Northeastern University, Khoury College of Computer Science

CS 6120 — Assignment 6

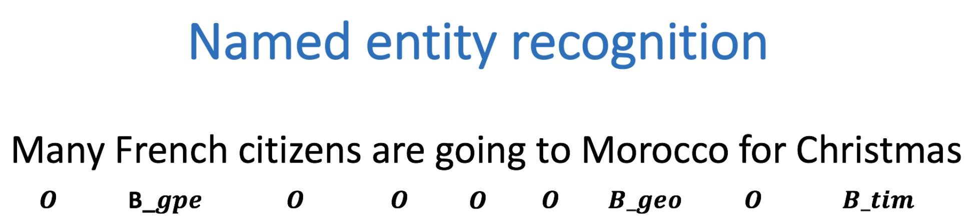
Due: March 20, 2025 (100 points)

YOUR NAME + LDAP

link-to-your-repository

In this assignment, we will building complicated recurrent neural networks (RNNs) with Tensorflow and Keras, while applying them to recognize named entities, i.e., Named Entity Recognition (NER). At the end of this assignment, you will have designed, implemented, trained, and evaluated an LSTM neural network. To do so, you will gain understanding of the data engineering needed for the training data, including featurization, word padding, and label representations.

Named Entity Recognition is a task that locates, identifies, and classifies specific types of data in a large set of unstructured text. These *types of data* are called *named entities*, which are real-world objects like person names, organizations, products, time expressions, quantities, monetary values, percentages, etc.: i.e., anything that can be named. For example, in the below, the words *French*, *Morocco*, and *Christmas* are labeled as `geopolitical entity`, `geopolitical entity`, and `time indicator`, and everything else is labeled as `O`. Other tags that you might expect to see include `person`, `tim` (time indicator), `art` (artifact), etc.



For this homework, you will need `numpy` and `tensorflow`. This assignment will be heavily structured and should provide a step by step account of how to engineer the RNN. We borrow heavily from DeepLearning.AI and credit this assignment with their materials, downloaded in the zip file at the course website.

Download and Setup

This assignment uses the named entity tagging dataset, originally assembled at Kaggle, and cleaned by DeepLearning.AI. Download it here:

https://course.ccs.neu.edu/cs6120s25/data/named-entities/ner-data.zip

In this zip file, you will obtain all the NER data, and a function that reads the data in, called `load_data.py`. To load the data in, you can simply import and call `load_data`, which will provide you with training features and training labels. We will be grading on `data/large`, but you can feel free to debug with `data/small`.

Question 1: Encode the Sentence (Features)

For this question in the function `sentence_vectorizer` in assignment6.py, replace the `None` modifiers with your own code. You will be making the appropriate sentence vectorizer layer in Tensorflow with the tf.keras.layers.TextVectorization layer. This layer transforms sentences (i.e., your raw text) into integers, so they can be fed into a model you will build in Q3. 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:

1. padding token: "", integer mapped: 0
2. unknown token "[UNK]", integer mapped: 1

You can test your function out with the following code

```
tf.keras.utils.set_random_seed(33) ## Do not change this line.
train_sentences = load_data('data/large/train/sentences.txt')
```

```
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}")
```

This produces the following output:

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

Question 2: Encode the Labels

For this question in the function `label_vectorizer` in assignment6.py, replace the `None` modifiers with your own code. This layer transforms labels (i.e., the entity labels) into integers so they can be fed into a model you will build in Q3. 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.

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.

Tag Maps

You will notice in the function signature of `label_vectorizer`, there is an argument called `tag_map`. This is a dictionary of labels, which will translate the labels to a numerical categorical number. For example, if the following is a usage pattern of `tag_map`:

```
In  [1]: tag_map["B-geo"]
Out [1]: 2
```

The prepositions in the tags mean:

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

For example, for the text "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", the tags would look like:

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. You can create this map yourself; note that there is a file called `tags.txt` in `data/large` and `data/small`.

Padding

You will additionally need to *pad* the labels because the number of labels must match the number of words. 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:

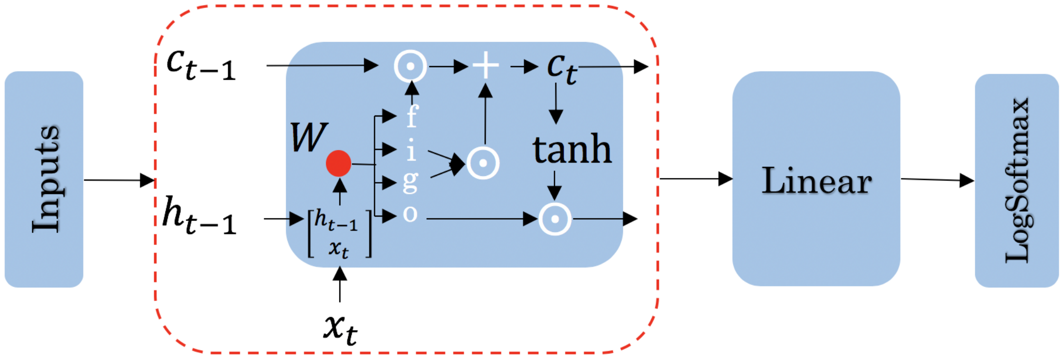
- Tensorflow has built-in functions for padding.
- Padding will be performed uniformly per dataset (train, validation, and test) using the maximum sentence length in each dataset and the size of each sentence is exactly the same as the size of their respective labels.

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`. 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 sequences will be padded at the beginning. You need to pass the argument `post`.
- `value`: Padding value. The default value is 0.

Question 3: Build the Model

You will now implement the model that will be able to determine the tags of sentences with the following model architecture as follows.



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.

Q 3.1: Model Architecture: LSTM

For this question in the function `NER` in assignment6.py, you will implement a recurrent LSTM neural network model with the architecture shown above. 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: 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 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: 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: An LSTM layer. Arguments you'll 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, since there will be an output for each word in the sentence, it is `(batch_size, sentence_length, units)`.

- tf.keras.layers.Dense: The parameters for this layer are:
  1. `units`: It is the number of units chosen for this dense layer, i.e., it is the dimensionality of the output space. In this case, each value passed through the Dense layer must be mapped into a vector with length `num_of_classes` (in this case, `len(tags)`).
  2. `activation`: This is the activation that will be performed after computing the values in the Dense layer. Since the Dense layer comes before the LogSoftmax step, you can pass the LogSoftmax function as activation function here. You can find the implementation for LogSoftmax under `tf.nn`. So you may call it as `tf.nn.log_softmax`. See its documentation here.

Q 3.2: Model Loss Function

You will also need the loss function for your model before compiling. Our loss is different, in that we need to ignore the padding. There is actually a parameter that you can set in the function `tf.keras.losses.SparseCategoricalCrossentropy`, which will disregard entries with a particular value.

For this question in the function `masked_loss` in assignment6.py, you will implement the cross-entropy function that ignores sentence padding.

Q 3.3: Model Evaluation

While training with `model.fit`, it is helpful to evaluate the accuracy of your model to see if it is learning effectively. For this question in the function `masked_accuracy` in assignment6.py, you will implement an accuracy metric for use in validation and evaluation.

Question 4: Train Your Model!

For model data ingestion, it is easy to pass as input a `tf.Dataset` object. For this, we can use datasets from tensor slices. In the function below, by setting the `tfdata` flag, you can use the vectorizers you had created in Q1 and Q2.

```
def generate_dataset(sentences, labels, sentence_vectorizer,
                     label_vectorizer, tag_map, tfdata=True):
    '''
    Returns a tf.Dataset or the vectorizers, depending on whether or
    not tfdata = True.
    '''
    sentences_ids = sentence_vectorizer(sentences)
    labels_ids = label_vectorizer(labels, tag_map = tag_map)

    if tfdata:  # User wants a tf.Dataset
      dataset = tf.data.Dataset.from_tensor_slices((sentences_ids, labels_ids))
      return dataset
    else:       # User just wants the vectorizers
      return sentences_ids, labels_ids
```

Once you've generated your `tf.Dataset` object, you can pass it to the `fit` method after compiling your model.

```
model = NER(len(tag_map), len(vocab))
model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
              loss = masked_loss,
               metrics = [masked_accuracy])
model.fit(train_dataset.batch(BATCH_SIZE),
          validation_data = val_dataset.batch(BATCH_SIZE),
          shuffle=True,
          epochs = 1, steps_per_epoch=100)
```

Go ahead and train your model on your data. When you've finished, save the model to a file called `assignment6.h5`. You can do so with keras's model.save functionality. Upload this file to Gradescope.

Question 5: Model Prediction

For this question in the function `predict` in assignment6.py, 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.
You can test out your prediction function with the following examples.

```
# New york times news:
sentence = "Peter Parker , the White House director of trade and manufacturing
            policy of U.S , said in an interview on Sunday morning that the White
            House was working to prepare for the possibility of a second wave of
            the coronavirus in the fall , though he said it wouldn ’t necessarily
            come"
predictions = predict(sentence, model, sentence_vectorizer, tag_map)
for x,y in zip(sentence.split(’ ’), predictions):
    if y != ’O’:
        print(x,y)
```