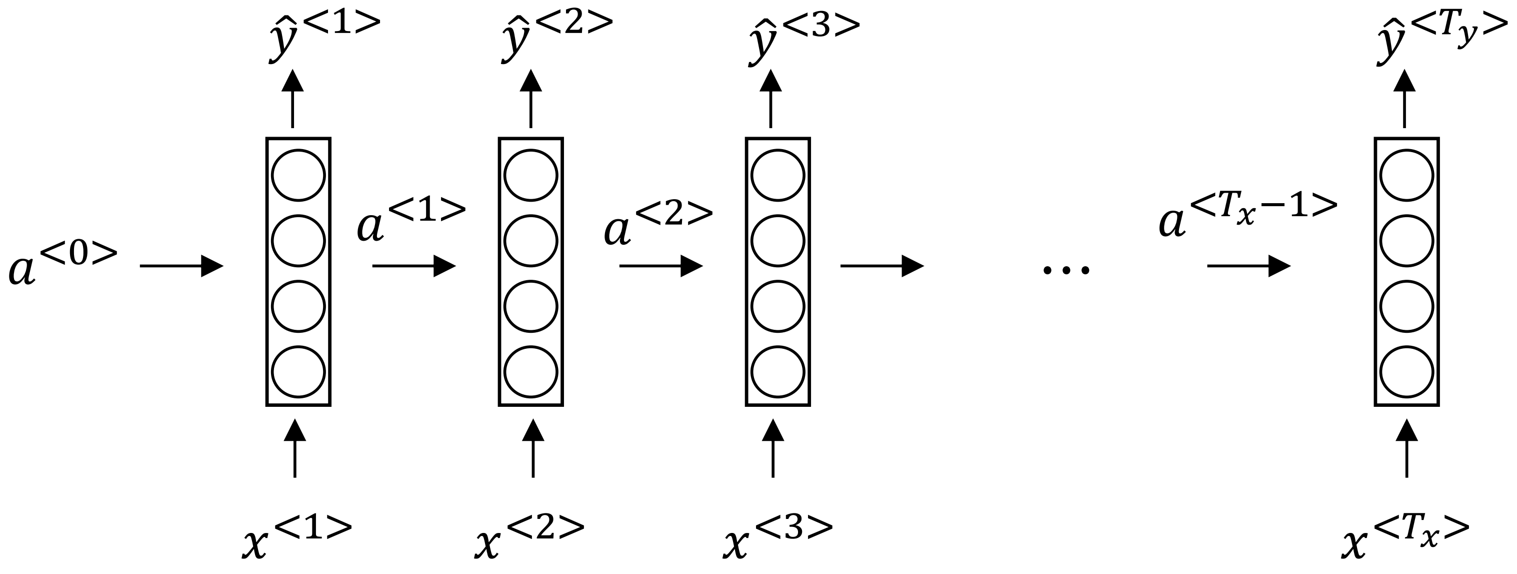
Unlike feedforward networks, recurrent networks have a topology that can contain cycles, with feedback from network outputs to inputs

Thus, they have the ability to process sequential data, and can maintain and can have various types of state / "memory" (of "short" and "long" term)

For example, in the case of IMDB, each review can be processed sequentially by being maintained a state relative to the words already seen (even if each review is analyzed independently)

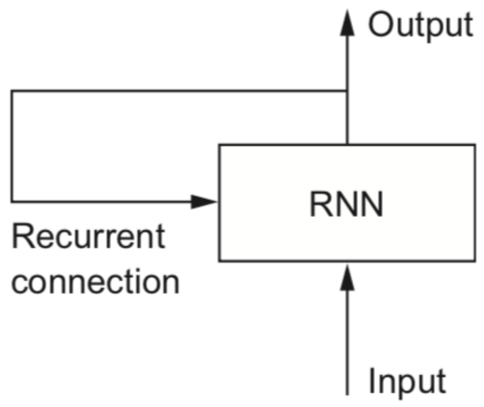
Training algorithms based on gradient descent (e.g Backprop through time) propagate errors through the chain rule from the last state to the initial state

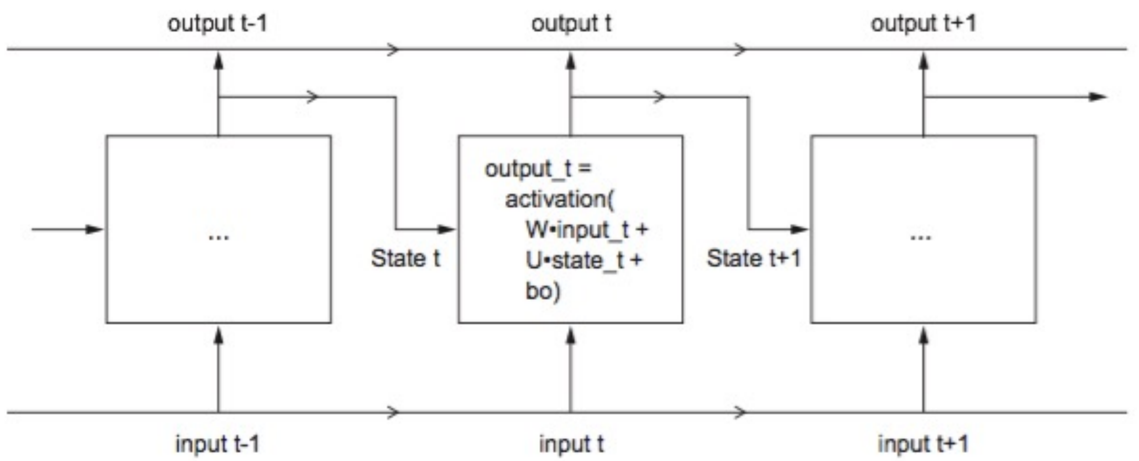
Recurrent neural networks



Calculation of outputs and states →

← Propagation of error





Example of a recurrent network
Layer ***SimpleRNN*** in keras

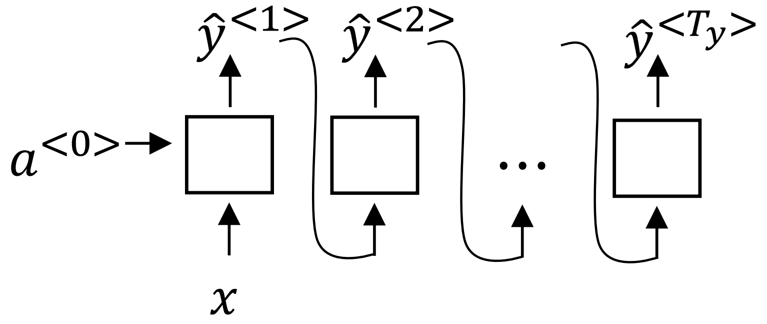
RNN – simple implementation

```
import numpy as np

timesteps = 100
input_features = 32
output_features = 64
inputs = np.random.random((timesteps, input_features))
state_t = np.zeros((output_features,))
W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))

successive_outputs = []
for input_t in inputs:
       output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
       successive_outputs.append(output_t)
       state_t = output_t
final_output_sequence = np.stack(successive_outputs, axis=0)
```

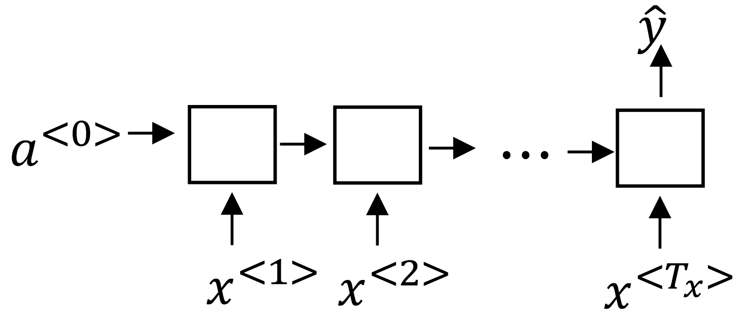
This part would need to be replaced by the real inputs

For a real example, weights would be trained !

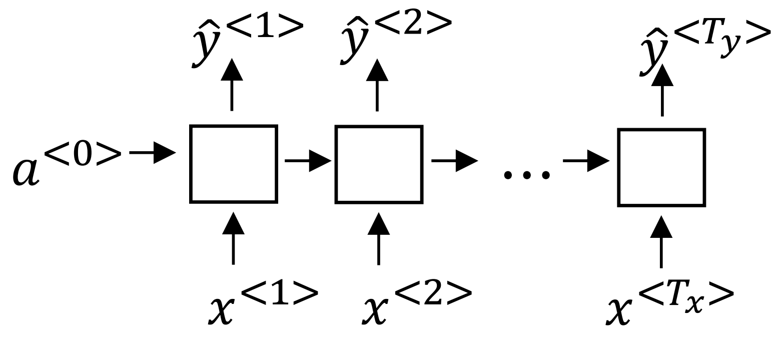
Several types of RNNs



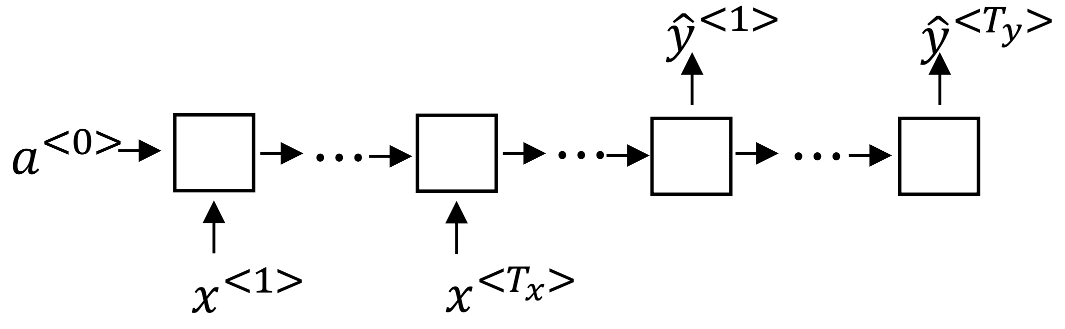
One to many



Many to one



Many to many



Many to many

Andrew Ng

Example RNN- IMDB

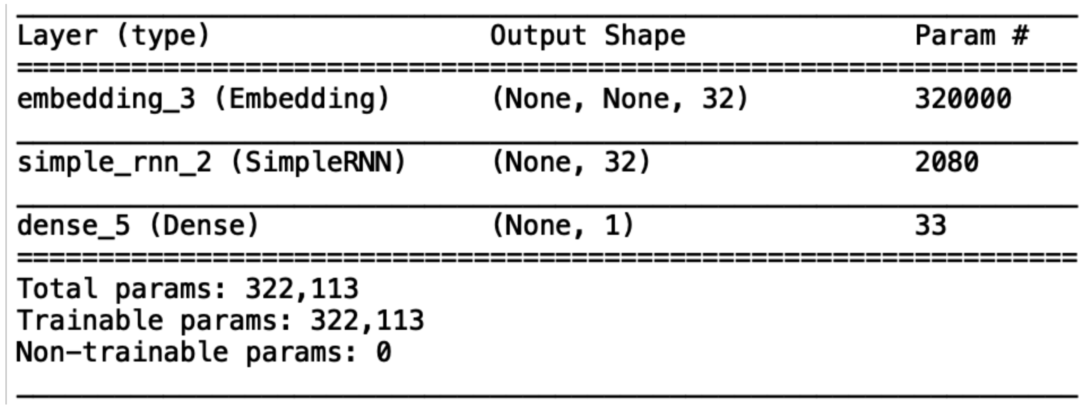
```
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing import sequence

max_features = 10000
maxlen = 500
batch_size = 32

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
print(len(x_train), 'train sequences and ', len(input_test), 'test sequences')
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)

print(x_train.shape, x_test.shape)
```

Jupyter Notebook:
ex_imdb_rnn.ipynb

Example
RNN- IMDB

```
model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

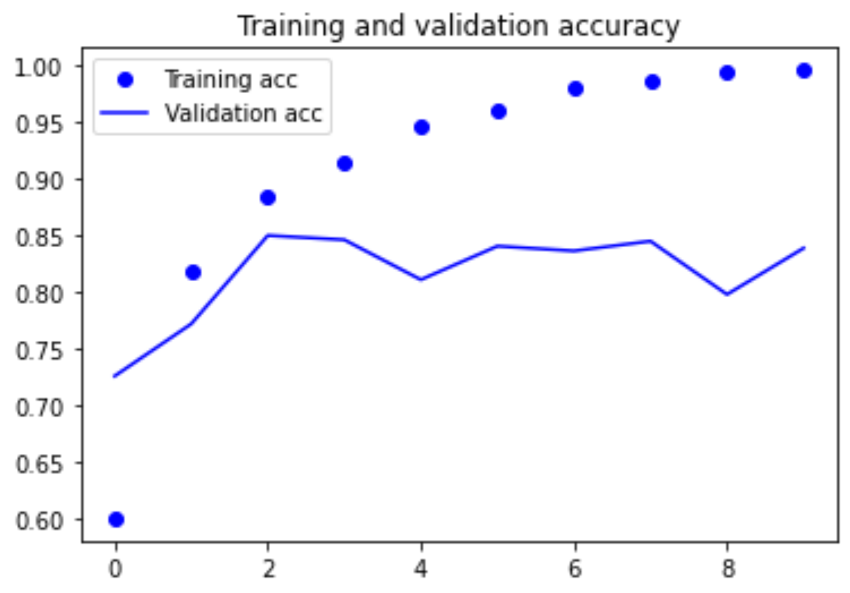
Layer Many to One
It only gives output at the end
Flag "return_sequences"
Used to Many to Many

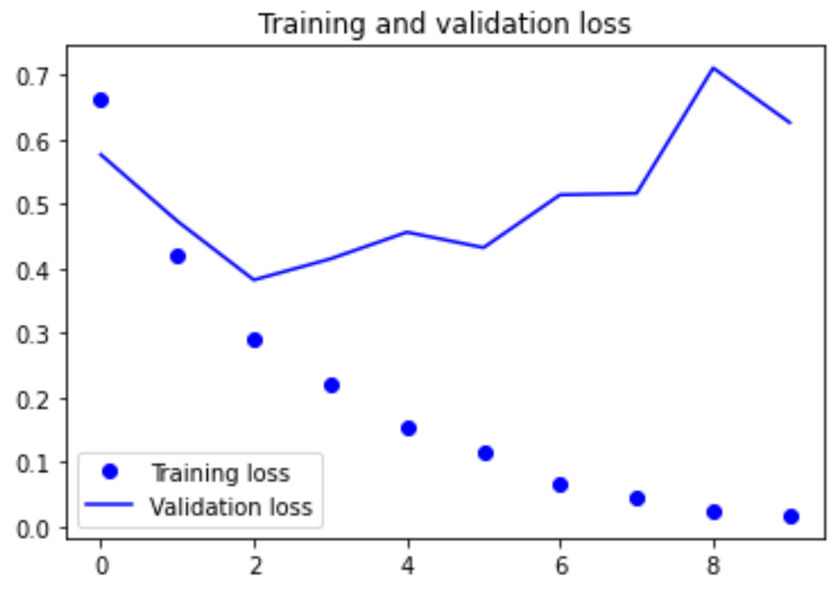


```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, None, 32)          320000
_________________________________________________________________
simple_rnn_2 (SimpleRNN)     (None, 32)                2080
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 33
=================================================================
Total params: 322,113
Trainable params: 322,113
Non-trainable params: 0
_________________________________________________________________
```

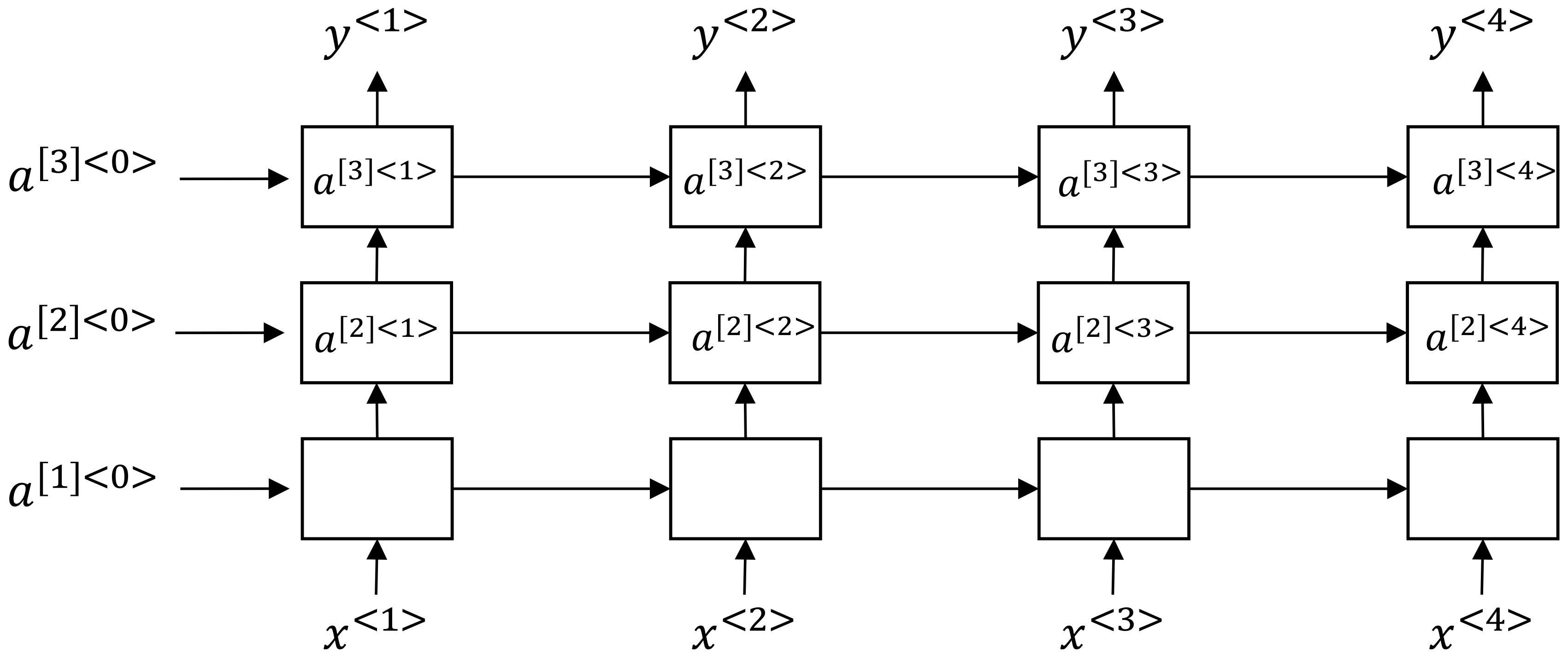
Example
RNN- IMDB

```
model.compile(optimizer='rmsprop',
          loss='binary_crossentropy’,
          metrics=['acc'])

history = model.fit(input_train, y_train,
                    epochs=10, batch_size=128,
                    validation_split=0.2)

model.evaluate(input_test, y_test)
```





RNNs with multiple layers



Andrew Ng

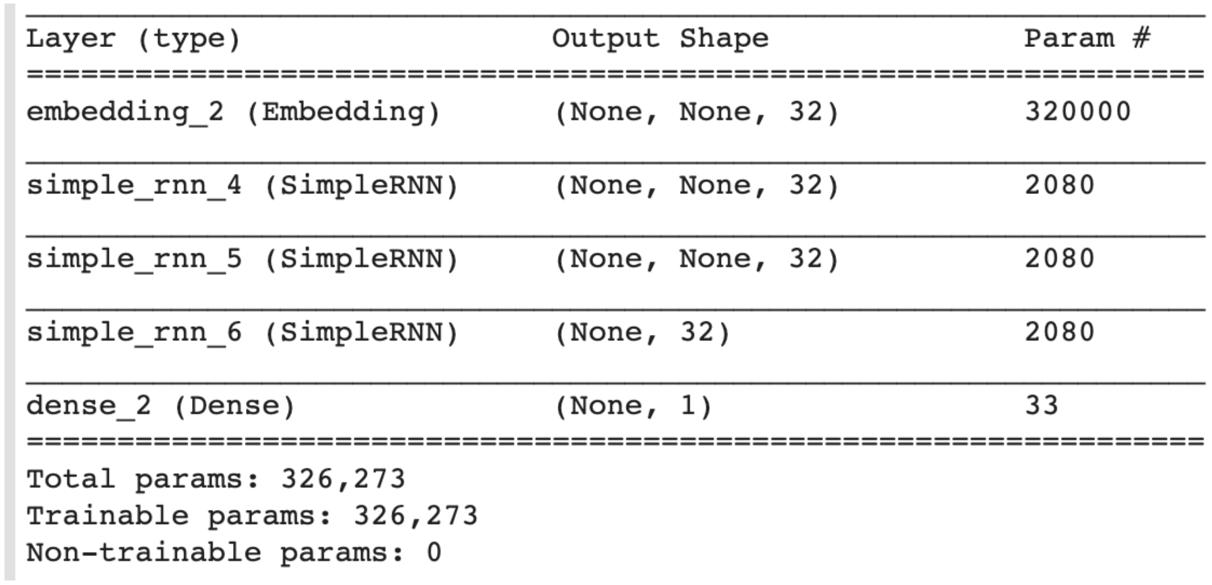
Example RNN- IMDB Multiple layers

```
model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32, return_sequences = True))
model.add(SimpleRNN(32, return_sequences = True))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))
model.summary()

model.compile(optimizer='rmsprop',
        loss='binary_crossentropy', metrics=['acc'])

history = model.fit(input_train, y_train,
                epochs=10, batch_size=128,
                validation_split=0.2)
```

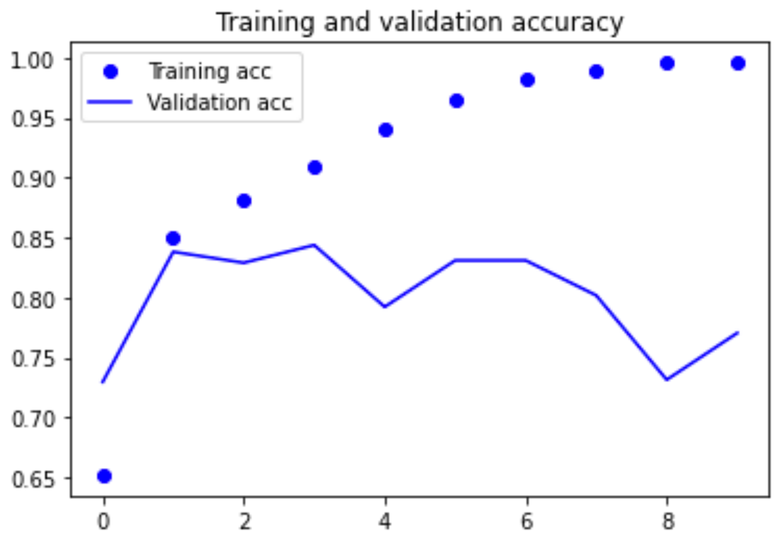
Flag “return_sequences” used to pass outputs for other layers

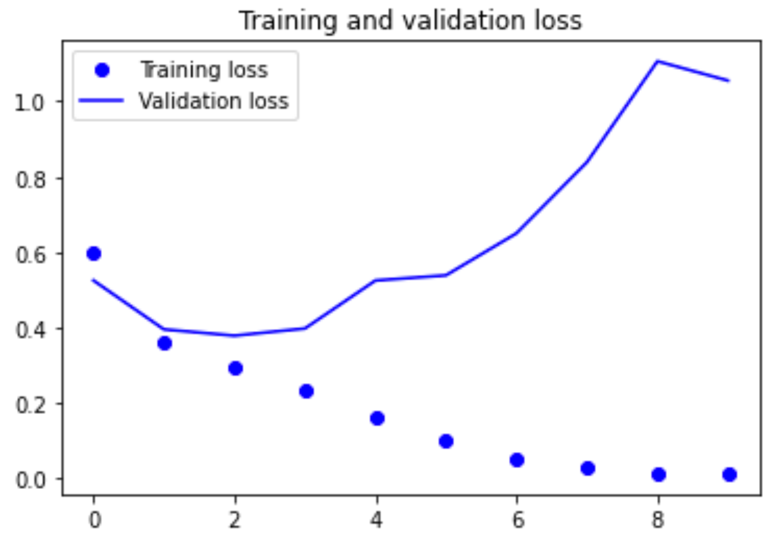
Compare the results with the previous model. What can be done to improve ?

Example RNN multiple layers



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_2 (Embedding)      (None, None, 32)          320000    
_________________________________________________________________
simple_rnn_4 (SimpleRNN)     (None, None, 32)          2080      
_________________________________________________________________
simple_rnn_5 (SimpleRNN)     (None, None, 32)          2080      
_________________________________________________________________
simple_rnn_6 (SimpleRNN)     (None, 32)                2080      
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 33        
=================================================================
Total params: 326,273
Trainable params: 326,273
Non-trainable params: 0
```





Accuracy – test set = 80-81%
Problemas com overfitting

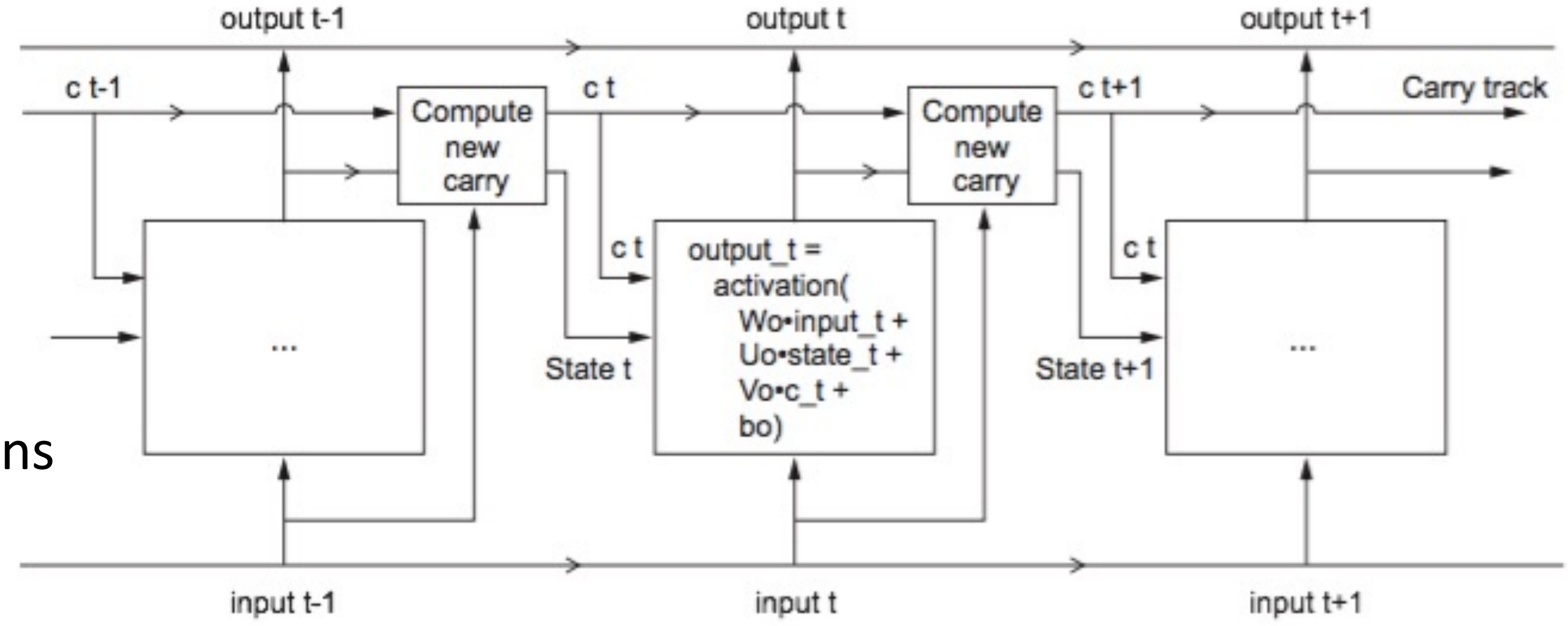
LSTMs

LSTM – Long Short-Term Memory – are recurrent networks that are characterized by maintaining "memory"

They have been introduced to solve the so-called problems of ”vanishing gradients”

c_t – *carry*

Calculated from 3 transformations

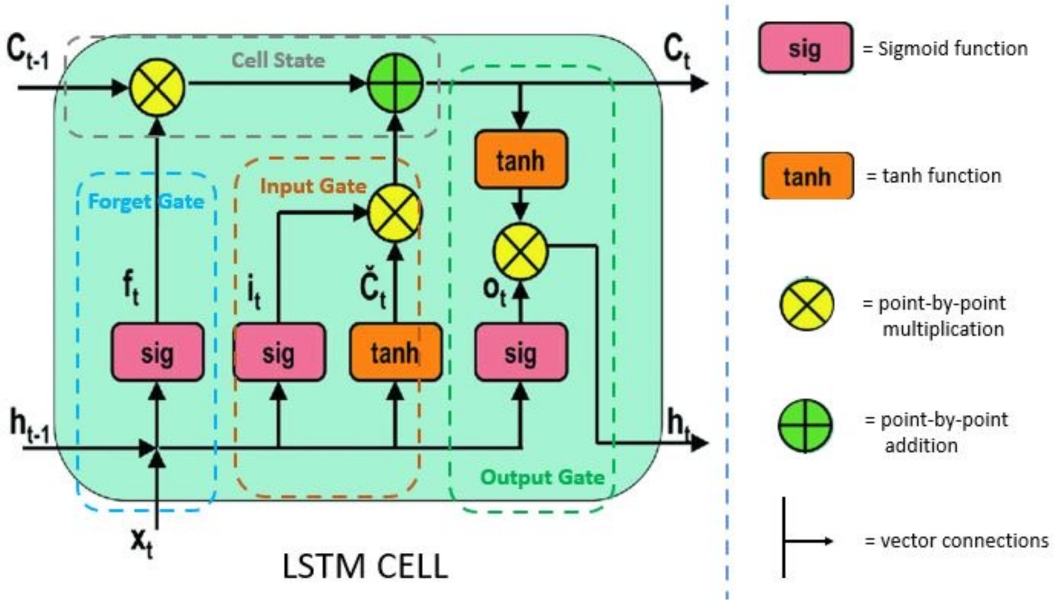


LSTMs - pseudocode

```
output_t = activation(dot(state_t, Uo) + dot(input_t, Wo) + dot(c_t, Vo) + bo)
i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk)
```

```
c_t+1 = i_t * k_t + c_t * f_t
```

Names of the gates (forget, input, …) are merely indicative of their perceived “role”

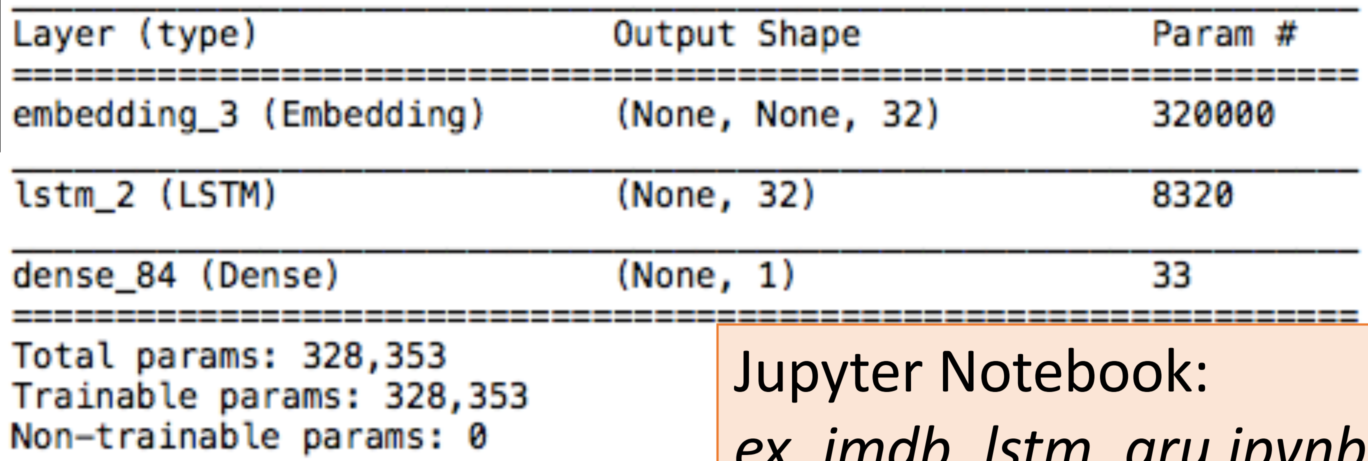


Example LSTM - IMDB

```
model = models.Sequential()
model.add(Embedding(10000, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
          loss='binary_crossentropy',
          metrics=['acc'])

model.summary()
```

LSTM layer can be replaced by GRU:

```
from keras.layers import GRU

...
model.add(GRU(32))
...
```



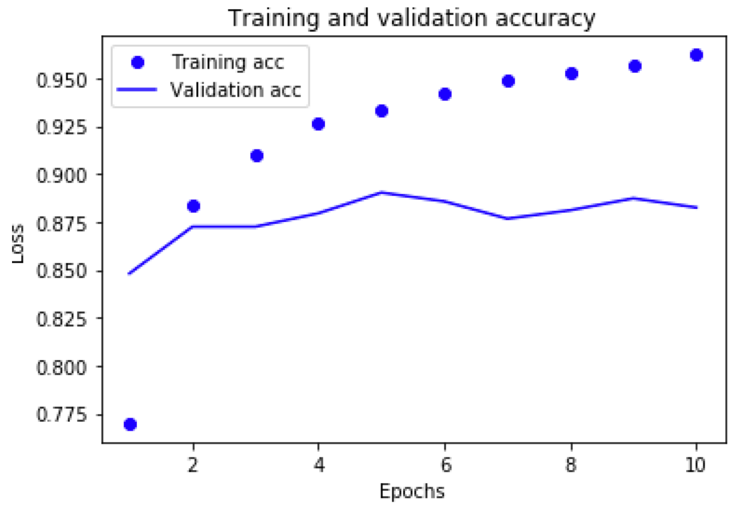
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, None, 32)          320000
_________________________________________________________________
lstm_2 (LSTM)                (None, 32)                8320
_________________________________________________________________
dense_84 (Dense)             (None, 1)                 33
=================================================================
Total params: 328,353
Trainable params: 328,353
Non-trainable params: 0
```

Jupyter Notebook:
ex_imdb_lstm_gru.ipynb

Example LSTM - IMDB

```
history = model.fit(input_train, y_train,
                    epochs=4,
                    batch_size=128,
                    validation_split=0.2)

res = model.evaluate(input_test, y_test)
print(res)
```

Exercises: explore multi-layer LSTM models; explore with GRU layers; explore LSTM with Dropout



Recurrent networks: supplementary notes

LSTM layers can be replaced by GRU layers with similar functionality (simpler – fewer weights to train)

GRU or LSTM layers can be stacked creating models with higher capacity, just as we did for *SimpleRNN* layers

You can consider using Dropout in these layers to address overfitting (check the examples in the notebook for LSTM)

Bidirectional recurrent networks may be used in cases where both the natural order of the sequences and their reverse may make sense – layer Bidirectional from Keras

Recurrent networks: other examples

LSTM networks are widely used in time series forecasting scenarios (see exemple below)

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter10_dl-for-timeseries.ipynb

Multiple classes

In many cases, we are interested in classifying texts in a number of classes larger than 2

One example is provided by the Reuters dataset, where the aim is to classify newswires into 46 mutually exclusive topics

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/chapter04_getting-started-with-neural-networks.ipynb

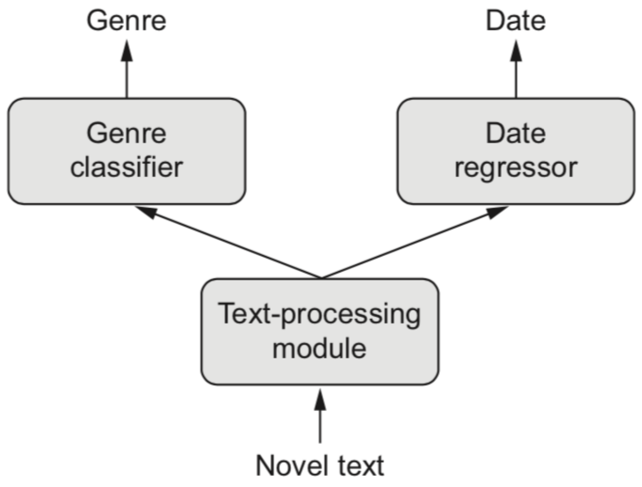
(example with DNNs – feedforward; exercise – implement RNNs/LSTMs)

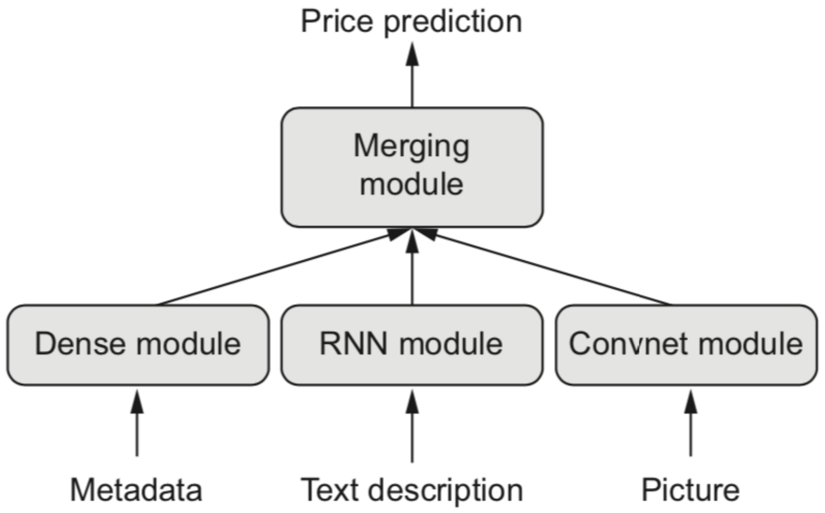
Problems with multiple inputs and outputs

The examples we've seen so far always had a single type of input and an output variable

In many cases, we may have problems in which it makes sense to have entries of various types (textual, images, etc.) and in which we may have several output variables

In such cases, it makes no sense to train individual models because the inputs/outputs are not independent





Keras functional API

Jupyter Notebook:
ex_functionalapi_housing.ipynb

- It allows to define NN architectures in a more flexible way
- NNs are created based on the definition of graphs in a functional way
- Example: a simple DNN and a model with two distinct outputs

```
input_tensor = Input(shape=(train_data.shape[1],))
x = layers.Dense(64, activation='relu')(input_tensor)
x = layers.Dense(64, activation='relu')(x)
output_tensor = layers.Dense(1,activation= "linear")(x)
…
model = Model(input_tensor, output_tensor)
```

Models with multiple outputs - example

```
import numpy as np
train_targets_bin = np.digitize(train_targets,bins=[20])
test_targets_bin = np.digitize(test_targets,bins=[20])

input_tensor = Input(shape=(train_data.shape[1],))
x = layers.Dense(64, activation='relu')(input_tensor)
x = layers.Dense(64, activation='relu')(x)
output_tensor1 = layers.Dense(1,activation= "linear", name= "output1")(x)
output_tensor2 = layers.Dense(1,activation= "sigmoid", name = "output2")(x)
model = Model(input_tensor, [output_tensor1, output_tensor2])
model.compile(optimizer='rmsprop', loss=['mse','binary_crossentropy'], loss_weights =
[0.05, 1])

model.fit(train_data, [train_targets, train_targets_bin], epochs=20, batch_size=16)
print( model.evaluate(test_data, [test_targets, test_targets_bin]) )
```

Dataset Boston Housing
Create binary variable (price above 20 or not)

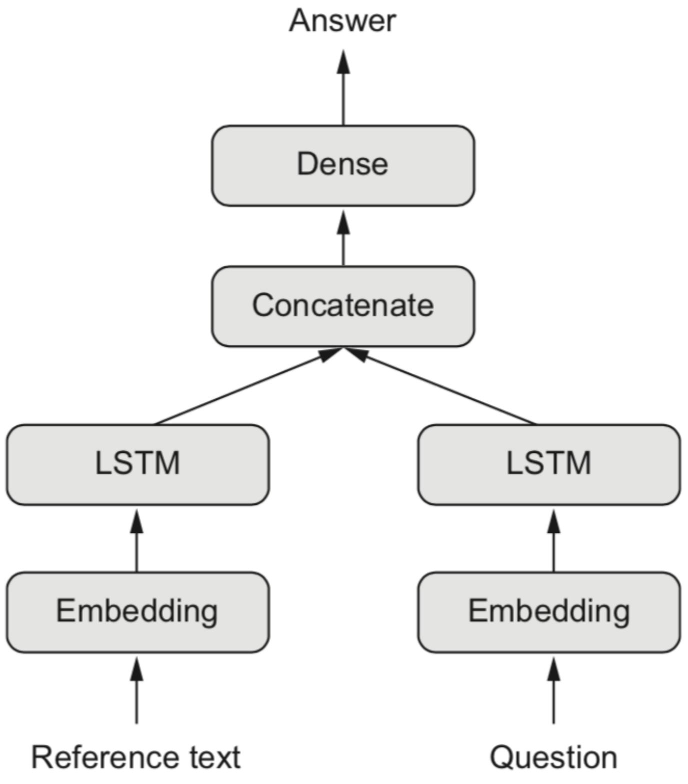
Define model with two output tensors

Define weights for losses of each output

Train and evaluate the model in the test set

Models with multiple inputs

Code is just to exemplify ! Not complete !



```
from tensorflow.keras import layers, input
from tensorflow.keras.models import Model

text_input= Input(shape=(None,), dtype='int32', name='text')

embedded_text = layers.Embedding(64, 10000)(text_input)
encoded_text = layers.LSTM(32)(embedded_text)
quest_input= Input(shape=(None,), dtype='int32', name=‘quest')
embedded_quest = layers.Embedding(64, 10000)(quest_input)
encoded_quest= layers.LSTM(32)(embedded_quest)
concaten = layers.concatenate([encoded_text, encoded_quest],
                                                axis = -1)
answer = layers.Dense(500,activation=“softmax”)(concaten)
model = Model([text_input, question_input], answer)
model.compile (…)
…
model.fit([text_data, quest_data], answers_data, …)
```

Process data for training

Other possibilities

Functional API allows:

- To define complete models as “layers”, used as sub-models for larger models
- Share weights between different layers of the network
- Implement networks with residual connections: summing outputs of distinct layers

Exercise

- Using Google Colab load the python notebooks from F. Chollet book, chapter 11. Change the runtime configuration to use GPUs or TPUs
- Run and analyse the examples
- Design the application of similar approaches for datasets in your practical work