Deep Learning Series

Episode 3 - “La revanche des token”

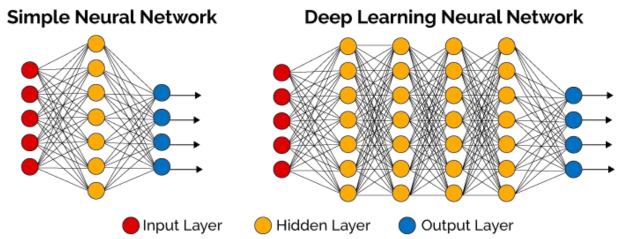
Recap from last episode

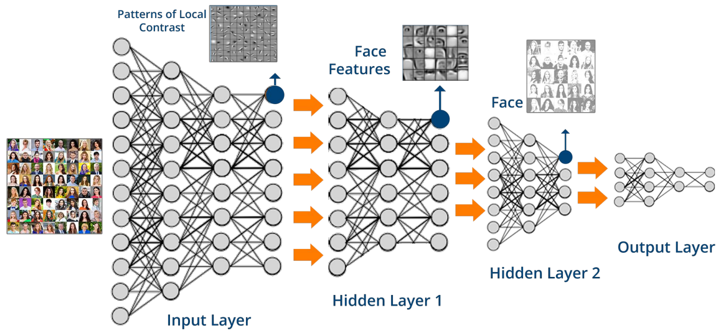
What is Deep Learning ?

Deep Learning is a **subfield of Machine Learning** : A specific way of learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations

Other approaches to **Machine Learning** tend to focus on learning only one or two layers of representations of the data -> Shallow Learning

Deep Learning = Deep Sequence of simple data transformations (layers)



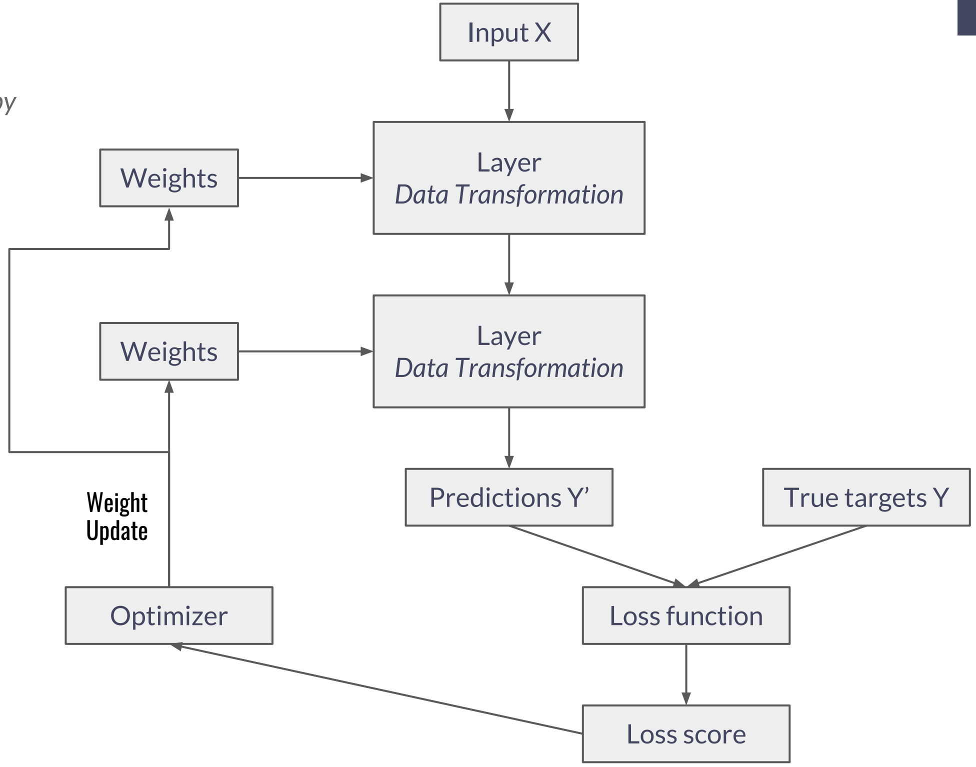


Layered representations of the data are almost always learned via models called ***Neural Networks***

How Deep Learning works ?

The Deep Learning workflow

The transformation implemented by a layer is parameterized by its weights.



Use this score as a feedback signal to adjust the value of the weights a little in order to lower the score for the current example.

This job is done by an optimizer, which implements the Backpropagation algorithm

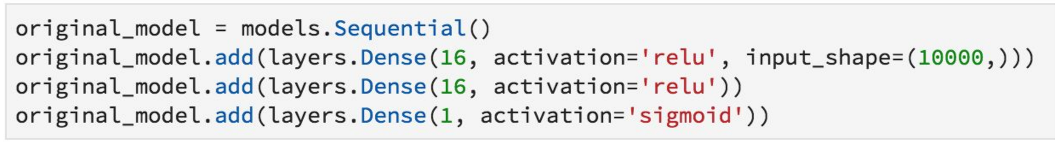
A loss function measures the quality of the network's output.

It computes a distance score between the prediction and the true target

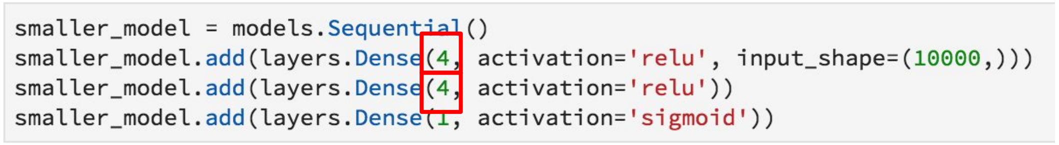
Fighting overfitting : Reducing the network’s size

Simplest way to reduce overfitting : Reducing the number of learnable parameters (=*degrees of freedom*) in the model

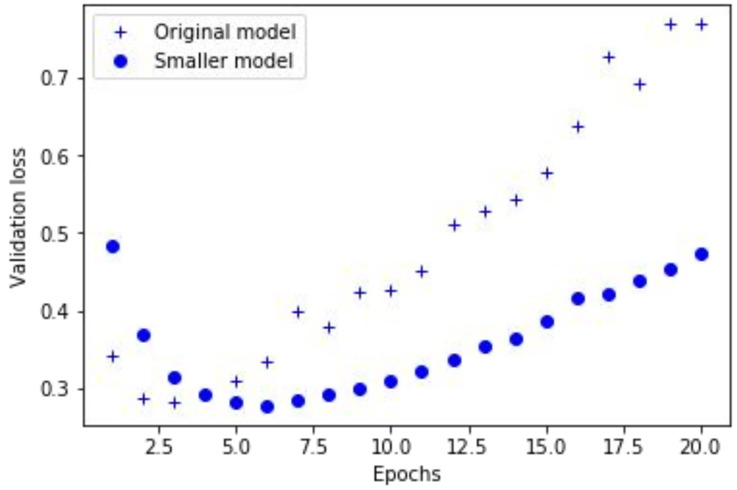
- ➢ A model with more parameters has more memorization capacity
- ➢ With fewer learnable parameters, the model will have to learn **compressed representations** to minimize its loss
- ➢ Find the right amount of parameters by tuning the model’s configuration



```
original_model = models.Sequential()
original_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
original_model.add(layers.Dense(16, activation='relu'))
original_model.add(layers.Dense(1, activation='sigmoid'))
```



```
smaller_model = models.Sequential()
smaller_model.add(layers.Dense(4, activation='relu', input_shape=(10000,)))
smaller_model.add(layers.Dense(4, activation='relu'))
smaller_model.add(layers.Dense(1, activation='sigmoid'))
```



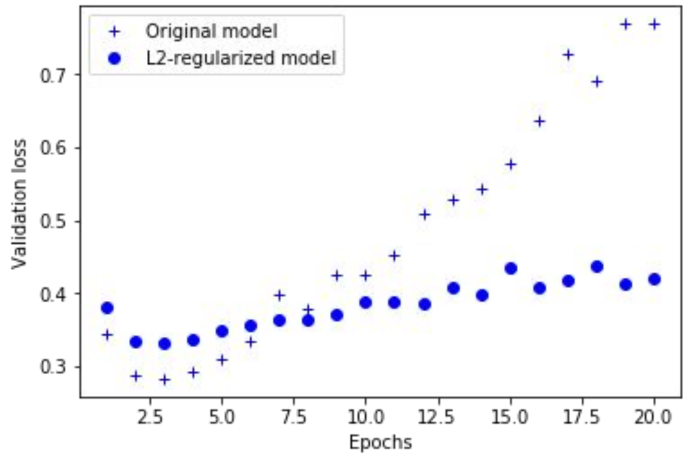
Fighting overfitting : Adding weight regularization

Principle of *Occam's razor* : Given two explanations for something, the explanation most likely to be correct is the simplest one.

- ➢ Simpler models are less likely to overfit that complex ones
- ➢ Simpler model => Forcing its weights to take only small values (more regular distribution of weights)
- ➢ This is done by adding to the lost function a cost associated with having large weights (L1 or L2 regularization)

```
original_model = models.Sequential()
original_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
original_model.add(layers.Dense(16, activation='relu'))
original_model.add(layers.Dense(1, activation='sigmoid'))
```

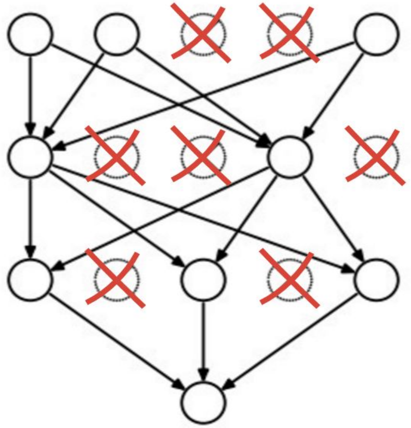
```
l2_model = models.Sequential()
l2_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                          activation='relu', input_shape=(10000,)))
l2_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                          activation='relu'))
l2_model.add(layers.Dense(1, activation='sigmoid'))
```



Fighting overfitting : Adding dropout

Dropout : Randomly dropping out (setting to 0) a number of output features of a given layer during training. The dropout rate is usually between 0.2 and 0.5. Use all outputs at test time.

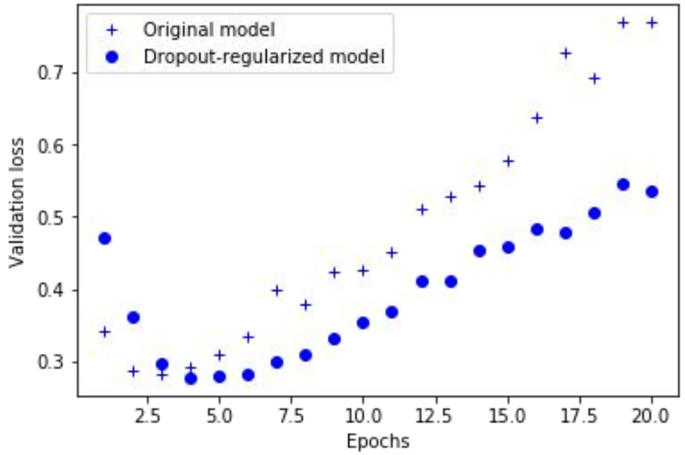
- Core idea : **Introducing noise** in the output values can **break up "*bad luck circumstance patterns*"** in the training data. Information need to be encoded at several places to be relevant.



```
original_model = models.Sequential()
original_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
original_model.add(layers.Dense(16, activation='relu'))
original_model.add(layers.Dense(1, activation='sigmoid'))
```

⇩

```
dpt_model = models.Sequential()
dpt_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(16, activation='relu'))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(1, activation='sigmoid'))
```

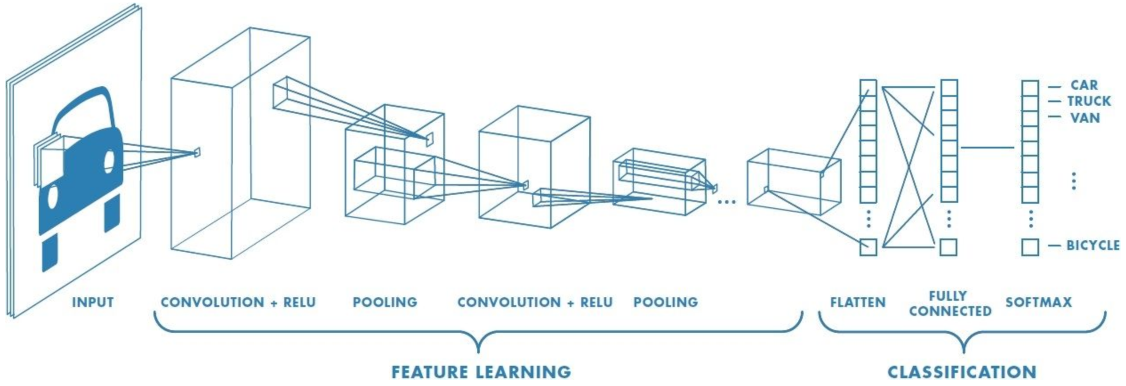


What did we see ?

Understanding Convolutional Neural Networks

Using Data Augmentation

Using pretrained networks



Source : https://www.mathworks.com/discovery/convolutional-neural-network.html

Pretrained Network for Feature Extraction

- Convolutional Neural Networks are the best type of machine learning models for computer vision tasks
  - It is possible to train them from scratch even on very small datasets
- On a small dataset, overfitting will be the main issue. Data augmentation is a powerful way to fight it for image data
- It's easy to reuse an existing convnet that was trained on a larger dataset for Feature Extraction
- Fine-tuning adapts to a new problem some of the representations previously learned by an existing model
  - This pushes performances a bit further

Deep Learning for text and sequences

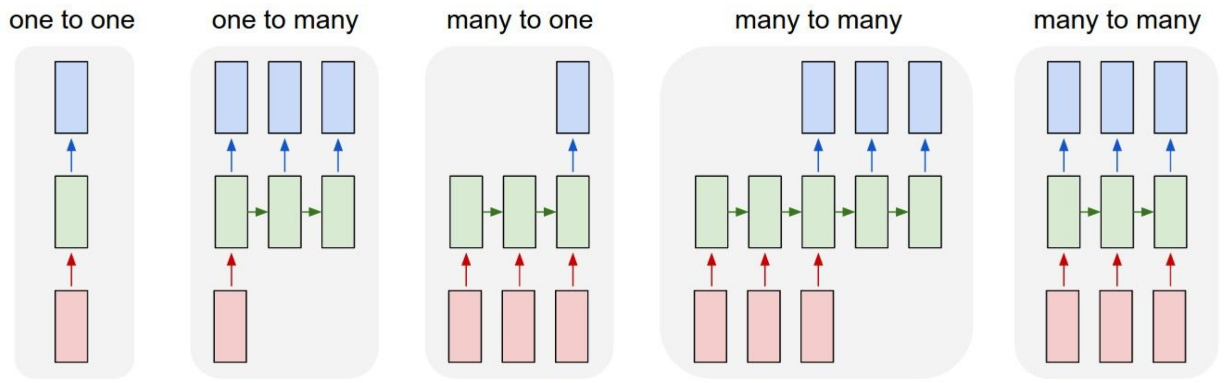
Embeddings & Recurrent Neural Networks

What will we see ?

Embeddings for text data

Recurrent Neural Networks

1D Convolutional Neural Networks



Source : http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Quelles applications ?

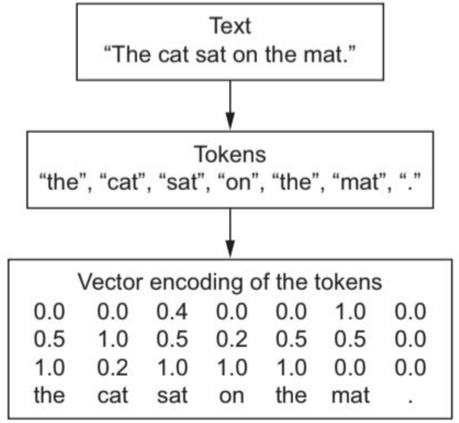
- Document and time series classification
  - *Ex : Identifying the topic of an article, the author of a book*
- Time series comparisons
- Sequence-to-Sequence learning
  - *Ex : Decoding an English sentence into French*
- Sentiment Analysis
- Time series forecasting

Working with Text Data

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Vectorizing text

- Deep Learning models don't take raw text as input, they only work with numerical tensors
    - We need to convert sentences into sequences of vectors
    - 1 token (1 word) = 1 vector
- How to associate a vector with a token ?
    - One-Hot encoding : Sparse vectors
    - Token Embedding : Dense, shorter vectors

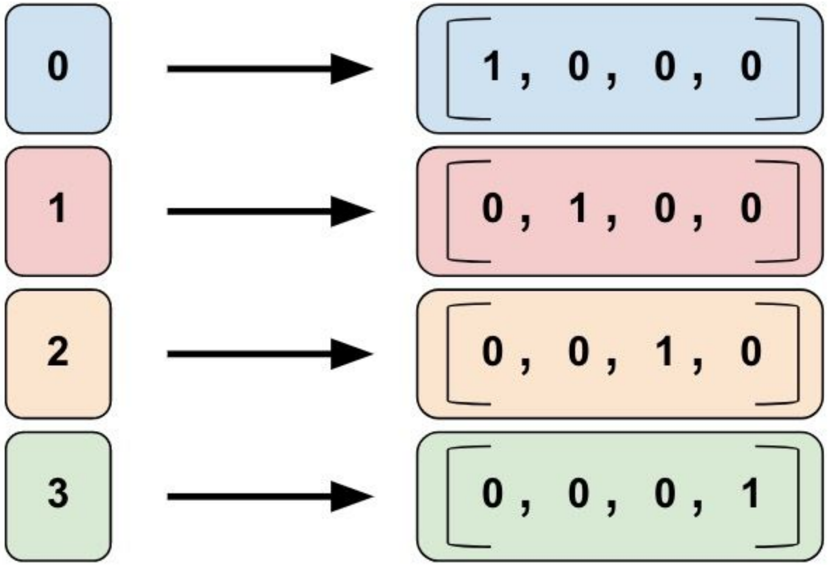


Source : Deep Learning with Python, François Chollet

One-Hot encoding

- One of the most common way to turn a token into a vector
- How to create the vectors ?
  - Associate a unique integer index with every word (*N* words)
  - Turn this integer index *i* into a binary vector of size *N*, which is all-zeros except for the *ith* entry, which is 1

- Variant : One-Hot hashing trick
  - When your vocabulary size is too large
  - Hash words into vectors of fixed size
  - But susceptible to ***hash collisions***

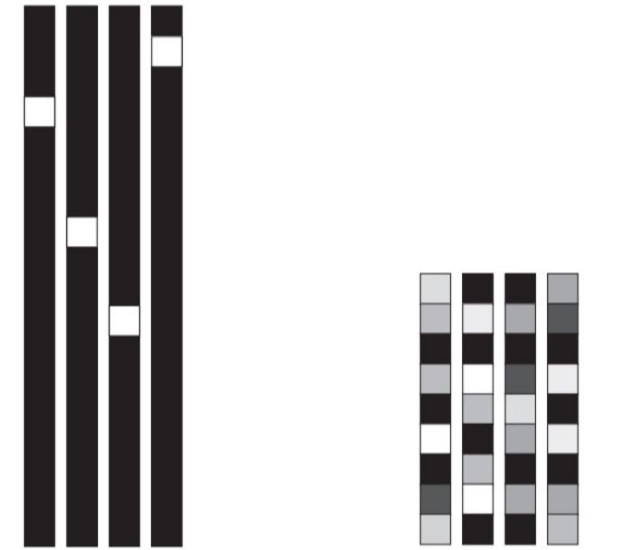


Source : https://www.tensorflow.org/get_started/feature_columns

Word Embeddings

- Word Embeddings are low-dimensional floating-point vectors
  - Common sizes : 256, 512, 1024
- They are learned from data

- Two ways to obtain word embeddings :
  - Learn embeddings **jointly with the main task** you care about
    - Start with random word vectors and learn them through back-propagation
  - Load a **pre-computed word embeddings** learned using a different machine learning task
    - Pre-trained word embeddings

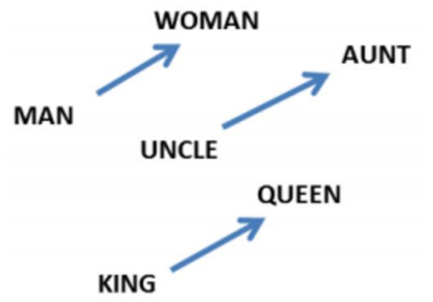


Source : Deep Learning with Python, François Chollet

How to interpret word embeddings ?

- The geometric relationships between word vectors should reflect the semantic relationships between these words
  - Word embeddings are meant to map human language into a geometric space

- We would expect the geometric distance between two semantically related words to be short
- We also want specific directions in the embedding space to be meaningful
  - *Ex : "Female vector" to convert king into queen*

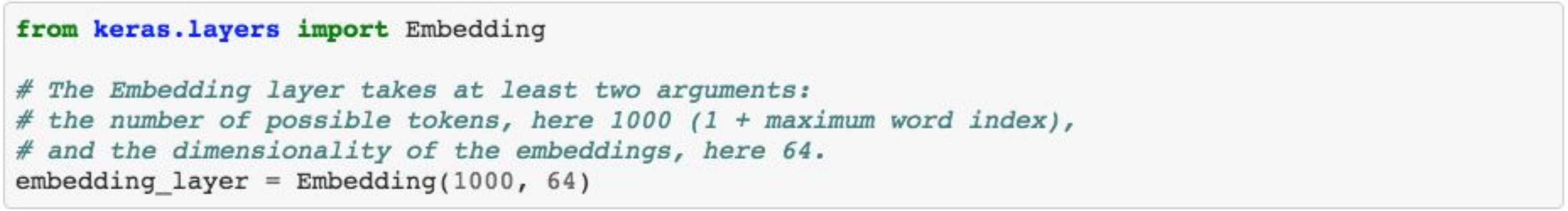


From *Mikolov et al.* (2013a)

$V(King) - V(Man) + V(Woman) \approx V(Queen)$

Learning word embeddings with the Embedding Layer

- There is **no ideal word embedding space** that would perfectly map human language and could be used for any task
- Need to **learn a new embedding space** with every new task
- There is a special layer for that in Keras : the Embedding layer



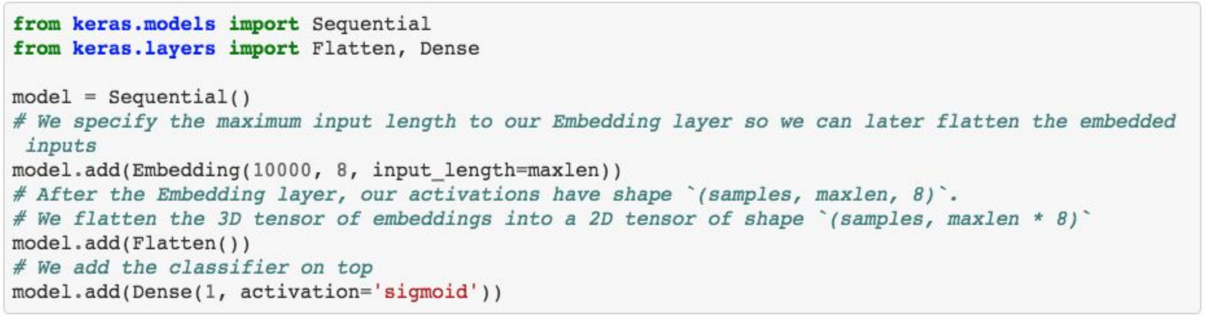
```
from keras.layers import Embedding

# The Embedding layer takes at least two arguments:
# the number of possible tokens, here 1000 (1 + maximum word index),
# and the dimensionality of the embeddings, here 64.
embedding_layer = Embedding(1000, 64)
```

- The *Embedding* layer can be seen as a dictionary that maps integer indices to dense vectors
  - It takes integers as input
  - Returns the associated vector

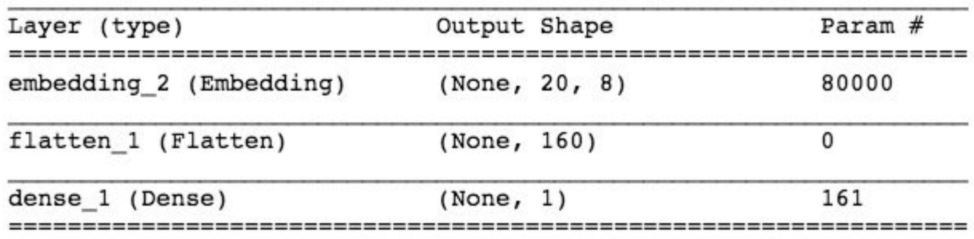
- **Input** : 2D tensor of shape (*samples, sequence_length*)
- **Output** : 3D tensor of shape (*samples, sequence_length, embedding_dimensionality*)

Learning word embeddings with the Embedding Layer

➢ *Example for sentiment prediction*



```python
from keras.models import Sequential
from keras.layers import Flatten, Dense

model = Sequential()
# We specify the maximum input length to our Embedding layer so we can later flatten the embedded
 inputs
model.add(Embedding(10000, 8, input_length=maxlen))
# After the Embedding layer, our activations have shape `(samples, maxlen, 8)`.
# We flatten the 3D tensor of embeddings into a 2D tensor of shape `(samples, maxlen * 8)`
model.add(Flatten())
# We add the classifier on top
model.add(Dense(1, activation='sigmoid'))
```



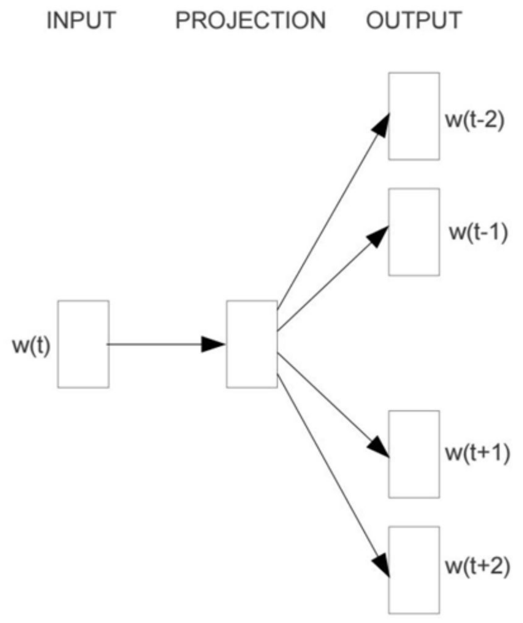
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 20, 8)             80000
_________________________________________________________________
flatten_1 (Flatten)          (None, 160)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 161
=================================================================
```

By flattening the embedded sequences, the model treats each word in the input sequences separately, without considering inter-word relationships and sentence structure.

Using pre-trained word embeddings

- When not enough data is available, it is hard to learn meaningful word representations from scratch
- Instead of learning word embeddings jointly with the main task, you can load embedding vectors from a pre-computed embedding space
    - Which is highly structured
    - That capture generic aspects of language structure
- It is the same approach as using pre-trained convnets in image classification

- Such embeddings are generally computed using word-occurrence statistics
    - They are trained with text corpus like Wikipedia



Source : http://rohanvarma.me/Word2Vec/

Using pre-trained word embeddings

- *Example for sentiment prediction*

```python
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

- The *Embedding* layer has a single weight matrix : a 2D float matrix
  - Each entry *i* is the word vector associated with index *i*

```python
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 100, 100)          1000000
_________________________________________________________________
flatten_2 (Flatten)          (None, 10000)             0
_________________________________________________________________
dense_2 (Dense)              (None, 32)                320032
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 33
=================================================================
Total params: 1,320,065
Trainable params: 320,065
Non-trainable params: 1,000,000
_________________________________________________________________
```

Don't forget to precise the first layer to be not trainable.
The pre-trained parts should not be updated during training, to avoid forgetting what we already know.

Understanding Recurrent Neural Networks

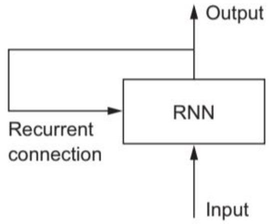
Don’t forget the past

Intuition

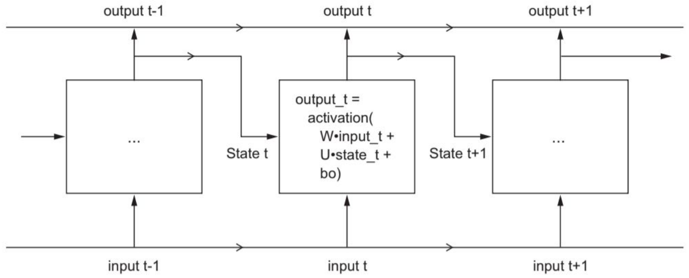
- Densely connected networks and convnets have no memory : each input is processed independently
  - To process sequence or time series data, you need to show the entire sequence to the network at once
- Humans process a sequence word by word, while keeping memories of what came before

- A Recurrent Neural Network adopts the same principle, in a very simplified version
  - Processes a sequence by iterating through it and maintaining a state containing past information
  - It's a neural network with an internal loop

```
state_t = 0
for input_t in input_sequence:
        output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
        state_t = output_t
```



Recurrent Neural Network : A network with a loop



A simple RNN, unrolled over time

Source : Deep Learning with Python, François Chollet

Recurrent Layer in Keras

➢ This simple process is present in Keras as a SimpleRNN layer

```
from keras.layers import Dense

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

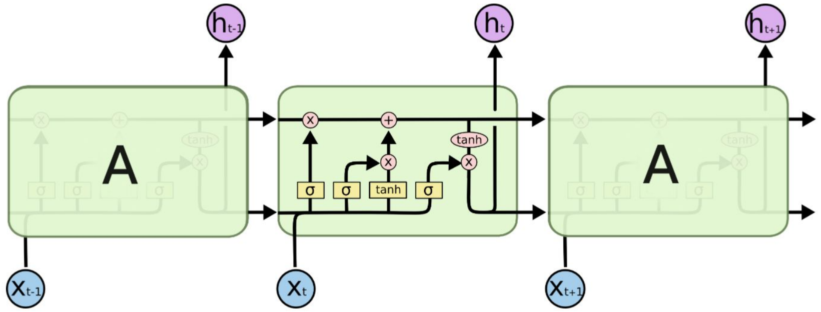
➢ This layer can be run in two modes
  - Return the full sequences of successive outputs for each timestep
  - Return only the last output for each input sequence

➢ These two modes are controlled by the *return_sequences* constructor argument

If you stack several recurrent layers one after the other in order to increase the representational power of the network, all intermediate layers must return full sequences of outputs.

Understanding the LSTM and GRU layers

- In practice, one never uses the SimpleRNN layer
  - They are two simplistic to be of real use
- Major issue : it has high difficulties to learn long-term dependencies
  - Due to the vanishing gradient problem
- LSTM and GRU layers are designed to solve this problem
  - LSTM : Long Short-Term Memory
  - GRU : Gated Recurrent Unit
- LSTM
  - Adds a way to carry information across many timesteps
  - Information from a timestep can be transported into another timestep
- GRU
  - Use the same principle as LSTM
  - Cheaper to run but less representational power



The repeating module in an LSTM contains four interacting layers.

```
from keras.layers import LSTM

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

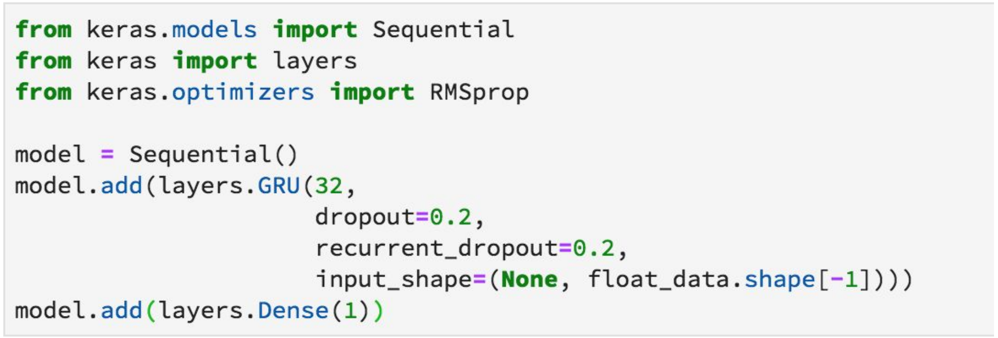
Source : http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Advanced use of RNNs

Towards infinity, and beyond

Recurrent dropout to fight overfitting

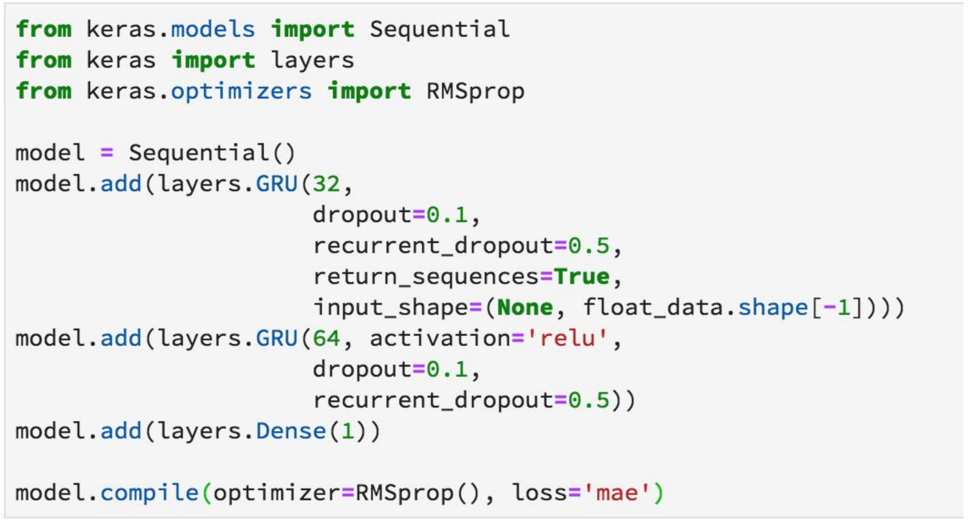
- Like with densely connected and convolutional neural networks, dropout can be used for Recurrent Neural Networks to reduce overfitting
  - The **same dropout mask** should be applied at every timestep
  - Allows to properly propagate its learning error through time
- In Keras, there are two dropout-related arguments
  - *dropout* : dropout rate for input units
  - *recurrent_dropout* : dropout rate of the recurrent units



```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32,
                     dropout=0.2,
                     recurrent_dropout=0.2,
                     input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))
```

Stacking recurrent layers

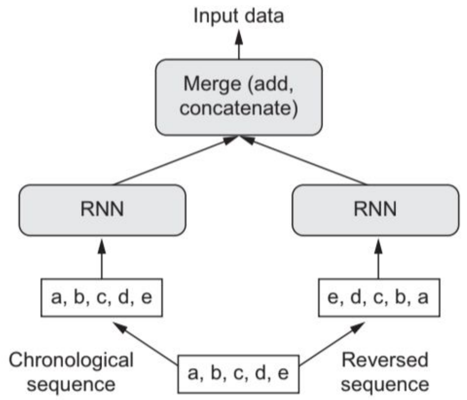
- Stacking recurrent layers helps increasing the representational capacity of the network
  - What currently power the Google Translate algorithm is a stack of seven large LSTM layers
- To stack recurrent layers on top of each other in Keras, all intermediate layers should return their full sequence of outputs rather than their output at the last timestep
  - This is done by specifying *return_sequences=True*



```python
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32,
                     dropout=0.1,
                     recurrent_dropout=0.5,
                     return_sequences=True,
                     input_shape=(None, float_data.shape[-1])))
model.add(layers.GRU(64, activation='relu',
                     dropout=0.1,
                     recurrent_dropout=0.5))
model.add(layers.Dense(1))

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

Bidirectional Recurrent Neural Networks

- Variant of the common RNN that is frequently used in natural-language processing
- It exploits the order-sensitivity of RNNs
- Consists of using two regular RNNs (LSTM or GRU), each of which processes the input sequence in one direction
  - Chronologically and anti-chronologically
  - Then merges their representations
- By processing a sequence both ways, it can catch patterns that may be overlooked by an unidirectional RNN



```
model = Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.Bidirectional(layers.LSTM(32)))
model.add(layers.Dense(1, activation='sigmoid'))

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

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

Source : Deep Learning with Python, François Chollet

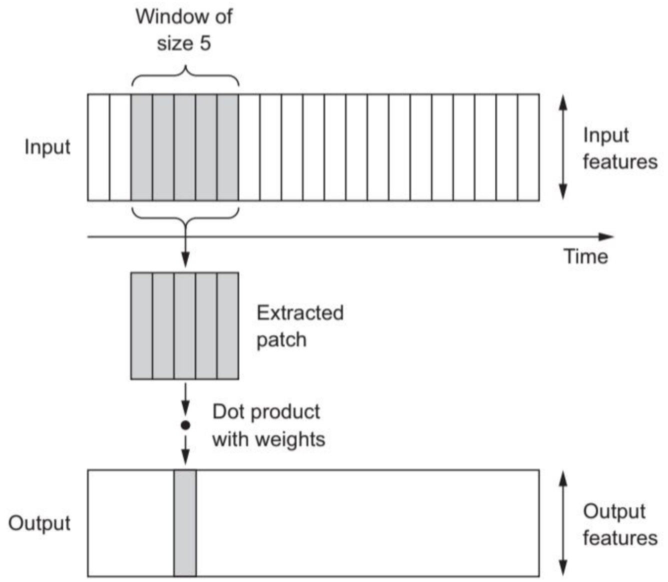
Sequence processing with convnets

What ? Convnets ? For sequence data ??

Understanding 1D convolution for sequence data

- Convolutional Neural Networks perform particularly well on image data
  - They are composed of 2D convolutions
- The same principle can be applied for sequence data with 1D convolutions to extract local 1D patches from sequences
- Such layers can recognize local patterns in a sequence
  - Translation invariant since the same input transformation is performed on every patch
- 2D pooling was used to spatially downsample image tensors
  - We can use the 1D equivalent : 1D pooling

```
model = Sequential()
model.add(layers.Embedding(max_features, 128, input_length=max_len))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
```

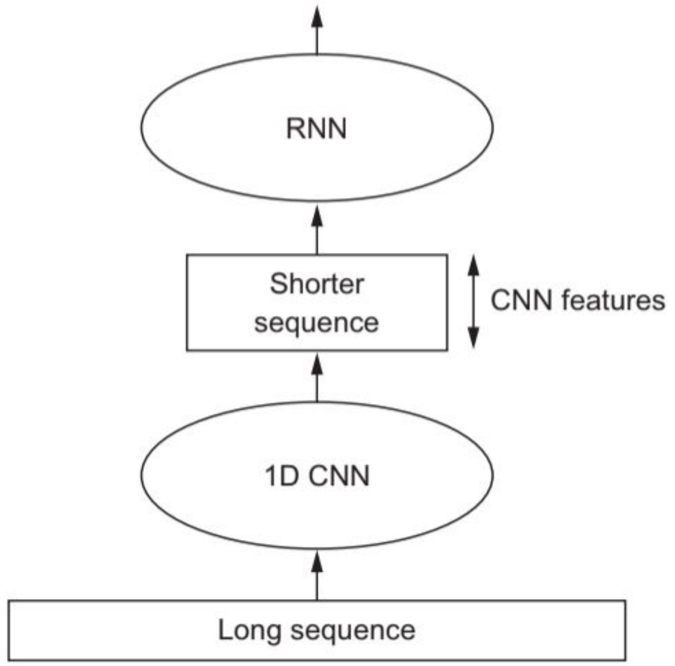


Source : Deep Learning with Python, François Chollet

Combining CNNs and RNNs to process long sequences

- 1D convnets process input patches independently
  - They aren't sensitive to the order of timesteps
  - It often fails at giving meaningful results in cases when the order counts
  - But it is cheaper to train than RNNs
- RNNs have other characteristics
  - Longer to train, but can catch temporal dependencies
  - Still difficult to train on very long sentences

- One solution : Combining CNNs and RNNs
  - Use a 1D-convnet as a preprocessing step before an RNN
  - The convnet will downsample the sequence



Source : Deep Learning with Python, François Chollet

Take aways

What did we learn ?

What did we learn ?

- Techniques for working with sequence data
  - Tokenize text
  - Embeddings
  - RNNs : SimpleRNN, LSTM, GRU
- Increase performances of RNNs by using dropout, stacking layers and using bidirectional RNNs
- Use 1D convnets and combine them with RNNs

- If global order matters (ex: time series), it’s preferable to use a RNN
- If global order isn’t fundamentally meaningful, using a 1D convnet are cheaper and work just as well