

# Assignment 1: Deep N-grams

Welcome to the first graded assignment of course 3. In this assignment you will explore Recurrent Neural Networks (RNN).

In this notebook you will apply the following steps:

- Convert a line of text into a tensor
- Create a tensorflow dataset
- Define a GRU model using TensorFlow
- Train the model using TensorFlow
- Compute the accuracy of your model using the perplexity
- Generate text using your own model

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.

[Edit Meta](#)

## Table of Contents

- [Overview](#)
- [1 - Data Preprocessing Overview](#)
  - [1.1 - Loading in the Data](#)
  - [1.2 - Create the vocabulary](#)
  - [1.3 - Convert a Line to Tensor](#)
    - [Exercise 1 - line to tensor](#)
  - [1.4 - Prepare your data for training and testing](#)
  - [1.5 - Tensorflow dataset](#)
  - [1.6 - Create the input and the output for your model](#)
    - [Exercise 2 - data generator](#)
  - [1.7 - Create the training dataset](#)
- [2 - Defining the GRU Language Model \(GRULM\)](#)
  - [Exercise 3 - GRULM](#)
- [3 - Training](#)
  - [Exercise 4 - train\\_model](#)
- [4 - Evaluation](#)
  - [4.1 - Evaluating using the Deep Nets](#)
  - [Exercise 5 - log\\_perplexity](#)
- [5 - Generating Language with your Own Model](#)
  - [Optional Exercise 6 - GenerativeModel \(Not graded\)](#)
- [On statistical methods](#)

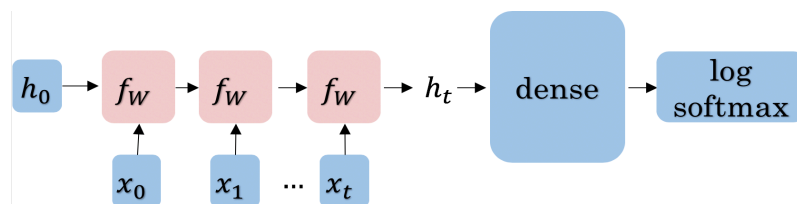
[Edit Meta](#)

## Overview

In this lab, you'll delve into the world of text generation using Recurrent Neural Networks (RNNs). Your primary objective is to predict the next set of characters based on the preceding ones. This seemingly straightforward task holds immense practicality in applications like predictive text and creative writing.

The journey unfolds as follows:

- **Data Preprocessing:** You'll start by converting lines of text into numerical tensors, making them machine-readable.
- **Dataset Creation:** Next, you'll create a TensorFlow dataset, which will serve as the backbone for supplying data to your model.
- **Neural Network Training:** Your model will be trained to predict the next set of characters, specifying the desired output length.
- **Character Embeddings:** Character embeddings will be employed to represent each character as a vector, a fundamental technique in natural language processing.
- **GRU Model:** Your model utilizes a Gated Recurrent Unit (GRU) to process character embeddings and make sequential predictions. The following figure gives you a summary of what you are about to implement.



- **Prediction Process:** The model's predictions are achieved through a linear layer and log-softmax computation.

This overview sets the stage for your exploration of text generation. Get ready to unravel the secrets of language and embark on a journey into the realm of creative writing and predictive text generation.

And as usual let's start by importing all the required libraries.

In [1]:

```
import os
import traceback
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

import numpy as np
import random as rnd

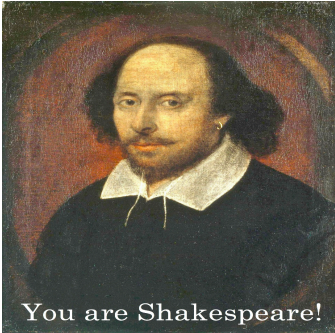
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.layers import Input

from termcolor import colored

# set random seed
```

In [2]:

1 - Data Preprocessing Overview



You are Shakespeare!

In this section, you will prepare the data for training your model. The data preparation involves the following steps:

- Dataset Import: Begin by importing the dataset. Each sentence is structured as one line in the dataset. To ensure consistency, remove any extra space from these lines using the `strip` function.
- Data Storage: Store each cleaned line in a list. This list will serve as the foundational dataset for your text generation task.
- Character-Level Processing: Since the goal is character generation, it's essential to process the text at the character level, not the word level. This involves converting each individual character into a numerical representation. To achieve this:
  - Use the `tf.strings.unicode_split` function to split each sentence into its constituent characters.
  - Utilize `tf.keras.layers.StringLookup` to map these characters to integer values. This transformation lays the foundation for character-based modeling.
- TensorFlow Dataset Creation: Create a TensorFlow dataset capable of producing data in batches. Each batch will consist of `batch_size` sentences, with each sentence containing a maximum of `max_length` characters. This organized dataset is essential for training your character generation model.

These preprocessing steps ensure that your dataset is meticulously prepared for the character-based text generation task, allowing you to work seamlessly with the Shakespearean corpus data.

## 1.1 - Loading in the Data

In [3]:

```
dirname = 'data/'
filename = 'shakespeare_data.txt'
lines = [] # storing all the lines in a variable.

counter = 0

with open(os.path.join(dirname, filename)) as files:
    for line in files:
        # remove leading and trailing whitespace
        pure_line = line.strip().lower()

        # if pure_line is not the empty string,
        if pure_line:
            # append it to the list
            lines.append(pure_line)

n_lines = len(lines)

Number of lines: 125097
```

Let's examine a few lines from the corpus. Pay close attention to the structure and style employed by Shakespeare in this excerpt. Observe that character names are written in uppercase, and each line commences with a capital letter. Your task in this exercise is to construct a generative model capable of emulating this particular structural style.

In [4]:

BENVOLIO            Here were the servants of your adversary,  
 And yours, close fighting ere I did approach:  
 I drew to part them: in the instant came  
 The fiery Tybalt, with his sword prepared,  
 Which, as he breathed defiance to my ears,  
 He swung about his head and cut the winds,  
 Who nothing hurt withal hiss'd him in scorn:  
 While we were interchanging thrusts and blows,

Edit Meta

## 1.2 - Create the vocabulary

In the following code cell, you will create the vocabulary for text processing. The vocabulary is a crucial component for understanding and processing text data. Here's what the code does:

- Concatenate all the lines in our dataset into a single continuous text, separated by line breaks.
- Identify and collect the unique characters that make up the text. This forms the basis of our vocabulary.
- To enhance the vocabulary, introduce two special characters:
  - [UNK]: This character represents any unknown or unrecognized characters in the text.
  - "" (empty character): This character is used for padding sequences when necessary.
- The code concludes with the display of statistics, showing the total count of unique characters in the vocabulary and providing a visual representation of the complete character set.

In [5]:

Edit Meta

```
text = "\n".join(lines)
# The unique characters in the file
vocab = sorted(set(text))
vocab.insert(0, "[UNK]") # Add a special character for any unknown
vocab.insert(1, "") # Add the empty character for padding.

print(f'{len(vocab)} unique characters')
```

82 unique characters  
 [UNK]  
 ! \$ % ' ( ) , - . 0 1 2 3 4 5 6 7 8 9 : ; ? A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ ] a b c d e f g h i j k  
 m n o p q r s t u v w x y z |

Edit Meta

## 1.3 - Convert a Line to Tensor

Now that you have your list of lines, you will convert each character in that list to a number using the order given by your vocabulary. You can use `tf.strings.unicode_split` to split the text into characters.

In [6]:

Edit Meta

```
line = "Hello world!"
chars = tf.strings.unicode_split(line, input_encoding='UTF-8')
tf.Tensor([b'H' b'e' b'l' b'l' b'o' b' ' b'w' b'o' b'r' b'l' b'd' b'!'], shape=(12,), dtype=string)
```

Edit Meta

Using your vocabulary, you can convert the characters given by `unicode_split` into numbers. The number will be the index of the character in the given vocabulary.

In [7]:

Edit Meta

```
print(vocab.index('a'))
print(vocab.index('e'))
print(vocab.index('i'))
print(vocab.index('o'))
print(vocab.index('u'))
print(vocab.index(' '))
print(vocab.index('2'))
```

55  
 59  
 63  
 69  
 75  
 4  
 16  
 17

Edit Meta

Tensorflow has a function `tf.keras.layers.StringLookup` that does this efficiently for list of characters. Note that the output object is of type `tf.Tensor`. Here is the result of applying the StringLookup function to the characters of "Hello world"

In [8]:

Edit Meta

```
ids = tf.keras.layers.StringLookup(vocabulary=list(vocab), mask_token=None)(chars)
tf.Tensor([34 59 66 66 69 4 77 69 72 66 58 5], shape=(12,), dtype=int64)
```

[Edit Meta](#)

## Exercise 1 - line\_to\_tensor

**Instructions:** Write a function that takes in a single line and transforms each character into its unicode integer. This returns a list of integers, which we'll re to as a tensor.

In [32]:

[Edit Meta](#)

```
# GRADED FUNCTION: line_to_tensor
def line_to_tensor(line, vocab):
    """
    Converts a line of text into a tensor of integer values representing characters.

    Args:
        line (str): A single line of text.
        vocab (list): A list containing the vocabulary of unique characters.

    Returns:
        tf.Tensor(dtype=int64): A tensor containing integers corresponding to the characters in the `line`.
    """
    ### START CODE HERE ###

    # Split the input line into individual characters
    chars = tf.strings.unicode_split(line, input_encoding='UTF-8')

    # Map characters to their respective integer values using StringLookup
    ids = tf.keras.layers.StringLookup(vocabulary=vocab, mask_token=None, output_mode='int', dtype=tf.int64)

    # Apply the Lookup Layer to convert characters to integers
    char_tensor = ids(chars)
    ### END CODE HERE ###

    return char_tensor
```

In [33]:

[Edit Meta](#)

```
# Test your function
tmp_ids = line_to_tensor('abc xyz', vocab)
print(f"Result: {tmp_ids}")
```

Result: [55 56 57 4 78 79 80]

Output type: &lt;class 'tensorflow.python.framework.ops.EagerTensor'&gt;

[Edit Meta](#)

### Expected output

Result: [55 56 57 4 78 79 80]

Output type: &lt;class 'tensorflow.python.framework.ops.EagerTensor'&gt;

In [34]:

[Edit Meta](#)

# UNIT TEST

All test passed!

[Edit Meta](#)

You will also need a function that produces text given a numeric tensor. This function will be useful for inspection when you use your model to generate new text, because you will be able to see words rather than lists of numbers. The function will use the inverse Lookup function `tf.keras.layers.StringLookup` with `invert=True` in its parameters.

In [35]:

[Edit Meta](#)

```
def text_from_ids(ids, vocab):
    """
    Converts a tensor of integer values into human-readable text.

    Args:
        ids (tf.Tensor): A tensor containing integer values (unicode IDs).
        vocab (list): A list containing the vocabulary of unique characters.

    Returns:
        str: A string containing the characters in human-readable format.
    """
    # Initialize the StringLookup Layer to map integer IDs back to characters
    chars_from_ids = tf.keras.layers.StringLookup(vocabulary=vocab, invert=True, mask_token=None)

    # Use the layer to decode the tensor of IDs into human-readable text
```

[Edit Meta](#)

Use the function for decoding the tensor produced by "Hello world!"

In [36]:

Edit Meta

Out[36]: b'Hello world!'

Edit Meta

## 1.4 - Prepare your data for training and testing

As usual, you will need some data for training your model, and some data for testing its performance. So, we will use 124097 lines for training and 1000 lines for testing.

In [37]:

Edit Meta

```
train_lines = lines[:-1000] # Leave the rest for training
eval_lines = lines[-1000:] # Create a holdout validation set

print(f"Number of training lines: {len(train_lines)}")
```

Number of training lines: 124097  
Number of validation lines: 1000

Edit Meta

## 1.5 - TensorFlow dataset

Most of the time in Natural Language Processing, and AI in general you use batches when training your models. Here, you will build a dataset that takes in some text and returns a batch of text fragments (Not necessarily full sentences) that you will use for training.

- The generator will produce text fragments encoded as numeric tensors of a desired length

Once you create the dataset, you can iterate on it like this:

```
data_generator.take(1)
```

This generator returns the data in a format that you could directly use in your model when computing the feed-forward of your algorithm. This batch dataset generator returns batches of data in an endless way.

So, let's check how the different parts work with a corpus composed of 2 lines. Then, you will use these parts to create the first graded function of this notebook.

In order to get a dataset generator that produces batches of fragments from the corpus, you first need to convert the whole text into a single line, and then transform it into a single big tensor. This is only possible if your data fits completely into memory, but that is the case here.

In [38]:

Edit Meta

```
all_ids = line_to_tensor("\n".join(["Hello world!", "Generative AI"]), vocab)
```

Out[38]: <tf.Tensor: shape=(26,), dtype=int64, numpy=
array([34, 59, 66, 66, 69, 4, 77, 69, 72, 66, 58, 5, 3, 33, 59, 68, 59,
 72, 55, 74, 63, 76, 59, 4, 27, 35])>

Edit Meta

Create a dataset out of a tensor like input. This initial dataset will dispatch numbers in packages of a specified length. For example, you can use it for getting the 10 first encoded characters of your dataset. To make it easier to read, we can use the `text_from_ids` function.

In [39]:

Edit Meta

```
ids_dataset = tf.data.Dataset.from_tensor_slices(all_ids)

[b'H', b'e', b'l', b'l', b'o', b' ', b'w', b'o', b'r', b'l']
```

Edit Meta

But we can configure this dataset to produce batches of the same size each time. We could use this functionality to produce text fragments of a desired size (`seq_length + 1`). We will explain later why you need an extra character into the sequence.

In [40]:

Edit Meta

```
seq_length = 10
```

Edit Meta

You can verify that the data generator produces encoded fragments of text of the desired length. For example, let's ask the generator to produce 2 batches of data using the function `data_generator.take(2)`

In [41]:

Edit Meta

```
for seq in data_generator.take(2):
    tf.Tensor([34 59 66 66 69  4 77 69 72 66 58], shape=(11,), dtype=int64)
    tf.Tensor([ 5  3 33 59 68 59 72 55 74 63 76], shape=(11,), dtype=int64)
```

Edit Meta

But as usual, it is easier to understand if you print it in human readable characters using the `'text_from_ids'` function.

In [42]:

```
i = 1
for seq in data_generator.take(2):
    print(f"{i}. {text_from_ids(seq, vocab).numpy()}")
```

1. b'Hello world'  
2. b'!\nGenerativ'

Edit Meta

Edit Meta

## 1.6 - Create the input and the output for your model

In this task you have to predict the next character in a sequence. The following function creates 2 tensors, each with a length of `seq_length` out of the in sequence of length `seq_length + 1`. The first one contains the first `seq_length` elements and the second one contains the last `seq_length` elements. For example, if you split the sequence `['H', 'e', 'l', 'l', 'o']`, you will obtain the sequences `['H', 'e', 'l', 'l']` and `['e', 'l', 'l', 'o']`.

In [43]:

```
def split_input_target(sequence):
    """
    Splits the input sequence into two sequences, where one is shifted by one position.

    Args:
        sequence (tf.Tensor or list): A list of characters or a tensor.

    Returns:
        tf.Tensor, tf.Tensor: Two tensors representing the input and output sequences for the model.
    """
    # Create the input sequence by excluding the last character
    input_text = sequence[:-1]
    # Create the target sequence by excluding the first character
    target_text = sequence[1:]
```

Edit Meta

Edit Meta

Look the result using the following sequence of characters

In [44]:

```
Out[44]: (['T', 'e', 'n', 's', 'o', 'r', 'f', 'l', 'o'],
          ['e', 'n', 's', 'o', 'r', 'f', 'l', 'o', 'w'])
```

Edit Meta

Edit Meta

The first sequence will be the input and the second sequence will be the expected output

Edit Meta

Now, put all this together into a function to create your batch dataset generator

## Exercise 2 - data\_generator

**Instructions:** Create a batch dataset from the input text. Here are some things you will need.

- Join all the input lines into a single string. When you have a big dataset, you would better use a flow from directory or any other kind of generator.
- Transform your input text into numeric tensors
- Create a TensorFlow DataSet from your numeric tensors: Just feed the numeric tensors into the function `tf.data.Dataset.from_tensor_slices`
- Make the dataset produce batches of data that will form a single sample each time. This is, make the dataset produce a sequence of `seq_length + 1` rather than single numbers at each time. You can do it using the `batch` function of the already created dataset. You must specify the length of the produced sequences (`seq_length + 1`). So, the sequence length produced by the dataset will `seq_length + 1`. It must have that extra element so you will get the input and the output sequences out of the same element. `drop_remainder=True` will drop the sequences that do not have the required length. This could happen each time that the dataset reaches the end of the input sequence.
- Use the `split_input_target` to split each element produced by the dataset into the mentioned input and output sequences. The input will have the first `seq_length` elements, and the output will have the last `seq_length`. So, after this step, the dataset generator will produce batches of pairs (input/output) sequences.
- Create the final dataset, using `dataset_xy` as the starting point. You will configure this dataset to shuffle the data during the generation of the data with the specified `BUFFER_SIZE`. For performance reasons, you would like that tensorflow pre-process the data in parallel with training. That is called `prefetching`, and it will be configured for you.

In [45]:

Edit Meta

```
# GRADED FUNCTION: create_batch_dataset
def create_batch_dataset(lines, vocab, seq_length=100, batch_size=64):
    """
    Creates a batch dataset from a list of text lines.

    Args:
        lines (list): A list of strings with the input data, one line per row.
        vocab (list): A list containing the vocabulary.
        seq_length (int): The desired length of each sample.
        batch_size (int): The batch size.

    Returns:
        tf.data.Dataset: A batch dataset generator.
    """
    # Buffer size to shuffle the dataset
    # (TF data is designed to work with possibly infinite sequences,
    # so it doesn't attempt to shuffle the entire sequence in memory. Instead,
    # it maintains a buffer in which it shuffles elements).
    BUFFER_SIZE = 10000

    # For simplicity, just join all lines into a single line
    single_line_data = "\n".join(lines)

    ### START CODE HERE ###

    # Convert your data into a tensor using the given vocab
    all_ids = line_to_tensor(single_line_data, vocab)
    # Create a TensorFlow dataset from the data tensor
    ids_dataset = tf.data.Dataset.from_tensor_slices(all_ids)
    # Create a batch dataset
    data_generator = ids_dataset.batch(seq_length + 1, drop_remainder=True)
    # Map each input sample using the split_input_target function
    dataset_xy = data_generator.map(split_input_target)

    # Assemble the final dataset with shuffling, batching, and prefetching
    dataset = (
        dataset_xy
        .shuffle(BUFFER_SIZE)
        .batch(batch_size, drop_remainder=True)
        .prefetch(tf.data.experimental.AUTOTUNE)
    )

    ### END CODE HERE ###
```

In [46]:

Edit Meta

```
# test your function
tf.random.set_seed(1)
dataset = create_batch_dataset(train_lines[1:100], vocab, seq_length=16, batch_size=2)

print("Prints the elements into a single batch. The batch contains 2 elements: ")

for input_example, target_example in dataset.take(1):
    print("\n\033[94mInput0\t:", text_from_ids(input_example[0], vocab).numpy())
    print("\n\033[93mTarget0\t:", text_from_ids(target_example[0], vocab).numpy())

    print("\n\033[94mInput1\t:", text_from_ids(input_example[1], vocab).numpy())
```

Prints the elements into a single batch. The batch contains 2 elements:

Input0 : b'and sight distra'

Target0 : b'nd sight distrac'

Input1 : b'when in his fair'

Target1 : b'hen in his fair '

Edit Meta

#### Expected output

Prints the elements into a single batch. The batch contains 2 elements:

Input0 : b'and sight distra'

Target0 : b'nd sight distrac'

Input1 : b'when in his fair'

Target1 : b'hen in his fair '

In [47]:

Edit Meta

# UNIT TEST

All test passed!

Edit Meta

## 1.7 - Create the training dataset

Now, you can generate your training dataset using the functions defined above. This will produce pairs of input/output tensors each time the batch generator creates an entry.

In [62]:

Edit Meta

# Batch size

BATCH\_SIZE = 64

Edit Meta

## 2 - Defining the GRU Language Model (GRULM)

Now that you have the input and output tensors, you will go ahead and initialize your model. You will be implementing the GRULM, gated recurrent unit model. To implement this model, you will be using TensorFlow. Instead of implementing the GRU from scratch (you saw this already in a lab), you will use the necessary methods from a built-in package. You can use the following packages when constructing the model:

- `tf.keras.layers.Embedding`: Initializes the embedding. In this case it is the size of the vocabulary by the dimension of the model. [docs](#)
  - `Embedding(vocab_size, embedding_dim)`.
  - `vocab_size` is the number of unique words in the given vocabulary.
  - `embedding_dim` is the number of elements in the word embedding (some choices for a word embedding size range from 150 to 300, for example).
- `tf.keras.layers.GRU`: TensorFlow GRU layer. [docs](#)) Builds a traditional GRU of `rnn_units` with dense internal transformations. You can read the paper here: <https://arxiv.org/abs/1412.3555>
  - `units`: Number of recurrent units in the layer. It must be set to `rnn_units`.
  - `return_sequences`: It specifies if the model returns a sequence of predictions. Set it to `True`.
  - `return_state`: It specifies if the model must return the last internal state along with the prediction. Set it to `True`.
- `tf.keras.layers.Dense`: A dense layer. [docs](#). You must set the following parameters:
  - `units`: Number of units in the layer. It must be set to `vocab_size`.
  - `activation`: It must be set to `log_softmax` function as described in the next line.
- `tf.nn.log_softmax`: Log of the output probabilities. [docs](#)
  - You don't need to set any parameters, just set the activation parameter as `activation=tf.nn.log_softmax`.

## Exercise 3 - GRULM

**Instructions:** Implement the GRULM class below. You should be using all the methods explained above.



In [63]:

Edit Meta

```
# GRADED CLASS: GRULM
class GRULM(tf.keras.Model):
    """
    A GRU-based language model that maps from a tensor of tokens to activations over a vocabulary.

    Args:
        vocab_size (int, optional): Size of the vocabulary. Defaults to 256.
        embedding_dim (int, optional): Depth of embedding. Defaults to 256.
        rnn_units (int, optional): Number of units in the GRU cell. Defaults to 128.

    Returns:
        tf.keras.Model: A GRULM language model.
    """
    def __init__(self, vocab_size=256, embedding_dim=256, rnn_units=128):
        super().__init__(self)

        ### START CODE HERE ###

        # Create an embedding layer to map token indices to embedding vectors
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        # Define a GRU (Gated Recurrent Unit) layer for sequence modeling
        self.gru = tf.keras.layers.GRU(rnn_units, return_sequences=True, return_state=True)
        # Apply a dense layer with log-softmax activation to predict next tokens
        self.dense = tf.keras.layers.Dense(vocab_size, activation=tf.nn.log_softmax)

        ### END CODE HERE ###

    def call(self, inputs, states=None, return_state=False, training=False):
        x = inputs
        # Map input tokens to embedding vectors
        x = self.embedding(x, training=training)
        if states is None:
            # Get initial state from the GRU layer
            states = self.gru.get_initial_state(x)
        x, states = self.gru(x, initial_state=states, training=training)
        # Predict the next tokens and apply log-softmax activation
        x = self.dense(x, training=training)
        if return_state:
            return x, states
        else:
            return x
```

Edit Meta

Now, you can define a new GRULM model. You must set the `vocab_size` to 82; the size of the embedding `embedding_dim` to 256, and the number of units that will have you recurrent neural network `rnn_units` to 512

In [64]:

Edit Meta

```
# Length of the vocabulary in StringLookup Layer
vocab_size = 82

# The embedding dimension
embedding_dim = 256

# RNN Layers
rnn_units = 512

model = GRUML(
    vocab_size=vocab_size,
    embedding_dim=embedding_dim,
```

In [65]:

Edit Meta

```
# testing your model

try:
    # Simulate inputs of length 100. This allows to compute the shape of all inputs and outputs of our network
    model.build(input_shape=(BATCH_SIZE, 100))
    model.call(Input(shape=(100)))
    model.summary()
except:
    print("\033[91mError! \033[0mA problem occurred while building your model. This error can occur due to wrong initializat
    traceback.print_exc()
```

Model: "grulm\_10"

Layer (type)	Output Shape	Param #
embedding_10 (Embedding)	(None, 100, 256)	20992
gru_10 (GRU)	[(None, 100, 512), (None, 512)]	1182720
dense_10 (Dense)	(None, 100, 82)	42066

=====  
 Total params: 1245778 (4.75 MB)  
 Trainable params: 1245778 (4.75 MB)  
 Non-trainable params: 0 (0.00 Byte)

Edit Meta

**Expected output**

Model: "grulm"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 100, 256)	20992
gru (GRU)	[(None, 100, 512), (None, 512)]	1182720
dense (Dense)	(None, 100, 82)	42066

=====  
 Total params: 1245778 (4.75 MB)  
 Trainable params: 1245778 (4.75 MB)  
 Non-trainable params: 0 (0.00 Byte)

In [66]:

Edit Meta

# UNIT TEST

Test case 1:  
 All tests passed!  
 Test case 2:  
 All tests passed!

Edit Meta

Now, let's use the model for predicting the next character using the untrained model. At the beginning the model will generate only gibberish.

In [67]:

Edit Meta

```
for input_example_batch, target_example_batch in dataset.take(1):
    print("Input: ", input_example_batch[0].numpy()) # Lets use only the first sequence on the batch
    example_batch_predictions = model(tf.constant([input_example_batch[0].numpy()]))
```

Input: [73 74 4 74 62 59 4 74 77 63 66 63 61 62 74 4 69 60 4 73 75 57 62 4  
 58 55 79 3 27 73 4 55 60 74 59 72 4 73 75 68 73 59 74 4 60 55 58 59  
 74 62 4 63 68 4 74 62 59 4 77 59 73 74 11 3 49 62 63 57 62 4 56 79  
 4 55 68 58 4 56 79 4 56 66 55 57 65 4 68 63 61 62 74 4 58 69 74 62  
 4 74 55 65]

(1, 100, 82) # (batch\_size, sequence\_length, vocab\_size)

Edit Meta

The output size is (1, 100, 82). We predicted only on the first sequence generated by the batch generator. 100 is the number of predicted characters. It has exactly the same length as the input. And there are 82 values for each predicted character. Each of these 82 real values are related to the logarithm likelihood of each character to be the next one in the sequence. The bigger the value, the higher the likelihood. As the network is not trained yet, all those values must be very similar and random. Just check the values for the last prediction on the sequence.

In [68]:

Edit Meta

```
Out[68]: array([-4.4088335, -4.4119616, -4.393828 , -4.413433 , -4.4189663,
               -4.4034944, -4.4075117, -4.407446 , -4.415791 , -4.4138393,
               -4.4223967, -4.396336 , -4.42516 , -4.3974514, -4.411654 ,
               -4.3826222, -4.428223 , -4.4171233, -4.4142256, -4.4057827,
               -4.4013367, -4.411088 , -4.410768 , -4.4142027, -4.4022512,
               -4.3959484, -4.4166875, -4.407511 , -4.389319 , -4.4163833,
               -4.404793 , -4.4176474, -4.41274 , -4.3959994, -4.4114213,
               -4.3845882, -4.3901653, -4.393563 , -4.4084597, -4.414881 ,
               -4.40546 , -4.4229794, -4.4214263, -4.4062247, -4.4125457,
               -4.386656 , -4.4163156, -4.4005685, -4.4077897, -4.394546 ,
               -4.396066 , -4.41906 , -4.392824 , -4.3910785, -4.414643 ,
               -4.4106984, -4.3851995, -4.401676 , -4.395236 , -4.398728 ,
               -4.4248853, -4.406341 , -4.4219065, -4.39669 , -4.4104176,
               -4.3961096, -4.4158907, -4.4001155, -4.413916 , -4.395164 ,
               -4.393838 , -4.404393 , -4.427202 , -4.401356 , -4.409143 ,
               -4.4015627, -4.3999166, -4.4142365, -4.417395 , -4.4169803,
               -4.4067388, -4.405907 ], dtype=float32)
```

Edit Meta

And the simplest way to choose the next character is by getting the index of the element with the highest likelihood. So, for instance, the prediction for the l characeter would be:

In [69]:

```
last_character = tf.math.argmax(example_batch_predictions[0][99])
15
```

Edit Meta

And the prediction for the whole sequence would be:

In [70]:

```
sampled_indices = tf.math.argmax(example_batch_predictions[0], axis=1)
[19 67 58 58 13 9 58 58 58 11 50 68 50 56 65 58 35 26 58 72 9 64 56 9
 11 11 56 55 2 76 76 52 20 20 9 9 9 19 67 57 72 19 58 58 20 52 11 9
 9 9 58 68 68 57 58 13 9 58 58 19 19 58 10 55 55 55 68 44 56 57 56 56
 24 52 57 11 24 11 56 24 63 2 50 56 28 24 57 68 44 56 65 58 11 19 44 9
 58 58 52 15]
```

Edit Meta

Those 100 numbers represent 100 predicted characters. However, humans cannot read this. So, let's print the input and output sequences using our `text_from_ids` function, to check what is going on.

In [71]:

```
print("Input:\n", text_from_ids(input_example_batch[0], vocab))
print()
```

Edit Meta

Input:

```
tf.Tensor(b'st the twilight of such day\nAs after sunset fadeth in the west,\nWhich by and by black night doth tak', shape=
dtype=string)
```

Next Char Predictions:

```
tf.Tensor(b'5mdd.(ddd,XnXbkdI?dr(jb(,,ba\trvvZ66(((5mcr5dd6Z,(((dnncd.(dd55d)aaanRbcbb:Zc,:,b:i\trxB:cnRbkD,5R(ddZ1', shape=
dtype=string)
```

Edit Meta

As expected, the untrained model just produces random text as response of the given input. It is also important to note that getting the index of the maximu score is not always the best choice. In the last part of the notebook you will see another way to do it.

Edit Meta

### 3 - Training

Now you are going to train your model. As usual, you have to define the cost function and the optimizer. You will use the following built-in functions provide TensorFlow:

- `tf.losses.SparseCategoricalCrossentropy()`: The Sparce Categorical Cross Entropy loss. It is the loss function used for multiclass classificati
  - `from_logits=True`: This parameter informs the loss function that the output values generated by the model are not normalized like a probability distribution. This is our case, since our GRULM model uses a `log_softmax` activation rather than the `softmax`.
- `tf.keras.optimizers.Adam`: Use Adaptive Moment Estimation, a stochastic gradient descent method optimizer that works well in most of the case: Set the `learning_rate` to 0.00125.

#### Exercise 4 - train\_model

**Instructions:** Compile the GRULM model using a `SparseCategoricalCrossentropy` loss and the `Adam` optimizer

In [72]:

```
# GRADED FUNCTION: Compile model

def compile_model(model):
    """
    Sets the loss and optimizer for the given model

    Args:
        model (tf.keras.Model): The model to compile.

    Returns:
        tf.keras.Model: The compiled model.
    """
    ### START CODE HERE ###

    # Define the loss function. Use SparseCategoricalCrossentropy
    loss = tf.losses.SparseCategoricalCrossentropy(from_logits=True)

    # Define and Adam optimizer
    opt = tf.keras.optimizers.Adam(learning_rate=0.00125)
    # Compile the model using the parametrized Adam optimizer and the SparseCategoricalCrossentropy function
    model.compile(optimizer=opt, loss=loss)

    ### END CODE HERE ###
```

In [73]:

```
## UNIT TEST

All test passed!
```

Now, train your model for 10 epochs. With GPU this should take about one minute. With CPU this could take several minutes.

In [ ]:

```
EPOCHS = 10

# Compile the model
model = compile_model(model)
# Fit the model

Epoch 1/10
790/790 [=====] - 12s 11ms/step - loss: 2.0142
Epoch 2/10
790/790 [=====] - 9s 10ms/step - loss: 1.4789
Epoch 3/10
790/790 [=====] - 9s 10ms/step - loss: 1.3793
Epoch 4/10
494/790 [=====>.....] - ETA: 3s - loss: 1.3441
```

You can uncomment the following cell to save the weights of your model. This allows you to use the model later.

In [ ]:

```
# If you want, you can save the final model. Here is deactivated.
#output_dir = './model/'
#
#try:
#    shutil.rmtree(output_dir)
#except OSError as e:
#    pass
```

The model was only trained for 10 epochs. We pretrained a model for 30 epochs, which can take about 5 minutes in a GPU.

## 4 - Evaluation

### 4.1 - Evaluating using the Deep Nets

Now that you have learned how to train a model, you will learn how to evaluate it. To evaluate language models, we usually use perplexity which is a measure of how well a probability model predicts a sample. Note that perplexity is defined as:

$$P(W) = \frac{1}{|W|} \prod_{i=1}^N \frac{1}{P(w_i | w_1, \dots, w_{i-1})}$$

$$P(W) = \sqrt[N]{\prod_{i=1}^N \frac{1}{P(w_i | w_1, \dots, w_{i-1})}}$$

As an implementation hack, you would usually take the log of that formula (to enable us to use the log probabilities we get as output of our `RNN`, convert exponents to products, and products into sums which makes computations less complicated and computationally more efficient).

$$\begin{aligned}
 \log P(W) &= \log \left( \prod_{i=1}^N \frac{1}{P(w_i | w_1, \dots, w_{i-1})} \right) \\
 \log P(W) &= \log \left( \frac{1}{\prod_{i=1}^N P(w_i | w_1, \dots, w_{i-1})} \right) \\
 &= \log \left( \prod_{i=1}^N \frac{1}{P(w_i | w_1, \dots, w_{i-1})} \right)^{\frac{1}{N}} \\
 &= \log \left( \left( \prod_{i=1}^N \frac{1}{P(w_i | w_1, \dots, w_{i-1})} \right)^{\frac{1}{N}} \right) \\
 &= \log \left( \prod_{i=1}^N P(w_i | w_1, \dots, w_{i-1}) \right)^{-\frac{1}{N}} \\
 &= \log \left( \left( \prod_{i=1}^N P(w_i | w_1, \dots, w_{i-1}) \right)^{-\frac{1}{N}} \right) \\
 &= -\frac{1}{N} \log \left( \prod_{i=1}^N P(w_i | w_1, \dots, w_{i-1}) \right) \\
 &= -\frac{1}{N} \log \left( \prod_{i=1}^N P(w_i | w_1, \dots, w_{i-1}) \right) \\
 &= -\frac{1}{N} \sum_{i=1}^N \log P(w_i | w_1, \dots, w_{i-1}) \\
 &= -\frac{1}{N} \sum_{i=1}^N \log P(w_i | w_1, \dots, w_{i-1})
 \end{aligned}$$

### Exercise 5 - log\_perplexity

**Instructions:** Write a program that will help evaluate your model. Implementation hack: your program takes in `preds` and `target`. `preds` is a tensor of log probabilities. You can use `tf.one_hot` to transform the `target` into the same dimension. You then multiply them and sum them. For the sake of simplicity, we suggest you use the NumPy functions `sum`, `mean` and `equal`. Good luck!

[Edit Meta](#)

#### Hints

[Edit Meta](#)

In [ ]:

```

# GRADED FUNCTION: log_perplexity
def log_perplexity(preds, target):
    """
    Function to calculate the log perplexity of a model.

    Args:
        preds (tf.Tensor): Predictions of a list of batches of tensors corresponding to lines of text.
        target (tf.Tensor): Actual list of batches of tensors corresponding to lines of text.

    Returns:
        float: The log perplexity of the model.
    """
    PADDING_ID = 1
    ### START CODE HERE ###

    # Calculate log probabilities for predictions using one-hot encoding
    log_p = np.sum(preds * np.eye(preds.shape[-1])[target], axis=-1) # HINT: tf.one_hot() should replace one of the Nones
    # Identify non-padding elements in the target
    non_pad = 1.0 - np.equal(target, PADDING_ID) # You should check if the target equals to PADDING_ID
    # Apply non-padding mask to log probabilities to exclude padding
    log_p = log_p * non_pad # Get rid of the padding
    # Calculate the log perplexity by taking the sum of log probabilities and dividing by the sum of non-padding elements
    log_ppx = np.sum(log_p, axis=-1) / np.sum(non_pad, axis=-1) # Remember to set the axis properly when summing up
    # Compute the mean of log perplexity
    log_ppx = np.mean(log_ppx) # Compute the mean of the previous expression

    ### END CODE HERE ###

```

[Edit Meta](#)

In [ ]:

```

#UNIT TESTS

```

[Edit Meta](#)

Now load the provided pretrained model just to ensure that results are consistent for the upcoming parts of the notebook. You need to instantiate the GRU model and then load the saved weights.

In [ ]:

```
# Load the pretrained model. This step is optional.
vocab_size = len(vocab)
embedding_dim = 256
rnn_units = 512

model = GRULM(
    vocab_size=vocab_size,
    embedding_dim=embedding_dim,
    rnn_units = rnn_units)
model.build(input_shape=(100, vocab_size))
```

Edit Meta

Edit Meta

Now, you will use the 1000 lines of the corpus that were reserved at the beginning of this notebook as test data. You will apply the same preprocessing as you did for the train dataset: get the numeric tensor from the input lines, and use the `split_input_target` to generate the inputs and the expected outputs.

Second, you will predict the next characters for the whole dataset, and you will compute the perplexity for the expected outputs and the given predictions.

In [ ]:

```
#for line in eval_lines[1:3]:
eval_text = "\n".join(eval_lines)
eval_ids = line_to_tensor([eval_text], vocab)
input_ids, target_ids = split_input_target(tf.squeeze(eval_ids, axis=0))

preds, status = model(tf.expand_dims(input_ids, 0), training=False, states=None, return_state=True)

#Get the log perplexity
log_ppx = log_perplexity(preds, tf.expand_dims(target_ids, 0))
```

Edit Meta

Edit Meta

**Expected Output:** The log perplexity and perplexity of your model are around 1.22 and 3.40 respectively.

Edit Meta

So, the log perplexity of the model is 1.22. It is not an easy to interpret metric, but it can be used to compare among models. The smaller the value the better the model.

Edit Meta

## 5 - Generating Language with your Own Model

Your GRULM model demonstrates an impressive ability to predict the most likely characters in a sequence, based on log scores. However, it's important to acknowledge that this model, in its default form, is deterministic and can result in repetitive and monotonous outputs. For instance, it tends to provide the same answer to a question consistently.

To make your language model more dynamic and versatile, you can introduce an element of randomness into its predictions. This ensures that even if you feed the model in the same way each time, it will generate different sequences of text.

To achieve this desired behavior, you can employ a technique known as random sampling. When presented with an array of log scores for the N character your dictionary, you add an array of random numbers to this data. The extent of randomness introduced into the predictions is regulated by a parameter called "temperature". By comparing the random numbers to the original input scores, the model adapts its choices, offering diversity in its outputs.

This doesn't imply that the model produces entirely random results on each iteration. Rather, with each prediction, there is a probability associated with choosing a character other than the one with the highest score. This concept becomes more tangible when you explore the accompanying Python code.

In [ ]:

```
def temperature_random_sampling(log_probs, temperature=1.0):
    """Temperature Random sampling from a categorical distribution. The higher the temperature, the more
    random the output. If temperature is close to 0, it means that the model will just return the index
    of the character with the highest input log_score

    Args:
        log_probs (tf.Tensor): The log scores for each character in the dictionary
        temperature (number): A value to weight the random noise.
    Returns:
        int: The index of the selected character
    """
    # Generate uniform random numbers with a slight offset to avoid log(0)
    u = tf.random.uniform(minval=1e-6, maxval=1.0 - 1e-6, shape=log_probs.shape)

    # Apply the Gumbel distribution transformation for randomness
    g = -tf.math.log(-tf.math.log(u))

    # Adjust the logits with the temperature and choose the character with the highest score
```

Edit Meta

Edit Meta

Now, it's time to bring all the elements together for the exciting task of generating new text. The `GenerativeModel` class plays a pivotal role in this process, offering two essential functions:

1. `generate_one_step` : This function is your go-to method for generating a single character at a time. It accepts two key inputs: an initial input sequence and a state that can be thought of as the ongoing context or memory of the model. The function delivers a single character prediction and an updated state, which can be used as the context for future predictions.
2. `generate_n_chars` : This function takes text generation to the next level. It orchestrates the iterative generation of a sequence of characters. At each iteration, `generate_one_step` is called with the last generated character and the most recent state. This dynamic approach ensures that the generated

evolves organically, building upon the context and characters produced in previous steps. Each character generated in this process is collected and stored in the result list, forming the final output text.

## Optional Exercise 6 - GenerativeModel (Not graded)

**Instructions:** Implementing the One-Step Generator

In this task, you will create a function to generate a single character based on the input text, using the provided vocabulary and the trained model. Follow these steps to complete the `generate_one_step` function:

1. Start by transforming your input text into a tensor using the given vocab. This will convert the text into a format that the model can understand.
2. Utilize the trained model with the `input_ids` and the provided states to predict the next characters. Make sure to retrieve the updated states from this prediction because they are essential for the final output.
3. Since we are only interested in the next character prediction, keep only the result for the last character in the sequence.
4. Employ the temperature random sampling technique to convert the vector of scores into a single character prediction. For this step, you will use the `predicted_logits` obtained in the previous step and the temperature parameter of the model.
5. To transform the numeric prediction into a human-readable character, use the `text_from_ids` function. Be mindful that `text_from_ids` expects a list as its input, so you need to wrap the output of the `temperature_random_sampling` function in square brackets [...]. Don't forget to use `self.vocab` as the second parameter for character mapping.
6. Finally, return the `predicted_chars`, which will be a single character, and the states tensor obtained from step 2. These components are essential for maintaining the sequence and generating subsequent characters.

In [ ]:

Edit Meta

```
# UNGRADED CLASS: GenerativeModel
class GenerativeModel(tf.keras.Model):
    def __init__(self, model, vocab, temperature=1.0):
        """
        A generative model for text generation.

        Args:
            model (tf.keras.Model): The underlying model for text generation.
            vocab (list): A list containing the vocabulary of unique characters.
            temperature (float, optional): A value to control the randomness of text generation. Defaults to 1.0.
        """
        super().__init__()
        self.temperature = temperature
        self.model = model
        self.vocab = vocab

    @tf.function
    def generate_one_step(self, inputs, states=None):
        """
        Generate a single character and update the model state.

        Args:
            inputs (string): The input string to start with.
            states (tf.Tensor): The state tensor.

        Returns:
            tf.Tensor, states: The predicted character and the current GRU state.
        """
        # Convert strings to token IDs.

        ### START CODE HERE ###

        # Transform the inputs into tensors
        input_ids = line_to_tensor(None, None)
        # Predict the sequence for the given input_ids. Use the states and return_state=True
        predicted_logits, states = self.model(input_ids, states, return_state=True)
        # Get only last element of the sequence
        predicted_logits = predicted_logits[0, -1, :]
        # Use the temperature_random_sampling to generate the next character.
        predicted_ids = temperature_random_sampling(predicted_logits, self.temperature)
        # Use the chars_from_ids to transform the code into the corresponding char
        predicted_chars = text_from_ids(predicted_ids, self.vocab)

        ### END CODE HERE ###

        # Return the characters and model state.
        return tf.expand_dims(predicted_chars, 0), states

    def generate_n_chars(self, num_chars, prefix):
        """
        Generate a text sequence of a specified length, starting with a given prefix.

        Args:
            num_chars (int): The length of the output sequence.
            prefix (string): The prefix of the sequence (also referred to as the seed).

        Returns:
            str: The generated text sequence.
        """
        states = None
        next_char = tf.constant([prefix])
        result = [next_char]
        for n in range(num_chars):
            next_char, states = self.generate_one_step(next_char, states)
            result.append(next_char)
```

In [ ]:

```
# UNIT TEST
# Fix the seed to get replicable results for testing
tf.random.set_seed(272)
gen = GenerativeModel(model, vocab, temperature=0.5)

print(gen.generate_n_chars(32, " "), '\n\n' + '_'*80)
print(gen.generate_n_chars(32, "Dear"), '\n\n' + '_'*80)
```

Edit Meta

Edit Meta

**Expected output**

hear he has a soldier.  
Here is a

---

Dear gold, if thou wilt endure the e

---

KING OF THE SHREW  
IV I beseech you,

---

In [ ]:

Edit Meta

Edit Meta

Now, generate a longer text. Let's check if it looks like Shakespeare fragment

In [ ]:

Edit Meta

```
tf.random.set_seed(np.random.randint(1, 1000))
gen = GenerativeModel(model, vocab, temperature=0.8)
import time
start = time.time()
print(gen.generate_n_chars(1000, "ROMEO "), '\n\n' + '_'*80)
```

Edit Meta

In the generated text above, you can see that the model generates text that makes sense capturing dependencies between words and without any input. A simple n-gram model would have not been able to capture all of that in one sentence.

Edit Meta

**On statistical methods**

Using a statistical method like the one you implemented in course 2 will not give you results that are as good as you saw here. Your model will not be able to encode information seen previously in the data set and as a result, the perplexity will increase. Remember from course 2 that the higher the perplexity, the worse your model is. Furthermore, statistical ngram models take up too much space and memory. As a result, they will be inefficient and too slow. Compared with deepnets, you can get a better perplexity. Note, learning about n-gram language models is still important and allows you to better understand deepnet