Assignment 2 - Named Entity Recognition (NER)

Welcome to the second programming assignment of Course 3. In this assignment, you will learn to build more complicated models with Tensorflow. By completing this assignment, you will be able to:

- Design the architecture of a neural network, train it, and test it.
- Process features and represents them
- Understand word padding
- Implement LSTMs
- Test with your own sentence

Before getting started take some time to read the following tips:

TIPS FOR SUCCESSFUL GRADING OF YOUR ASSIGNMENT:

- All cells are frozen except for the ones where you need to submit your solutions.
- You can add new cells to experiment but these will be omitted by the grader, so don't rely on newly created cells to host your solution code, use the provided places for this.
- You can add the comment # grade-up-to-here in any graded cell to signal the grader that it must only evaluate up to that point. This is helpful if you want to check if you are on the right track even if you are not done with the whole assignment. Be sure to remember to delete the comment afterwards!
- To submit your notebook, save it and then click on the blue submit button at the beginning of the page.

Outline

- [Exercise 6]()

1 - Introduction

Let's begin by defining what a named entity recognition (NER) is. NER is a subtask of information extraction that locates and classifies named entities in a text. The named entities could be organizations, persons, locations, times, etc.

For example:

Named entity recognition

Many	French	citizens	are	going	to	Morocco	for	Christmas
O	*B_gpe*	*O*	*O*	*O*	*O*	*B_geo*	*O*	*B_tim*

Is labeled as follows:

- French: geopolitical entity
- Morocco: geographic entity
- Christmas: time indicator

Everything else that is labeled with an `O` is not considered to be a named entity. In this assignment, you will train a named entity recognition system that could be trained in a few seconds (on a GPU) and will get around 75% accuracy. You will then evaluate your model and see you get 97% accuracy! Finally, you will be able to test your named entity recognition system with your own sentence.

In [3]:
```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

import numpy as np
import pandas as pd
import tensorflow as tf

# set random seeds to make this notebook easier to replicate
tf.keras.utils.set_random_seed(33)
```

In [2]:
```
import w2_unittest
```

2 - Exploring the Data

You will be using a dataset from Kaggle, which it will be preprocessed for you. The original data consists of four columns: the sentence number, the word, the part of speech of the word (it won't be used in this assignment), and the tags. A few tags you might expect to see are:

- geo: geographical entity
- org: organization
- per: person

- gpe: geopolitical entity
- tim: time indicator
- art: artifact
- eve: event
- nat: natural phenomenon
- O: filler word

In [4]:
```
# display original kaggle data
data = pd.read_csv("data/ner_dataset.csv", encoding = "ISO-8859-1")
train_sents = open('data/small/train/sentences.txt', 'r').readline()
train_labels = open('data/small/train/labels.txt', 'r').readline()
print('SENTENCE:', train_sents)
print('SENTENCE LABEL:', train_labels)
print('ORIGINAL DATA:\n', data.head())
del(data, train_sents, train_labels)
```

```
SENTENCE: Thousands of demonstrators have marched through London to protest
the war in Iraq and demand the withdrawal of British troops from that countr
y .

SENTENCE LABEL: O O O O O O B-geo O O O O O B-geo O O O O O B-gpe O O O O O

ORIGINAL DATA:
     Sentence #           Word  POS Tag
0  Sentence: 1      Thousands  NNS   O
1          NaN             of   IN   O
2          NaN  demonstrators  NNS   O
3          NaN           have  VBP   O
4          NaN        marched  VBN   O
```

2.1 - Importing the Data

In this part, you will import the preprocessed data and explore it.

In [5]:
```
def load_data(file_path):
    with open(file_path,'r') as file:
        data = np.array([line.strip() for line in file.readlines()])
    return data
```

In [6]:
```
train_sentences = load_data('data/large/train/sentences.txt')
train_labels = load_data('data/large/train/labels.txt')

val_sentences = load_data('data/large/val/sentences.txt')
val_labels = load_data('data/large/val/labels.txt')

test_sentences = load_data('data/large/test/sentences.txt')
test_labels = load_data('data/large/test/labels.txt')
```

3 - Encoding

3.1 Encoding the sentences

In this section, you will use `tf.keras.layers.TextVectorization` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/TextVectorization) to transform the sentences into integers, so they can be fed into the model you will build later on.

You can use `help(tf.keras.layers.TextVectorization)` to further investigate the object and its parameters.

The parameter you will need to pass explicitly is `standardize` . This will tell how the parser splits the sentences. By default, `standardize = 'lower_and_strip_punctuation'` , this means the parser will remove all punctuation and make everything lowercase. Note that this may influence the NER task, since an upper case in the middle of a sentence may indicate an entity. Furthermore, the sentences in the dataset are already split into tokens, and all tokens, including punctuation, are separated by a whitespace. The punctuations are also labeled. That said, you will use `standardize = None` so everything will just be split into single tokens and then mapped to a positive integer.

Note that `tf.keras.layers.TextVectorization` will also pad the sentences. In this case, it will always pad using the largest sentence in the set you call it with. You will be calling it for the entire training/validation/test set, but padding won't impact at all the model's output, as you will see later on.

After instantiating the object, you will need to adapt it to the **sentences training set**, so it will map every token in the training set to an integer. Also, it will by default create two tokens: one for unknown tokens and another for the padding token. Tensorflow maps in the following way:

Exercise 1

Instructions: Use the object tf.keras.layers.TextVectorization and the appropriate parameters to build a function that inputs an array of sentences and outputs an adapted sentence vectorizer and its vocabulary list.

In [7]:
```
# GRADED FUNCTION: get_sentence_vectorizer
def get_sentence_vectorizer(sentences):
    tf.keras.utils.set_random_seed(33) ## Do not change this line.
    """
    Create a TextVectorization layer for sentence tokenization and adapt it tc

    Parameters:
    sentences (list of str): Sentences for vocabulary adaptation.

    Returns:
    sentence_vectorizer (tf.keras.layers.TextVectorization): TextVectorizatior
    vocab (list of str): Extracted vocabulary.
    """
    ### START CODE HERE ###

    # Define TextVectorization object with the appropriate standardize paramet
    sentence_vectorizer = tf.keras.layers.TextVectorization(standardize=None)

    # Adapt the sentence vectorization object to the given sentences
    sentence_vectorizer.adapt(sentences)
    # Get the vocabulary
    vocab = sentence_vectorizer.get_vocabulary()

    ### END CODE HERE ###

    return sentence_vectorizer, vocab
```

In [8]:
```
test_vectorizer, test_vocab = get_sentence_vectorizer(train_sentences[:1000])
print(f"Test vocab size: {len(test_vocab)}")

sentence = "I like learning new NLP models !"
sentence_vectorized = test_vectorizer(sentence)
print(f"Sentence: {sentence}\nSentence vectorized: {sentence_vectorized}")
```

```
Test vocab size: 4650
Sentence: I like learning new NLP models !
Sentence vectorized: [ 296  314    1   59    1    1 4649]
```

Expected output:

```
Test vocab size: 4650
Sentence: I like learning new NLP models !
Sentence vectorized: [ 296  314    1   59    1    1 4649]
```

In [9]:
```
w2_unittest.test_get_sentence_vectorizer(get_sentence_vectorizer)
```

```
All tests passed
```

In [10]:
```
sentence_vectorizer, vocab = get_sentence_vectorizer(train_sentences)
```

```
WARNING:tensorflow:5 out of the last 36 calls to <function PreprocessingLaye
r.make_adapt_function.<locals>.adapt_step at 0x78e3a1ccfaf0> triggered tf.fu
nction retracing. Tracing is expensive and the excessive number of tracings
could be due to (1) creating @tf.function repeatedly in a loop, (2) passing
tensors with different shapes, (3) passing Python objects instead of tensor
s. For (1), please define your @tf.function outside of the loop. For (2), @t
f.function has reduce_retracing=True option that can avoid unnecessary retra
cing. For (3), please refer to https://www.tensorflow.org/guide/function#con
trolling_retracing (https://www.tensorflow.org/guide/function#controlling_re
tracing) and https://www.tensorflow.org/api_docs/python/tf/function (http
s://www.tensorflow.org/api_docs/python/tf/function) for  more details.
```

3.2 Encoding the labels

In this section you will encode the labels. The process is a bit simpler than encoding the sentences, because there are only a few tags, compared with words in the vocabulary. Note, also, that there will be one extra tag to represent the padded token that some sentences may have included. Padding will not interfere at all in this task, as you will see further on. Run the next cell to print one example of a tag related to one sentence.

Because there is no meaning in having an UNK token for labels and the padding token will be another number different from 0 (you will see why soon), TextVectorization is not a good choice.

You will need also to pad the labels, because the number of labels must match the number of words.

In [11]:
```
print(f"Sentence: {train_sentences[0]}")
print(f"Labels: {train_labels[0]}")
```

```
Sentence: Thousands of demonstrators have marched through London to protest
the war in Iraq and demand the withdrawal of British troops from that countr
y .
Labels: O O O O O O B-geo O O O O O B-geo O O O O O B-gpe O O O O O
```

You will build the next function to extract all the different tags in a given set of labels.

In [12]:
```
def get_tags(labels):
    tag_set = set() # Define an empty set
    for el in labels:
        for tag in el.split(" "):
            tag_set.add(tag)
    tag_list = list(tag_set)
    tag_list.sort()
    return tag_list
```

In [13]:
```
tags = get_tags(train_labels)
print(tags)
```

```
['B-art', 'B-eve', 'B-geo', 'B-gpe', 'B-nat', 'B-org', 'B-per', 'B-tim', 'I-
art', 'I-eve', 'I-geo', 'I-gpe', 'I-nat', 'I-org', 'I-per', 'I-tim', 'O']
```

Now you will need to generate a **tag map**, i.e., a mapping between the tags and **positive** integers.

In [14]:
```
def make_tag_map(tags):
    tag_map = {}
    for i,tag in enumerate(tags):
        tag_map[tag] = i
    return tag_map
```

The `tag_map` is a dictionary that maps the tags that you could have to numbers. Run the cell below to see the possible classes you will be predicting. The prepositions in the tags mean:

- I: Token is inside an entity.
- B: Token begins an entity.

If you had the sentence

"Sharon flew to Miami on Friday"

The tags would look like:

```
Sharon B-per
flew   O
to     O
Miami  B-geo
on     O
Friday B-tim
```

where you would have three tokens beginning with B-, since there are no multi-token entities in the sequence. But if you added Sharon's last name to the sentence:

"Sharon Floyd flew to Miami on Friday"

```
Sharon B-per
Floyd  I-per
flew   O
to     O
Miami  B-geo
on     O
Friday B-tim
```

Your tags would change to show first "Sharon" as B-per, and "Floyd" as I-per, where I- indicates an inner token in a multi-token sequence.

In [15]:
```
tag_map = make_tag_map(tags)
print(tag_map)
```

```
{'B-art': 0, 'B-eve': 1, 'B-geo': 2, 'B-gpe': 3, 'B-nat': 4, 'B-org': 5, 'B-
per': 6, 'B-tim': 7, 'I-art': 8, 'I-eve': 9, 'I-geo': 10, 'I-gpe': 11, 'I-na
t': 12, 'I-org': 13, 'I-per': 14, 'I-tim': 15, 'O': 16}
```

3.3 Padding the labels

In this section, you will pad the labels. TextVectorization already padded the sentences, so you must ensure that the labels are properly padded as well. This is not a hard task for two main reasons:

1. Tensorflow has built-in functions for padding
2. Padding will be performed uniformly per dataset (train, validation and test) using the maximum sentence length in each dataset and the size of each sentence is exactly the same as the size of their respective labels.

You will pad the vectorized labels with the value -1. You will not use 0 to simplify loss masking and evaluation in further steps. This is because to properly classify one token, a log softmax transformation will be performed and the index with greater value will be the index label. Since index starts at 0, it is better to keep the label 0 as a valid index, even though it is possible to also use 0 as a mask value for labels, but it would require some tweaks in the model architecture or in the loss computation.

Tensorflow provides the function `tf.keras.utils.pad_sequences` (https://www.tensorflow.org/api_docs/python/tf/keras/utils/pad_sequences). The arguments you will need are:

- `sequences` : An array with the labels.
- `padding` : The position where padding will take place, the standard is `pre` , meaning the

3.4 Building the label vectorizer

Now you're ready to code the label vectorizer.

Exercise 2

Instructions: You will build the label vectorizer, a function that inputs a list of labels and a tag mapping and outputs their respective label ids via a tag map lookup. The tensorflow function `pad_sequences` can be called by `tf.keras.utils.pad_sequences` . You may also type `help(tf.keras.utils.pad_sequences)` to see its documentation.

In [16]:
```
# GRADED FUNCTION: label_vectorizer
def label_vectorizer(labels, tag_map):
    """
    Convert list of label strings to padded label IDs using a tag mapping.

    Parameters:
    labels (list of str): List of label strings.
    tag_map (dict): Dictionary mapping tags to IDs.
    Returns:
    label_ids (numpy.ndarray): Padded array of label IDs.
    """
    label_ids = [] # It can't be a numpy array yet, since each sentence has a

    ### START CODE HERE ###

    # Each element in labels is a string of tags so for each of them:
    for element in labels:
        # Split it into single tokens. You may use .split function for strings
        tokens = element.split(' ')

        # Use the dictionaty tag_map passed as an argument to the label_vector
        # to make the correspondence between tags and numbers.
        element_ids = []

        for token in tokens:
            element_ids.append(tag_map[token])

        # Append the found ids to corresponding to the current element to labe
        label_ids.append(element_ids)

    # Pad the elements
    label_ids = tf.keras.utils.pad_sequences(label_ids,padding='post',value=-1


    ### END CODE HERE ###

    return label_ids
```

In [17]:
```
print(f"Sentence: {train_sentences[5]}")
print(f"Labels: {train_labels[5]}")
print(f"Vectorized labels: {label_vectorizer([train_labels[5]], tag_map)}")
```

```
Sentence: The party is divided over Britain 's participation in the Iraq con
flict and the continued deployment of 8,500 British troops in that country .
Labels: O O O O O B-gpe O O O O B-geo O O O O O O O B-gpe O O O O O
Vectorized labels: [[16 16 16 16 16  3 16 16 16 16  2 16 16 16 16 16 16 16
3 16 16 16 16 16]]
```

Expected output:

```
Sentence: The party is divided over Britain 's participation in the I
raq conflict and the continued deployment of 8,500 British troops in
that country .
Labels: O O O O O B-gpe O O O O B-geo O O O O O O O B-gpe O O O O O
Vectorized labels: [[16 16 16 16 16  3 16 16 16 16  2 16 16 16 16 16
16 16  3 16 16 16 16 16]]
```

In [18]:
```
w2_unittest.test_label_vectorizer(label_vectorizer)
```

```
All tests passed
```

4 Building the Dataset

In this section, you will build the dataset for training, validation and testing. You will be using `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) class, which provides an optimized way to handle data to feed into a tensorflow model. It may be not as straightforward as a pandas dataset, but it avoids keeping all the data in memory, thus it makes the training faster.

You will be using the `tf.data.Dataset.from_tensor_slices` function that converts any iterable into a Tensorflow dataset. You can pass a tuple of `(sentences,labels)` and Tensorflow will understand that each sentence is mapped to its respective label, therefore it is expected that if a tuple of arrays is passed, both arrays have the same length.

In [19]:
```
def generate_dataset(sentences, labels, sentence_vectorizer, tag_map):
    sentences_ids = sentence_vectorizer(sentences)
    labels_ids = label_vectorizer(labels, tag_map = tag_map)
    dataset = tf.data.Dataset.from_tensor_slices((sentences_ids, labels_ids))
    return dataset
```

The next cell will use the function defined above to generate a Tensorflow Dataset for each of the train, validation and test datasets.

In [20]:
```
train_dataset = generate_dataset(train_sentences,train_labels, sentence_vector
val_dataset = generate_dataset(val_sentences,val_labels,  sentence_vectorizer,
test_dataset = generate_dataset(test_sentences, test_labels,  sentence_vectori
```

In [21]:
```
# Exploring information about the training data
print(f'The number of outputs is {len(tags)}')
# The number of vocabulary tokens (including <PAD>)
g_vocab_size = len(vocab)
print(f"Num of vocabulary words in the training set: {g_vocab_size}")
print('The training size is', len(train_dataset))
print('The validation size is', len(val_dataset))
print('An example of the first sentence is\n\t', next(iter(train_dataset))[0].
print('An example of its corresponding label is\n\t', next(iter(train_dataset)
```

```
The number of outputs is 17
Num of vocabulary words in the training set: 29847
The training size is 33570
The validation size is 7194
An example of the first sentence is
	 [1046    6 1121   18 1832  232  543    7  528    2  158    5   60
9
  648    2  922    6  192   87   22   16   54    3    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0]
An example of its corresponding label is
	 [16 16 16 16 16 16  2 16 16 16 16 16  2 16 16 16 16 16  3 16 16 16
16 16
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 -1 -1 -1 -1 -1 -1 -1 -1]
```

3.5 - Considerations about RNNs and LSTMs inputs

Tensorflow implementation of RNNs (in particular LSTMs) allow you to pass a variable size of input sentences, however this cannot be done **in the same batch**. You must assure that, for each batch, the shapes for our input tensors are the same.

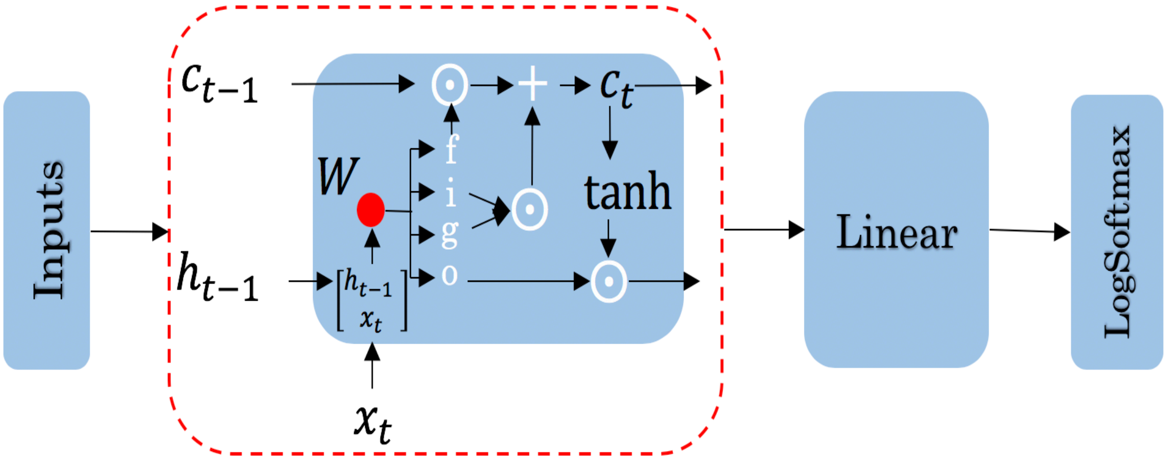
A second point here is that, for this purpose, the size of the padding should not influence the final result. Therefore, it does not matter if you perform the padding for each batch or in the entire dataset.

4 - Building the Model

4.1 Model structure

You will now implement the model that will be able to determine the tags of sentences like the following:

The model architecture will be as follows:



Concretely, your inputs will be sentences represented as tensors that are fed to a model with:

- An Embedding layer,
- A LSTM layer
- A Dense layer
- A log softmax layer.

You may choose between outputting only the very last LSTM output for each sentence, but you may also request the LSTM to output every value for a sentence - this is what you want. You will need every output, because the idea is to label every token in the sentence and not to predict the next token or even make an overall classification task for that sentence.

This implies that when you input a single sentence, such as `[452, 3400, 123, 0, 0, 0]` , the expected output should be an array for each word ID, with a length equal to the number of tags. This output is obtained by applying the LogSoftfmax function for each of the `len(tags)` values. So, in the case of the example array with a shape of `(6,)` , the output should be an array with a shape of `(6, len(tags))` .

In your case, you've seen that each sentence in the training set is 104 values long, so in a batch of, say, 64 tensors, the model shoud input a tensor of shape `(64,104)` and output another tensor with shape `(64,104,17)` .

Good news! We won't make you implement the LSTM cell drawn above. You will be in charge of the overall architecture of the model.

Exercise 3

Instructions: Implement the NER model, with the architecture discussed in the lectures. All the necessary layers are objects from the `tensorflow.keras.layers` library, but they are already loaded in memory, so you do not have to worry about function calls.

Please utilize help function e.g. `help(tf.keras.layers.Dense)` for more information on a layer

- [tf.keras.Sequential (https://www.tensorflow.org/api_docs/python/tf/keras/Sequential)](https://www.tensorflow.org/api_docs/python/tf/keras/Sequential): Combinator that applies layers serially (by function composition) - **this is not properly a layer (it is under `tensorflow.keras` only and not under `tensorflow.keras.layers`)**. It is in fact a Tensorflow [model (https://www.tensorflow.org/api_docs/python/tf/keras/Model)](https://www.tensorflow.org/api_docs/python/tf/keras/Model) object.
  - You can add the layers to a `Sequential` layer by calling the method `.add(layer)`.
  - You may skip the input shape and pass it in the first layer you instantiate, if necessary (RNNs usually don't need to fix an input length).
- [tf.keras.layers.Embedding (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding)](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding): Initializes the embedding layer. An embedding layer in tensorflow will input only **positive integers**.
  - `Embedding(input_dim, output_dim, mask_zero = False)`.
  - `input_dim` is the expected range of integers for each tensor in the batch. Note that the input_dim is not related to array size, but to the possible range of integers expected in the input. Usually this is the vocabulary size, but it may differ by 1, depending on further parameters. See below.
  - `output_dim` is the number of elements in the word embedding (some choices for a word embedding size range from 150 to 300, for example). Each word processed will be assigned an array of size `output_dim`. So if one array of shape (3,) is passed (example of such an array `[100,203,204]`), then the Embedding layer should have output shape (3,output_dim).
  - `mask_zero` is a boolean telling whether 0 is a mask value or not. If `mask_zero = True`, then some considerations must be done: 1. The value 0 should be reserved as the mask value, as it will be ignored in training. 2. You need to add 1 in `input_dim`, since now Tensorflow will consider that one extra 0 value may show up in each sentence.
- [tf.keras.layers.LSTM (https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM)](https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM): An LSTM layer.
  - `LSTM(units, return_sequences)` Builds an LSTM layer with hidden state and cell sizes equal to `units`. The arguments you will need: 1. `units`: It is the number of `LSTM` cells you will create to pass every input to. In this case, set the `units` as the Embedding `output_dim`. This is just a choice, in fact there is no static rule preventing one from choosing any amount of LSTM units. 2. `return_sequences`: A boolean, telling whether you want to return every output value from the LSTM cells. If `return_sequences = False`, then the LSTM output shape will be `(batch_size, units)`. Otherwise, it is `(batch_size, sentence_length, units)`, since there will be an output for each word in the sentence.
- [tf.keras.layers.Dense (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense)](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense): A dense layer.

Instructions: You will build a function that inputs the number of tags, the vocabulary size and an optional parameter to control the embedding dimension and outputs a tensorflow model as discussed in the lectures.

In [22]:
```
# GRADED FUNCTION: NER
def NER(len_tags, vocab_size, embedding_dim=50):
    """
    Create a Named Entity Recognition (NER) model.

    Parameters:
    len_tags (int): Number of NER tags (output classes).
    vocab_size (int): Vocabulary size (excluding padding index).

    Returns:
    model (Sequential): NER model.
    """

    model = tf.keras.Sequential(name='sequential')

    # Correct input_dim to vocab_size + 1 to account for mask_zero
    model.add(tf.keras.layers.Embedding(input_dim=vocab_size + 1,
                                        output_dim=embedding_dim,
                                        mask_zero=True))

    model.add(tf.keras.layers.LSTM(units=embedding_dim, return_sequences=True)

    model.add(tf.keras.layers.Dense(units=len_tags, activation=tf.nn.log_softm

    return model
```

In [23]:
```
w2_unittest.test_NER(NER)
```

```
All tests passed
```

4.2 Masked loss and metrics

Before training the model, you need to create your own function to compute the accuracy. Tensorflow has built-in accuracy metrics but you cannot pass values to be ignored. This will impact the calculations, since you must remove the padded values. Before diving into the exercise, let's just make some points clear.

Usually, the metric that inputs true labels and predicted labels and outputs how many times the predicted and true labels match is called `accuracy` . In some cases, however, there is one more step before getting the predicted labels. This may happen if, instead of passing the predicted labels, a vector of probabilities is passed. In such case, there is a need to perform an `argmax` for each prediction to find the appropriate predicted label. Such situations happen very often, therefore Tensorflow has a set of functions, with prefix `Sparse` , that performs this operation in the backend. Unfortunately, it does not provide values to ignore in the accuracy case. This is what you will work on now.

Note that the model's prediction has 3 axes:

- the number of examples (batch size)

- the number of words in each example (padded to be as long as the longest sentence in the batch)

Another important function is the loss function. In this case, you will use the Cross Entropy loss, but you need a multiclass implementation of it, also you may look for its `Sparse` version. Tensorflow has a SparseCategoricalCrossentropy loss function, which it is already imported by the name SparseCategoricalCrossEntropy.

[SparseCategoricalCrossentropy (https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy)](https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy): The Sparse Categorical Crossentropy Loss Function.

The arguments you will need:

1. `from_logits` : This indicates if the values are raw values or normalized values (probabilities). Since the last layer of the model finishes with a LogSoftMax call, the results are **not** normalized - they do not lie between 0 and 1.
2. `ignore_class` : This indicates which class should be ignored when computing the crossentropy. Remember that the class related to padding value is set to be 0.

Note: You do not need to worry if the outputs are normalized or not in the accuracy case. Can you guess why? :)

Exercise 4

Instructions: You will use a `tf.keras.losses.SparseCategoricalCrossentropy` object to create a loss function that ignores the padded value related to the label. **Remember that for padding you are using the value -1 and not 0, as opposed to the text padding!**

In [24]:
```
import tensorflow as tf

def masked_loss(y_true, y_pred):
    """
    Calculate the masked sparse categorical cross-entropy loss.

    Parameters:
    y_true (tensor): True labels.
    y_pred (tensor): Predicted logits.

    Returns:
    loss (tensor): Calculated loss.
    """

    ### START CODE HERE ###

    # Initialize the loss function with from_logits=True since y_pred contains
    loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True,

    # Ensure both y_true and y_pred are of type tf.float32 (if needed)
    y_true = tf.cast(y_true, tf.float32)
    y_pred = tf.cast(y_pred, tf.float32)

    # Compute the loss for each item in the batch
    loss = loss_fn(y_true, y_pred)

    # Apply the mask to ignore invalid labels (padding labels with -1)
    mask = tf.cast(tf.not_equal(y_true, -1), dtype=tf.float32)

    # Apply the mask to the loss
    loss = loss * mask

    # Compute the average loss (sum of valid losses divided by the number of v
    loss = tf.reduce_sum(loss) / tf.reduce_sum(mask)

    ### END CODE HERE ###

    return loss
```

In [25]:
```
true_labels = [0,1,2,0]
predicted_logits = [[-2.3,-0.51,-1.20] , [-1.61,-0.36,-2.30], [-2.30, -0.69,-0
print(masked_loss(true_labels, predicted_logits))
```

```
tf.Tensor(1.1242604, shape=(), dtype=float32)
```

Expected output:

```
tf.Tensor(1.1242604, shape=(), dtype=float32)
```

In [26]:
```
w2_unittest.test_masked_loss(masked_loss)
```

```
All tests passed
```

Exercise 5

Instructions: You will maked a masked version of the accuracy function. You will need to perform an argmax to get the predicted label for each element in the batch. Remember to provide the appropriate axis in the argmax function. Furthermore, remember to use only tensorflow operations. Even though numpy has every function you will need, to pass it as a loss function and/or metric function, you must use tensorflow operations, due to internal optimizations that Tensorflow performs for reliable fitting. The following tensorflow functions are already loaded in memory, so you can directly call them.

1. tf.equal, equivalent to np.equal
2. tf.cast, equivalent to np.astype
3. tf.reduce_sum, equiavalent to np.sum
4. tf.math.argmax, equivalent to np.argmax

In [27]:

```
# GRADED FUNCTION: masked_accuracy
def masked_accuracy(y_true, y_pred):
    """
    Calculate masked accuracy for predicted labels.

    Parameters:
    y_true (tensor): True labels.
    y_pred (tensor): Predicted logits.

    Returns:
    accuracy (tensor): Masked accuracy.

    """

    ### START CODE HERE ###

    # Calculate the loss for each item in the batch.
    # You must always cast the tensors to the same type in order to use them i
    y_true = tf.cast(y_true, tf.float32)
    # Create the mask, i.e., the values that will be ignored
    mask = tf.not_equal(y_true, -1)
    mask = tf.cast(mask, tf.float32)
    # Perform argmax to get the predicted values
    y_pred_class = tf.math.argmax(y_pred,axis=-1)
    y_pred_class = tf.cast(y_pred_class, tf.float32)
    # Compare the true values with the predicted ones
    matches_true_pred  = tf.equal(y_true, y_pred_class)
    matches_true_pred = tf.cast(matches_true_pred , tf.float32)
    # Multiply the acc tensor with the masks
    matches_true_pred *= mask
    # Compute masked accuracy (quotient between the total matches and the tota
    masked_acc = tf.reduce_sum(matches_true_pred) /tf.reduce_sum(mask)


    ### END CODE HERE ###

    return masked_acc
```

In [28]:
```
true_labels = [0,1,2,0]
predicted_logits = [[0.1,0.6,0.3] , [0.2,0.7,0.1], [0.1, 0.5,0.4], [0.4,0.4,0.
print(masked_accuracy(true_labels, predicted_logits))
```

```
tf.Tensor(0.5, shape=(), dtype=float32)
```

Expected output:

```
tf.Tensor(0.5, shape=(), dtype=float32)
```

In [29]:
```
w2_unittest.test_masked_accuracy(masked_accuracy)
```

```
 All tests passed
```

Now you will create the model and get a summary of its parameters and layers.

In [30]:
```
model = NER(len(tag_map), len(vocab))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 embedding_3 (Embedding)     (None, None, 50)          1492400   
                                                                 
 lstm_3 (LSTM)               (None, None, 50)          20200     
                                                                 
 dense_3 (Dense)             (None, None, 17)          867       
                                                                 
=================================================================
Total params: 1513467 (5.77 MB)
Trainable params: 1513467 (5.77 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Expected output:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 embedding (Embedding)       (None, None, 50)          1492400   
                                                                 
 lstm (LSTM)                 (None, None, 50)          20200     
                                                                 
 dense (Dense)               (None, None, 17)          867       
                                                                 
=================================================================
Total params: 1513467 (5.77 MB)
Trainable params: 1513467 (5.77 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

4.3 A note on padding

You will check now how padding does not affect the model's output. Of course the output dimension will change. If ten zeros are added at the end of the tensor, then the resulting output dimension will have 10 more elements (more specifically, 10 more arrays of length 17 each). However, those are removed from any calculation further on, so it won't impact at all the model's performance and training. You will be using the function tf.expand_dims.

In [31]:
```
x = tf.expand_dims(np.array([545, 467, 896]), axis = 0) # Expanding dims is ne
                                                        # since it expects bat

x_padded = tf.expand_dims(np.array([545, 467, 896, 0, 0, 0]), axis = 0)
```

Can you guess the final output prediction shape for each array defined above?

In [32]:
```
pred_x = model(x)
pred_x_padded = model(x_padded)
print(f'x shape: {pred_x.shape}\nx_padded shape: {pred_x_padded.shape}')
```

```
x shape: (1, 3, 17)
x_padded shape: (1, 6, 17)
```

If the last three elements of `pred_x_padded` are removed, both `pred_x` and `pred_x_padded[:3]` must have the same elements.

In [33]:
```
np.allclose(pred_x, pred_x[:3])
```

Out[33]:
```
True
```

Great! Now one last check: let's see that both `pred_x` and `pred_x_padded` return the same loss and accuracy values. For that, you will need a `y_true` and `y_true_padded` arrays.

In [34]:
```
y_true = tf.expand_dims([16, 6, 12], axis = 0)
y_true_padded = tf.expand_dims([16,6,12,-1,-1,-1], axis = 0) # Remember you mc
print(f"masked_loss is the same: {np.allclose(masked_loss(y_true,pred_x), mask
print(f"masked_accuracy is the same: {np.allclose(masked_accuracy(y_true,pred_
```

```
masked_loss is the same: True
masked_accuracy is the same: True
```

After this quick sanity check, you will now compile the model.

You will compile the model as follows:

1. Use the Adam optimizer to compute the stochastic gradient descent, with learning rate 0.01
2. Use the loss function `masked_loss` as loss function,
3. As evaluation metrics, you will use both masked_loss and masked_accuracy

In [35]:
```
model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
              loss = masked_loss,
               metrics = [masked_accuracy])
```

4.4 - Training the Model

You will now train the model.

Instructions:

- You will train it with `shuffle = True` , over 2 epochs and passing the validation dataset as `validation_data` .
- You will run into an error if you just pass the datasets as they are right now, because they are not prepared in batches. You must use the method `.batch` that returns a dataset already divided in batches

NOTE: The fitting takes about 1 minute to run. Only the first epoch is slow, the following ones are much faster.

In [36]:
```
tf.keras.utils.set_random_seed(33) ## Setting again a random seed to ensure re

BATCH_SIZE = 64

model.fit(train_dataset.batch(BATCH_SIZE),
          validation_data = val_dataset.batch(BATCH_SIZE),
          shuffle=True,
          epochs = 2)
```

```
Epoch 1/2
525/525 [==============================] - 40s 69ms/step - loss: 0.2602 - ma
sked_accuracy: 0.9323 - val_loss: 0.1397 - val_masked_accuracy: 0.9582
Epoch 2/2
525/525 [==============================] - 4s 8ms/step - loss: 0.1084 - mask
ed_accuracy: 0.9660 - val_loss: 0.1364 - val_masked_accuracy: 0.9585
```

Out[36]:
```
<keras.src.callbacks.History at 0x78e39821a3d0>
```

5 - Compute Accuracy

You will now evaluate on the test set. Previously, you have seen the accuracy on the training set and the validation (noted as eval) set. You will now evaluate on your test set. You already have a function to compute the accuracy.

In [42]:
```
# Convert the sentences into ids
test_sentences_id = sentence_vectorizer(test_sentences)
# Convert the labels into token ids
test_labels_id = label_vectorizer(test_labels,tag_map)
# Rename to prettify next function call
y_true = test_labels_id
y_pred = model.predict(test_sentences_id)
```

```
225/225 [==============================] - 1s 2ms/step
```

The next cell computes the accuracy for the test set.

In [43]:
```
print(f"The model's accuracy in test set is: {masked_accuracy(y_true,y_pred).r
```

```
The model's accuracy in test set is: 0.9576
```

6 - Testing with your Own Sentence

In this section you will make a predictor function to predict the NER labels for any sentence.

Exercise 6

Instructions: You will make a function `predict` that inputs one arbitrary sentence, a trained NER model, the sentence_vectorizer and the tag mapping and return a list of predicted NER labels. Remember that the sentences in pre-processing were already separated by token, so you do not need to worry about separating tokens such as commas or dots. You will just pass one sentence in the desired format, e.g., sentence = "I like apples , oranges and grapes ."

To get a single prediction from a tensorflow model, you will need to make some changes in the input array, since tensorflow expects a batch of sentences. You can use the function `tf.expand_dims` to do this.

In [44]:
```
# GRADED FUNCTION: predict
def predict(sentence, model, sentence_vectorizer, tag_map):
    """
    Predict NER labels for a given sentence using a trained model.

    Parameters:
    sentence (str): Input sentence.
    model (tf.keras.Model): Trained NER model.
    sentence_vectorizer (tf.keras.layers.TextVectorization): Sentence vectoriz
    tag_map (dict): Dictionary mapping tag IDs to labels.

    Returns:
    predictions (list): Predicted NER labels for the sentence.

    """

    ### START CODE HERE ###

    # Convert the sentence into ids
    sentence_vectorized = sentence_vectorizer(sentence)
    # Expand its dimension to make it appropriate to pass to the model
    sentence_vectorized = tf.expand_dims(sentence_vectorized, axis=0)
    # Get the model output
    output = model(sentence_vectorized)
    # Get the predicted labels for each token, using argmax function and speci
    outputs = np.argmax(output, axis = -1)
    # Next line is just to adjust outputs dimension. Since this function expec
    # so to avoid heavy notation below, let's transform it into [1,2,3]
    outputs = outputs[0]
    # Get a list of all keys, remember that the tag_map was built in a way tha
    labels = list(tag_map.keys())
    pred = []
    # Iterating over every predicted token in outputs list
    for tag_idx in outputs:
        pred_label = labels[tag_idx]
        pred.append(pred_label)

    ### END CODE HERE ###

    return pred
```

In [45]:
```
w2_unittest.test_predict(predict, model, sentence_vectorizer, tag_map)
```

```
All tests passed
```

In [46]:
```python
# Try the output for the introduction example
#sentence = "Many French citizens are goin to visit Morocco for summer"
#sentence = "Sharon Floyd flew to Miami last Friday"

# New york times news:
sentence = "Peter Parker , the White House director of trade and manufacturing
predictions = predict(sentence, model, sentence_vectorizer, tag_map)
for x,y in zip(sentence.split(' '), predictions):
    if y != 'O':
        print(x,y)
```

```
Peter B-per
Parker I-per
White B-org
House I-org
U.S B-org
Sunday B-tim
morning I-tim
White B-org
House I-org
```

Expected output:

```
Peter B-per
Parker I-per
White B-org
House I-org
Sunday B-tim
morning I-tim
White B-org
House I-org
```