P0: Introduction to Keras (Part I: formative assignment)

Objectives

- Introduce Keras and programming environments for Python notebooks
- Get acquainted with basic libraries useful for NLU (preprocessing, tokenization, embeddings, etc.)
- Be able to train and use neural network models for classification problems in a text domain

notebook file version: 25/26 v0.1

Python and programming environments

↳ 18 celdas ocultas

Introduction to Keras

Keras is a Python Deep Learning API, running on top of the machine learning platform TensorFlow. It was developed with a focus on enabling fast experimentation. It is intended to be simple, flexible and powerful, since it is a model-level library with high-level building blocks for developing deep-learning models:

- user-friendly API
- same code to run seamlessly on CPU or GPU
- built-in support for convolutional networks, recurrent networks (for sequence processing)...

At any time, you may consider checking the Keras API reference, Keras guides and existing cheat sheets to learning about parameters and available functions. You may need to follow the installation instructions to set up Keras in your Python environment.

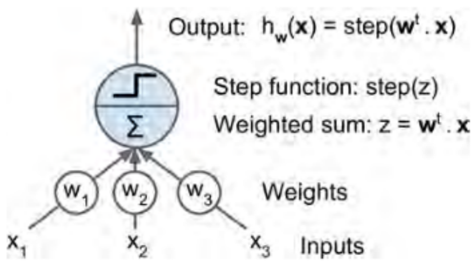
Preliminary concepts

Before we can work with Keras, it is important to remember at a high-level what the perceptron units and artificial neural networks (ANNs) look like.

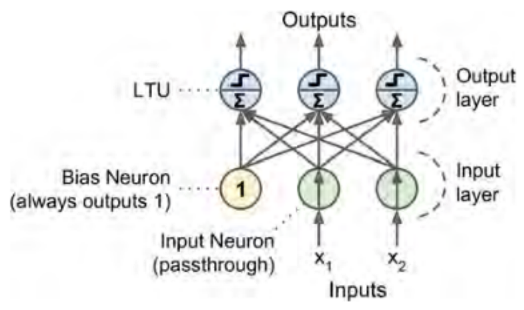
Notice that you will get to know (mathematically and in depth) about ANNs and advanced variants in other subjects (e.g., Machine Learning I, Machine Learning II, Deep Learning). Thus, the concise and brief descriptions included here are just intended to keep you moving on during these first weeks while being allowed to consult and ask for complementary materials. Some introductory books that you may find useful are:

- D. Jurafsky, J.H. Martin. Speech and Language Processing, Draft 2023. Chapters 7-8
- Y. Goldberg. Neural Network Methods for Natural Language Processing. Synthesis Lectures on Human Language Technologies. Springer. 2017. **Chapters 1-5**
- A. Géron. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition

Neural networks are composed of interconnected units. The most basic unit is called a Linear Threshold Unit, and it basically represents a function like this:


Output: $h_w(\mathbf{x}) = \text{step}(\mathbf{w}^t . \mathbf{x})$
Step function: step(z)
Weighted sum: $z = \mathbf{w}^t . \mathbf{x}$
Σ
w_1
w_2
w_3
Weights
x_1
x_2
x_3
Inputs

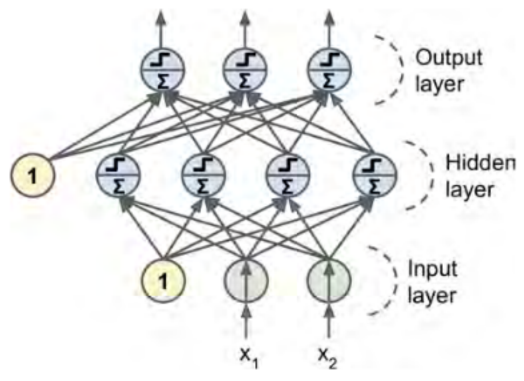
Some of these units can share inputs and be set in a row to obtain several outputs. This is what is known as the Perceptron.



Finally, we may include hidden-layers between the input and the output layers; this Multi-layer Perceptron. You are alredy studying some of these concepts later in Machine Learning I.

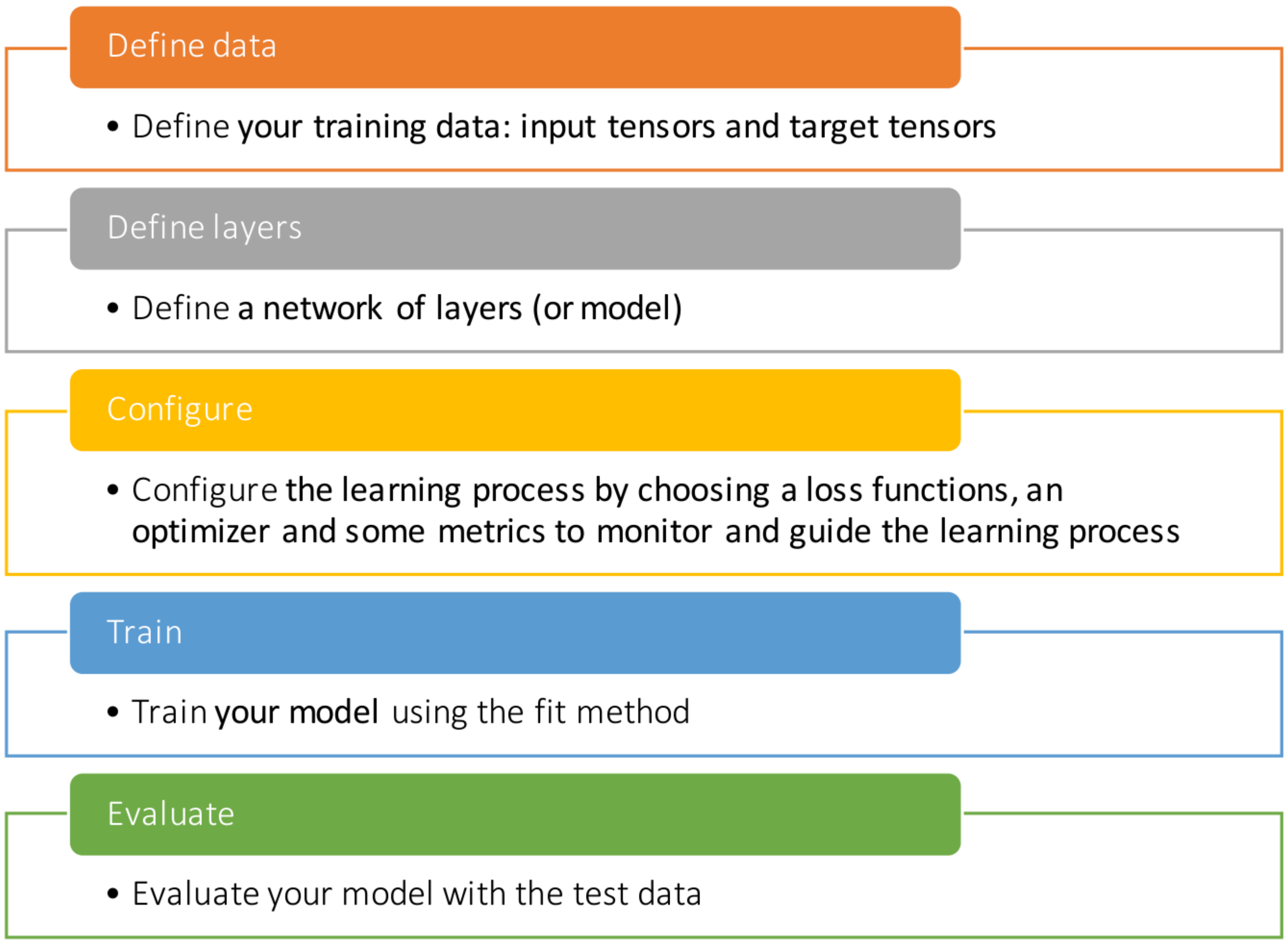
For now, the important point at the moment is to grasp that an ANN will somehow have several stacked layers made of units, and distinguish that there will be an input layer, hidden layers, and at least an output layer.

Such building blocks will be used in Keras as you will be shown in the following examples. Let us oversimplify a little bit, saying that, in Machine Learning the goal is to automatically learn the weights (w) by using algorithms based on Gradient Descent such as Backpropagation, provided there exists a dataset of inputs-outputs. Keras will allow you to define the ANN arquitecture, establish the training algorithm and feed it with the necessary data to adjust the model.



Keras workflow

Before working with actual Keras code, let's establish the typical **workflow** and related key conceptual steps you will need to consider.



Define data

- Define your training data: input tensors and target tensors

Define layers

- Define a network of layers (or model)

Configure

- Configure the learning process by choosing a loss functions, an optimizer and some metrics to monitor and guide the learning process

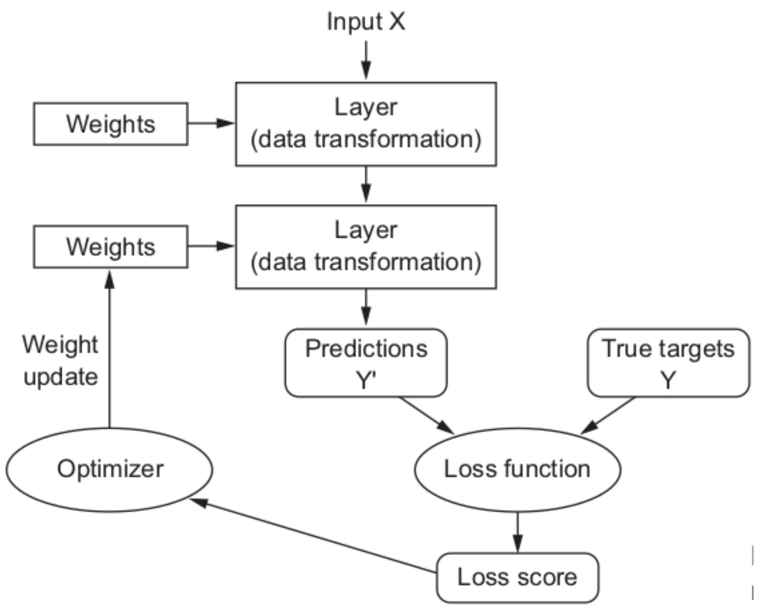
Train

- Train your model using the fit method

Evaluate

- Evaluate your model with the test data

In the following diagram, you can find the relationship between the different key concepts. It can be said that the layers will transform data inputs by means of weights and the predicted outputs will be computed. The learning process will be able to compare the predicted outputs with the true outputs by means of the loss function, and such information will be used by the optimizer to adjust or update the weights interatively until meeting some convergence criteria.



In the rest of this notebook, you will be presented and provided with code for three examples about typical supervised classification problems. You will be asked to run the code blindly (i.e., probably not undestanding/being explained completely every single item), observing the outcomes and identifying the key concepts introduced previously in the workflow. In each example, you will be proposed a few general questions to discuss with your partners and foster questioning-discussion with your in-person lecturer/teacher.

Let's play with our first example using an ANN model in Keras, on prediction of patients with diabetes.

Example 1: Prediction of patients with diabetes (the Pima Indians Dataset Database)

This is a small introductory example in Keras, whose code has been adapted from here. We are going to build a very simple feedforward network to predict if a patient will have diabetes based on a series of clinical indicators (number of pregnancies, blood glucose level, blood pressure, etc), taken from the dataset 'Pima Indians Dataset Database'.

```
from numpy import loadtxt
from keras.models import Sequential
from keras.layers import Input,Dense

dataset = loadtxt('https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv', delimiter=','
print(type(dataset))
X_train = dataset[:700,0:8]  #X train: features of the first 700 samples
y_train = dataset[:700,8]    #y_train: outputs of the first 700 samples
X_test = dataset[700:,0:8]   #y_test: features of the remainder samples
y_test = dataset[700:,8]     #y_train: outputs of the remainder samples
print("Dataset:", dataset[0])
print("X_train[0]",X_train[0])
print("y_train[0]",y_train[0])
```

```
<class 'numpy.ndarray'>
Dataset: [  6.    148.     72.     35.      0.     33.6     0.627  50.      1.   ]
X_train[0] [  6.    148.     72.     35.      0.     33.6     0.627  50.   ]
y_train[0] 1.0
```

We can can create a neural network model in three simple steps:

- First we define the architecture (topology) of the network, just adding layers and connections between layers
- Then we can compile the architecture we create (compile() method)
- Once the architecture is compiled, we can train the model with data (fit() method)

So, let's start with the network achitecture. There are two ways to define it: the sequential model and the functional API.

The sequential model allows us to create simple linear architectures in which one layer suceeds another one, and the input of the current layer is the output of the previous one.

For each layer we define the number of neurons and activation function. For the Input layer we define the dimensions of the feature vectors that will be feeded to it (shape).

```
model = Sequential([
  Input(shape=(8,)),
  Dense(12, activation='relu'),
  Dense(8, activation='relu'),
  Dense(1, activation='sigmoid')])
model.summary()
```

Next, we compile the architecture, initializing its weights, and prepare it for training. For this we have to define an error function (loss), an optimization algorithm (optimizer), and a metric to measure the results, although we can establish more parameters of the model.

The output, loss function and metric of the model have to be tailored for the kind of problem we are trying to solve. In this case, the model will try predict if a patient will develop diabetes; so this is a binary classification problem. Therefore, we will use a single neuron in the output layer, with a sigmoid activation function (in the interval [0,1]), binary crossentropy as our loss function, and accuracy (percentage of predictions that are correct) as metric.

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

Now we can train the model. We pass to the fit() method the inputs and outputs separately, the number of epochs is going to take the training, and a batch size (the number of samples that are going to be feeded to the network in each training cycle).

```
model.fit(X_train, y_train, epochs=100, batch_size=16)
```

Once trained, we can test its performance on the test set, with the evaluate() method:

```
loss, accuracy = model.evaluate(X_test, y_test)
print('Loss: %.2f, Accuracy: %.2f' % (loss*100,accuracy*100))
```

We can also check what predictions the model outputs, with the predict() method:

```
predictions = model.predict(X_test)
print("Type",type(predictions),predictions.shape)
print("Prediction",predictions[:10])
predictions = (model.predict(X_test) > 0.5).astype("int32")
for i in range(10):
  print('%s  \tprediction: %d\t(real value  %d)' % (X_test[i].tolist(), predictions[i][0], y_test[i]))
```

Key questions and points for discussion with your partner.

In the sequential NN model just created and trained...

- How many neurons/units are there in the input, hidden and output layers?
- What is it expected as an input? (format and length)
- What is the activation function used in the output layer?
- To which extent, do you think that the model is good or bad? Do you think that the model is well adjusted? does it learn well from the data?
- How many samples were used for training? and how many for testing?

Example 2: Digit recognition with the MNIST dataset

In this second example (adapted from this tutorial) we will see how it creates a network capable of determining, from images of hand-drawn digits, which digit is represented. For this, we will use the MNIST dataset. Again we will use a Keras sequential model, with some differences with respect to the prior example:

- We will use CNNs (Convolutional Neural Networks, a good architecture for image recognition). You will learn more about CNNs in the theory lectures, and in the Course *Deep Learning* next spring. There you will see that they are capable of capturing increasingly complex spatial relationships (lines, edges, shapes, etc) from the input image, using less units than traditional multilayer perceptrons.
- We will define the layers of the model as a list.
- We will use dropout as a regularization technique to avoid overfitting.
- This is a multiclass classification problem, so the output layer will have a Softmax activation function.

In this example, we will ignore how to process inputs, since the MNIST dataset is preloaded in Keras.

```
import numpy as np
np.random.seed(123)

from keras.models import Sequential
from keras.layers import Input, Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras import utils
from keras.datasets import mnist

#load the dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

print("X_train:",type(X_train),X_train.shape)
print("y_train:",type(y_train),y_train.shape, "-->", y_train[100])
```

At this point we have loaded the MNIST data for training in the arrays `X_train` e `Y_train`, while testing data is in `X_test` e `Y_test`. The content of `X_train` y `X_test` is integers in the [0,255] range. We will normalize them to a [0,1] range as real numbers.

```
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
X_train = np.expand_dims(X_train, -1)
X_test = np.expand_dims(X_test, -1)

print("X_train:",type(X_train),X_train.shape)
print("y_train[:10]:",y_train[:10])
```

Since this is is a multiclass classification problem, the output layer will have a unit per class (10 neurons). The expectation is that one of the neurons will present an activation near 1, while the rest will be near zero. To enable this operation, we transform the values of `y_train` e `y_test` to one-hot vectors.

```
Y_train = utils.to_categorical(y_train, 10)
Y_test = utils.to_categorical(y_test, 10)

print("y_train:",type(Y_train),Y_train.shape, "-->", Y_train[100])
```

Next we define the model architecture, and compile it. Since this is a multiclass classification problem, the loss function will be categorical crossentropy).

```
model = Sequential(
  [
    Input(shape=(28,28,1)),
    Conv2D(32, kernel_size=(3, 3), activation='relu'),
    Conv2D(32, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2,2)),
    Dropout(0.25),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
  ]
)

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
model.summary()
```

We train the model with `fit()`. We use the parameter `validation_split` to set apart 20% of the input data to test the performance of the model while training.

```
model.fit(X_train, Y_train, batch_size=32, epochs=10, validation_split=0.2, verbose=1)
```

Once trained, we can evaluate it on the test set.

```
loss,accuracy = model.evaluate(X_test, Y_test, batch_size=16, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[0], loss*100))
print("%s: %.2f%%" % (model.metrics_names[1], accuracy*100))
```

Let's see the outputs.

```
predictions = (model.predict(X_test) > 0.5).astype("int32")
print("Shape of predictions:",np.shape(predictions))
for i in range(10):
  print("%d => %s = %d (expected %d)" % \
      (i, predictions[i].tolist(), np.nonzero(predictions[i])[0][0].item(), y_test[i]))
```

Some general questions/points for discussion with your partner.

- How large is the input to the network? Why?
- How large is the output layer? Why?
- Which preprocessing has been carried out?
- How many layers does the model have? (Notice there are non-dense layers. Feel free to check the Keras documentation for further explanation. These type of layers will be introduced in other subjects.)
- What is the validation set? Were you given a validation set? How was the validation set created? Do you think is it useful or necessary? How good do you think the trained model is? Why?
- What epochs parameter is set to? What does it mean? What does it happen if it is larger/smaller?
- Compare the code between examples 1 and 2. Briefly describe/discuss the similarities and differences between both.

Example 3: Classifying sentiment of movie reviews

This third example is based on the Internet Movie Database (IMDB) and the related existing dataset. The dataset is included in Keras already, and it will be downloaded first time you use it. It consists of 50K polarized reviews (25K for training, 25K for testing, each with 50-50% negative/positive review labels). The goal is to learn to classify whether a textual review is positive or negative based on the text. This means it is a binary classification problem.

The *train_data* and *test_data* are lists of reviews. However, machine learning and deep learning models can only process numerical input. Therefore, raw text cannot be used directly and must first be encoded into a numerical format. A simple approach is to assign a unique numerical identifier to each word. For instance, we can build a vocabulary (a list of known words) and use the index of each word in that list as its numerical ID. This way, every known word can be represented as an integer.

For this example, each review is a list of word indices (encoding a sequence of words). In the cell below, the vocabulary is built using the 10k most frequent words in the text. This means that only those 10k words will be considered known (Later on, we'll see how to handle unknown words, i.e., those not included in the vocabulary and therefore lacking a specific ID).

Finally, the variables *train_labels* and *test_labels* are lists of 0s and 1s, where 0 stands for negative and 1 stands for positive.

```
from keras.datasets import imdb

# only keep the top num_words most frequent. Discard rare words.
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)

# word_index maps words to integers
word_index = imdb.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
decoded_review = ' '.join([reverse_word_index.get(i - 3, '?') for i in train_data[0]])
print(decoded_review)
```

You may have noticed already from previous examples some usage of the Numpy library. It provides a bunch of useful operations to deal with vectors, arrays, tensors. It is specially relevant when preprocessing and preparing the inputs.

Remember that neural networks not only require numerical inputs instead of words, but also expect input vectors to have a fixed size. Therefore, the following code snippet performs some basic preprocessing steps to build a representation known as a **bag of words (BoW)**.

There are many ways to convert variable-length text sequences into fixed-size inputs for neural networks. Some are simple, like bag of words, while others are more complex and expressive. As the name suggests, bag of words is a basic strategy that represents each text as a multiset of known words, ignoring word order and grammar, but capturing word presence and frequency. In this case, we use the **binary version**, where each input vector has the same size as the vocabulary, and each value is either 1 or 0 depending on whether the corresponding word appears in the text.

The code below performs the following steps:

- Vectorization to keep length manageable and get a specific data representation.
- Kind of one-hot encoding with 0s and 1s whether terms/tokens are or not present in a given review.
- Also labels need to be vectorized.

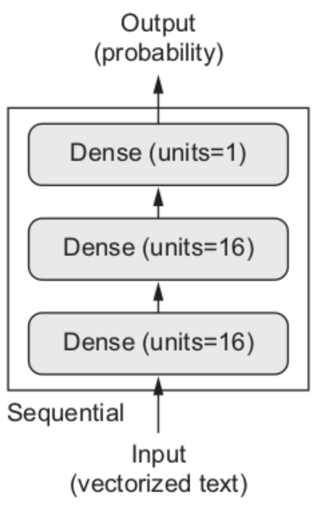
```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
  results = np.zeros((len(sequences), dimension))
  for i, sequence in enumerate(sequences):
    results[i, sequence] = 1.
  return results


x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)

y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

Now that we have the train and test sets as float vectors, we can proceed with the definition of the network architecture like in the image below.



Notice that the input layer is set to 10000 tokens in agreement with the vectorization done before. The Dense layers are indeed fully connected, and the Output layer will have a neuron with a sigmoid activation function for binary classification.

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Input(shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
model.summary()
```

You may compile the model in a similar way as done in other examples.

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

But notice that you may also set the learning rate parameter to gain more control on the learning process. You can find information about the available optimizers and their parameters here.

```
from keras import optimizers
model.compile(optimizer=optimizers.RMSprop(learning_rate=0.001),loss='binary_crossentropy', metrics=['accuracy'])
```

As a reference you can find in the following table a summary relating the problem types with typical activation functions used in the last layers and the loss function.

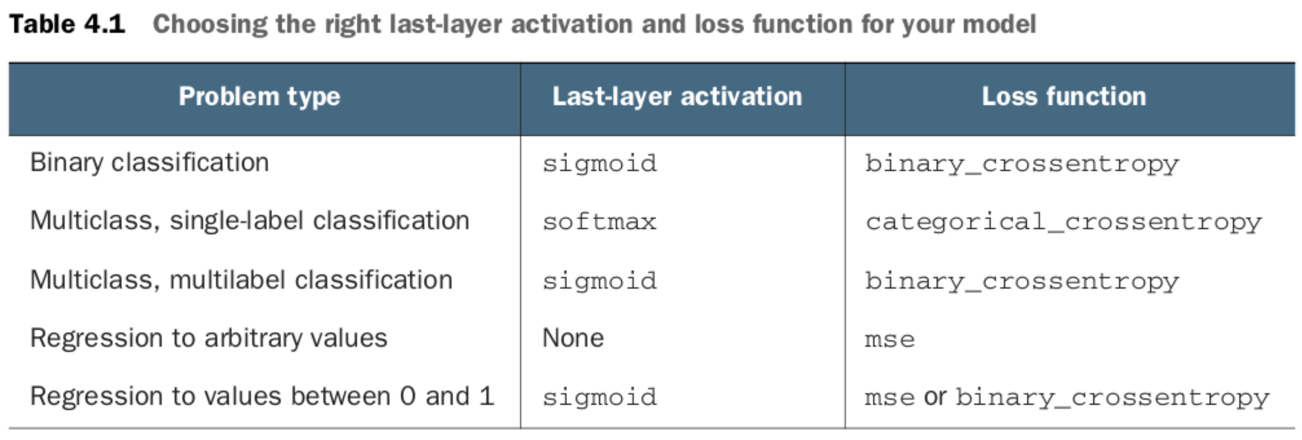


Table 4.1 **Choosing the right last-layer activation and loss function for your model**

Problem type	Last-layer activation	Loss function
Binary classification	sigmoid	binary_crossentropy
Multiclass, single-label classification	softmax	categorical_crossentropy
Multiclass, multilabel classification	sigmoid	binary_crossentropy
Regression to arbitrary values	None	mse
Regression to values between 0 and 1	sigmoid	mse or binary_crossentropy

We can set a subset for validation as shown next. In the .fit() method you can also specify the number of epochs and the samples used in minibatches. Notice that the model.fit() returns a History object, and you may try to use it *history_dict = history.history* for plotting training/validation loss/accuracy history.

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
```

```
partial_y_train = y_train[10000:]
history = model.fit(partial_x_train,partial_y_train,epochs=20, batch_size=512, validation_data=(x_val, y_val))
```

Now that you have inspected three examples, you are encouraged to browse the Keras documentation to find out more about methods, attributes, type of layers, loss and activation functions, etc. available in Keras.

Some general questions/points for discussion with your partner.

- How different is the way used to create the validation set in this example? Would it be equivalent to the way it was created in the previous example? Why? Why not?
- Would the presented model be enough to distinguish between these two reviews — “I love the movie, hate it wasn't longer.” (positive) and “I hate the movie, love it wasn't longer” (negative)? Why or why not?
- You get your model trained, do you think this is a good enough model?
- How do you use it to predict a label for a new data point? Try to write some code for this in a new code cell.
- How do you evaluate the model with the test data? Try to write some code for this in a new code cell.

You may try changing the network (type of layers, size, functions, etc.) and see how validation and test accuracy are affected.

```
# This is some spare code cell for you to try out something different.
```

Working with text input in practice

We have seen NNs do not work directly with text data, but with numbers that somehow represent the textual tokens. How text data is prepared, encoded, represented and fed to NNs is crucial and may be very important depending on the specific NLP task at hand.

Think about what text may actually be: Letters can be the single units of an alphabet (depends on the language, right?). Words can be sequences of letters, sentences can be sequences of words, paragraphs as a sequence of words, and so on. By now, we can assume in this lab that word segmentation or tokenization aim to separate words found in raw text.

What we need to handle text vectorization are Encodings and Word embeddings, for which Keras provides technical support. Both will be seen in detail in the theory lectures, but here you will get some informal introduction to get prepared for next assignment.

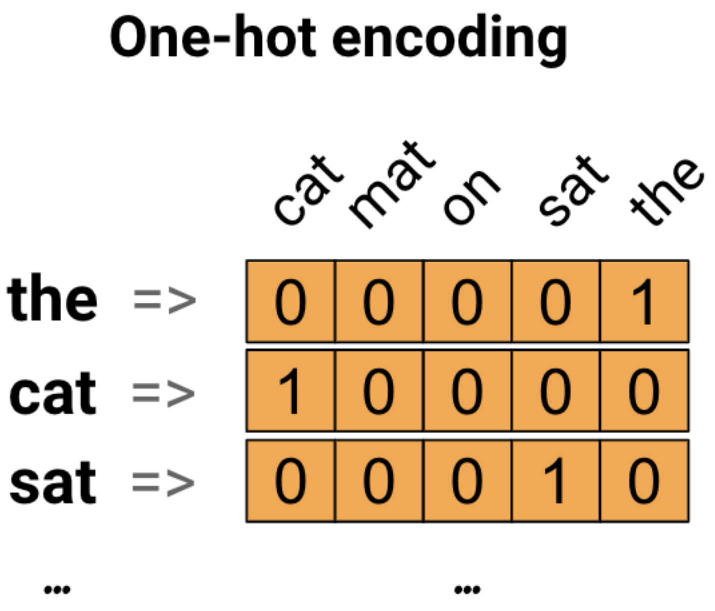
One-hot Encoding

There are many ways to proceed with vector representations for text data. One of the most common ones is using word-level one-hot encoding, which looks like obtaining sparse vectors, where most items are 0s and can be quite large (depending on the vocabulary).

Let's imagine we had only a sample sentence in raw text ("The cat sat on the mat"), brought here as a toy unrealistic example to raise discussion. The sentence would have 6 words but the vocabulary could just be 5 tokens long.

- Raw text => "The cat sat on the mat"
- Word-level Tokens => "cat", "mat", "on", "sat", "the"
- Length of vocabulary => 5
- One-hot vector for "cat" token => [1, 0, 0, 0, 0]
- A sentence can be encoded using binary BoW / multi-hot vectors vectors: [1,1,1,1,1] or a sequence of integer indexes: [5, 1, 4, 2, 3]

The image depicts how the one-hot encoding would be for each token.



One-hot encoding

	cat	mat	on	sat	the
the =>	0	0	0	0	1
cat =>	1	0	0	0	0
sat =>	0	0	0	1	0
...	...				

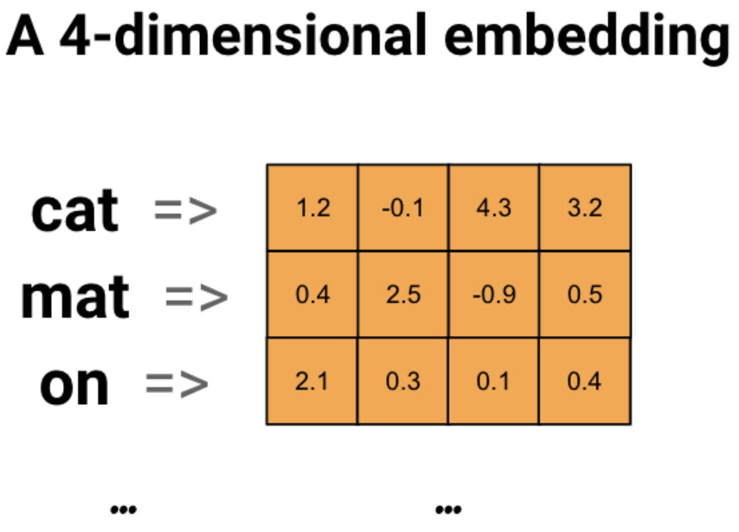
What would it happen if...

- switched the vector for "cat" and "the" in the *binary BoW* version? And in the "sequences" encoding?
- there were several documents?
- there were many documents leading to (a realistic situation with) thousands tokens?
- Can we handle unlimited number of tokens?
- Advantages and disadvantages of multi-hot binary encoding vs. Sequences?

Word Embeddings

An alternative to one-hot encodings are word embeddings. Word embeddings opt for dense representation and lower-dimension vectors. However, vector values need to be learned from data (so they are not intended for humans), but facilitates efficient vector arithmetic operations and thanks to their density, embeddings can encapsulate much richer information in a compact form.

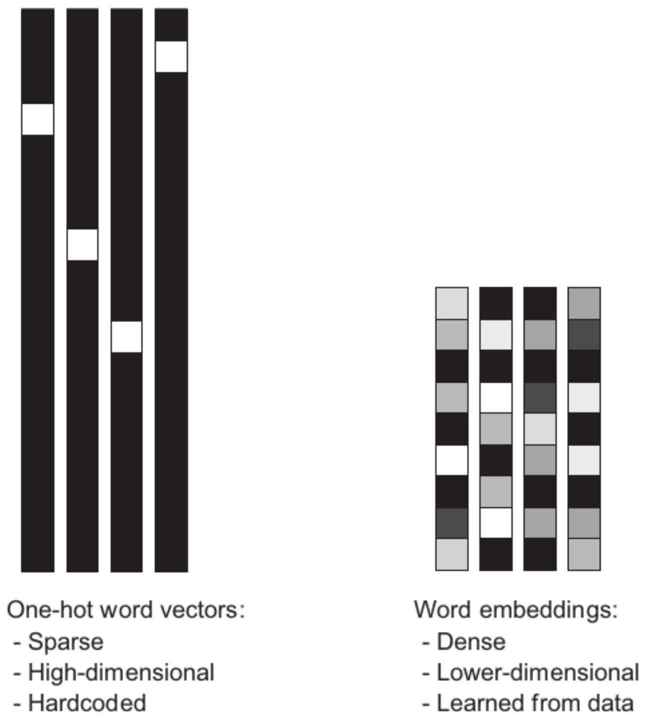
The image depicts how a 4-dimensional embedding may look like for some tokens.



A 4-dimensional embedding

cat =>	1.2	-0.1	4.3	3.2
mat =>	0.4	2.5	-0.9	0.5
on =>	2.1	0.3	0.1	0.4
...	...			

In this example, with just a few tokens, it may look like there would not be a gain, but think for a moment about having 100 documents leading to 10000 different tokens (just to mention some bigger numbers). You will notice that the size of the vector representing a token will remain the same dimension regardless.



You can find out more about these practical representations at https://www.tensorflow.org/text/guide/word_embeddings

How to load the Tokenizer class in Keras

Let's see an example of basic usage on using the Tokenizer facilitated by Keras. Other alternatives intended for text vectorization will be introduced and used in the notebook for the P0 final assignment.

```
from tensorflow.keras.preprocessing.text import Tokenizer #24-25: added tensorflow to the package path. It does not work othe
import numpy as np
```

How to transform texts to IDs. IDs are assigned based on frequency, i.e., the most common words will receive ID #1

```
# Creates a world-level tokenizer
tokenizer = Tokenizer()

texts = ["Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do.",
         "So she was considering in her own mind (as well as she could, for the hot day made her feel very sleepy and stupid)
         "There was nothing so VERY remarkable in that; nor did Alice think it so VERY much out of the way to hear the Rabbit

#It trains a tokenizer given a list of texts and builds the word index
tokenizer.fit_on_texts(texts)

# turns data text into lists of integer lists
# Basically: tokenizes the texts, and transform them into IDs
texts2ids = tokenizer.texts_to_sequences(texts)
print ("Texts as IDs:", texts2ids)

ids2texts = tokenizer.sequences_to_texts(texts2ids)
print ("IDs back to texts:", ids2texts)
```

```
Texts as IDs: [[8, 4, 14, 1, 15, 2, 16, 5, 17, 18, 6, 19, 20, 3, 21, 9, 5, 22, 10, 1, 23], [7, 11, 4, 24, 12, 6, 25, 26, 13, 27
IDs back to texts: ['alice was beginning to get very tired of sitting by her sister on the bank and of having nothing to do',
```

Notice that punctuation has been removed, the text has been lowercased, etc. This is because the default argument values of the class Tokenizer.

```
tf.keras.preprocessing.text.Tokenizer(
        num_words=None,
        filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n',
        lower=True, split=' ', char_level=False, oov_token=None,
        document_count=0, **kwargs
    )
```

They can be consulted at: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer

You have the freedom to set up the other parameters to your favorite choice, and use them in the assigments as needed to create your models.

More particularly, in the lab assignment we will modify two parameters: num_words and oov_token:

- **num_words**: the maximum number of words to keep, based on word frequency. Only the most common num_words-1 words will be kept.
- **oov_token**: Special token to handle unknown words, if given, it will be added to word_index (vocabulary) and used to replace **out-of-vocabulary** words during text_to_sequence calls.

At this point, you may have realized that assigning a single special token to represent all unknown words (OOV) might not always be the most effective solution. While this token allows us to handle words that are not in the vocabulary, it provides no specific information about them since the same token is used for every unknown word.

In Deep Learning, more advanced tokenization methods can capture richer information even for unknown words. One of the most widely used approaches is **subword tokenization**, which splits words into smaller components. In this setup, the vocabulary can include not only complete words but also smaller units such as syllables or individual characters.

This way, an unknown word can be represented as a sequence of known tokens, under the assumption that such reconstruction can provide more valuable information and help distinguish between different unknown words.

Some of the most popular methods, which will be covered in detail in the theoretical classes, include:

- Byte-Pair Encoding (BPE)
- WordPiece
- Unigram Language Model

To move on, we can consult what has been learned by the Tokenizer (i.e., the word index).

```
print ("tokenizer.word_counts:", tokenizer.word_counts)
print ()
print ("tokenizer.document_counts:", tokenizer.document_count)
print ()
print ("tokenizer.word_index:",tokenizer.word_index)
print ()
print ("tokenizer.word_docs", tokenizer.word_docs)
```

Notice tha the word index is giving you the index of each one-hot vector representing each single word considered in the vocabulary.

We could now to obtain one-hot encoding for the text sequence as follows.

```
# we are asking for a binary mode but other typical vectorizations are supported: "count", "tfidf", "freq"
one_hot_results = tokenizer.texts_to_matrix(texts, mode='binary')
print (one_hot_results)
print (one_hot_results[0])
```

Now it is time for you to look back the codes above and be sure you understand what they did.

How to save and load a tokenizer

```
import pickle

with open('tokenizer.pickle', 'wb') as handle:
          pickle.dump(tokenizer, handle, protocol=pickle.HIGHEST_PROTOCOL)

with open('tokenizer.pickle', 'rb') as handle:
          tokenizer = pickle.load(handle)
```

Example of a Tokenizer considering oov_tokens

We based our designs and prepare the pipeline of our NLP system considering the text corpora available for our task. However, even if we may handle a large vocabulary, it often happens that unknown/unseen tokens may still appear. In this case, we need to handle out-of-vocabulary (oov) tokens somehow. Fortunately, the class Tokenizer already comes oov tokens as illustrated next.

```
tokenizer = Tokenizer(oov_token="<unk>")
texts = ["Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do.",
        "So she was considering in her own mind (as well as she could, for the hot day made her feel very sleepy and stupid
        "There was nothing so VERY remarkable in that; nor did Alice think it so VERY much out of the way to hear the Rabbi
tokenizer.fit_on_texts(texts)

# Notice we are using here some words that haven't seen by the built tokenizer.
```

```
    oov_texts = ["Harry Potter and the Philoshopher stone"]

    texts2ids = tokenizer.texts_to_sequences(oov_texts)
    print ("Texts as IDs:", texts2ids)
    ids2texts = tokenizer.sequences_to_texts(texts2ids)
    print ("IDs back to texts:", ids2texts)
```

Example of a Tokenizer considering oov_tokens and limited num_words

Interestingly, the implementation of the Tokenizer can also limit the size of the vocabulary, using the most common words to create the word index.

```
    tokenizer = Tokenizer(oov_token="<unk>", num_words=15)
    texts = ["Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do.",
             "So she was considering in her own mind (as well as she could, for the hot day made her feel very sleepy and stupid)
             "There was nothing so VERY remarkable in that; nor did Alice think it so VERY much out of the way to hear the Rabbit
    tokenizer.fit_on_texts(texts)

    texts2ids = tokenizer.texts_to_sequences(texts) #
    print ("Texts as IDs:", texts2ids)
    ids2texts = tokenizer.sequences_to_texts(texts2ids)
    print ("IDs back to texts:", ids2texts)
```

Some general questions/points for discussion with your partner.

- What is *tokenizer.word_index*? what does it contain?
- What does *texts_to_sequences()* do? And what about *sequences_to_texts()*?
- How does the tokenization change when *num_words* is provided or not?
- How does the tokenization change when *oov_token* is provided or not?

TextVectorization

A more modern alternative to the class Tokenizer.

```
    texts = ["Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do.",
             "So she was considering in her own mind (as well as she could, for the hot day made her feel very sleepy and stupid)
             "There was nothing so VERY remarkable in that; nor did Alice think it so VERY much out of the way to hear the Rabbit

    text_vectorizer = layers.TextVectorization()
    text_vectorizer.adapt(texts)

    vectorized_text = text_vectorizer(texts)
    print(vectorized_text)
```

Notice that this is a layer. So, in order to use it, we have to add it as a layer in a model. In order to do that, we need to specify the imput of the model as one seting at a time.

```
    import tensorflow
    from keras.models import Sequential
    from keras.layers import Input,Dense

    model = Sequential()
    model.add(Input(shape=(1,), dtype=tensorflow.string))  #input must be one string at a time
    model.add(text_vectorizer)
    ...   #this code is incomplete. it won't work if you try to execute it! :)

    input_data = [["Harry Potter and the Philoshopher stone"], ["Once upon a time"]]
    model.predict(input_data)
```