INF8225 - TP3

Team:

- Renaud Lespérance (1802867)
- Morgan Péju (2103232)

Link colab : https://colab.research.google.com/drive/1RVMUcgWjEiwhrRSTYz_dJudvjhaiBH_7?usp=sharing

Machine translation

The goal of this TP is to build a machine translation model. You will be comparing the performance of three different architectures:

- A vanilla RNN
- A GRU-RNN
- A transformer

You are provided with the code to load and build the pytorch dataset, and the code for the training loop. You "only" have to code the architectures. Of course, the use of built-in torch layers such as `nn.GRU`, `nn.RNN` or `nn.Transformer` is forbidden, as there would be no exercise otherwise.

The source sentences are in english and the target language is french.

This is also for you the occasion to see what a basic machine learning pipeline looks like. Take a look at the given code, you might learn a lot!

Do not forget to **select the runtime type as GPU!**

Sources

- Dataset: Tab-delimited Bilingual Sentence Pairs

- The code is inspired by this pytorch tutorial.

This notebook is quite big, use the table of contents to easily navigate through it.

Imports and data initializations

We first download and parse the dataset. From the parsed sentences we can build the vocabularies and the torch datasets. The end goal of this section is to have an iterator that can yield the pairs of translated datasets, and where each sentences is made of a sequence of tokens.

Imports

In [ ]:
```
!python3 -m spacy download en > /dev/null
!python3 -m spacy download fr > /dev/null
!pip install torchinfo > /dev/null
!pip install einops > /dev/null
!pip install wandb > /dev/null


from itertools import takewhile
from collections import Counter, defaultdict
import numpy as np
from sklearn.model_selection import train_test_split
import pandas as pd
```

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data.dataset import Dataset
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence

import torchtext
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator, Vocab
from torchtext.datasets import IWSLT2016

import einops
import wandb
from torchinfo import summary

# from nltk.translate.bleu_score import sentence_bleu
```

The tokenizers are objects that are able to divide a python string into a list of tokens (words, punctuations, special tokens...) as a list of strings.

The special tokens are used for a particular reasons:

- \<*unk*>: Replace an unknown word in the vocabulary by this default token
- \<*pad*>: Virtual token used to as padding token so a batch of sentences can have a unique length
- \<*bos*>: Token indicating the beggining of a sentence in the target sequence
- \<*eos*>: Token indicating the end of a sentence in the target sequence

In [ ]:
```
# Original dataset, but there's a bug on Colab with it
# train, valid, _ = IWSLT2016(language_pair=('fr', 'en'))
# train, valid = list(train), list(valid)

# Another dataset, but it is too huge
# !wget https://www.statmt.org/wmt14/training-monolingual-europarl-v7/europarl-v7.en.gz
# !wget https://www.statmt.org/wmt14/training-monolingual-europarl-v7/europarl-v7.fr.gz
# !gunzip europarl-v7.en.gz
# !gunzip europarl-v7.fr.gz

# with open('europarl-v7.en', 'r') as my_file:
#     english = my_file.readlines()

# with open('europarl-v7.fr', 'r') as my_file:
#     french = my_file.readlines()

# dataset = [
#     (en, fr)
#     for en, fr in zip(english, french)
# ]
# print(f'\n{len(dataset):,} sentences.')

# dataset, _ = train_test_split(dataset, test_size=0.8, random_state=0)  # Remove 80% of the dataset (it would be huge otherwise)
# train, valid = train_test_split(dataset, test_size=0.2, random_state=0)  # Split between train and validation dataset

# Our current dataset
!wget http://www.manythings.org/anki/fra-eng.zip
!unzip fra-eng.zip


df = pd.read_csv('fra.txt', sep='\t', names=['english', 'french', 'attribution'])
train = [(en, fr) for en, fr in zip(df['english'], df['french'])]
train, valid = train_test_split(train, test_size=0.1, random_state=0)
print(len(train))

en_tokenizer, fr_tokenizer = get_tokenizer('spacy', language='en'), get_tokenizer('spacy', language='fr')
SPECIALS = ['<unk>', '<pad>', '<bos>', '<eos>']
```

```
--2022-04-07 20:26:50--  http://www.manythings.org/anki/fra-eng.zip
```

```
Resolving www.manythings.org (www.manythings.org)... 172.67.186.54, 104.21.92.44, 2606:4700:3033::ac43:ba36, ...
Connecting to www.manythings.org (www.manythings.org)|172.67.186.54|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 6532197 (6.2M) [application/zip]
Saving to: 'fra-eng.zip'

fra-eng.zip         100%[===================>]   6.23M  8.65MB/s    in 0.7s

2022-04-07 20:26:52 (8.65 MB/s) - 'fra-eng.zip' saved [6532197/6532197]

Archive:  fra-eng.zip
  inflating: _about.txt
  inflating: fra.txt
173106
```

Datasets

Functions and classes to build the vocabularies and the torch datasets. The vocabulary is an object able to transform a string token into the id (an int) of that token in the vocabulary.

In [ ]:

```python
class TranslationDataset(Dataset):
    def __init__(
            self,
            dataset: list,
            en_vocab: Vocab,
            fr_vocab: Vocab,
            en_tokenizer,
            fr_tokenizer,
        ):
        super().__init__()

        self.dataset = dataset
        self.en_vocab = en_vocab
        self.fr_vocab = fr_vocab
        self.en_tokenizer = en_tokenizer
        self.fr_tokenizer = fr_tokenizer

    def __len__(self):
        """Return the number of examples in the dataset.
        """
        return len(self.dataset)

    def __getitem__(self, index: int) -> tuple:
        """Return a sample.

        Args
        ----
            index: Index of the sample.

        Output
        ------
            en_tokens: English tokens of the sample, as a LongTensor.
            fr_tokens: French tokens of the sample, as a LongTensor.
        """
        # Get the strings
        en_sentence, fr_sentence = self.dataset[index]

        # To list of words
        # We also add the beggining-of-sentence and end-of-sentence tokens
        en_tokens = ['<bos>'] + self.en_tokenizer(en_sentence) + ['<eos>']
        fr_tokens = ['<bos>'] + self.fr_tokenizer(fr_sentence) + ['<eos>']

        # To list of tokens
        en_tokens = self.en_vocab(en_tokens)  # list[int]
        fr_tokens = self.fr_vocab(fr_tokens)

        return torch.LongTensor(en_tokens), torch.LongTensor(fr_tokens)
```

```
def yield_tokens(dataset, tokenizer, lang):
    """Tokenize the whole dataset and yield the tokens.
    """
    assert lang in ('en', 'fr')
    sentence_idx = 0 if lang == 'en' else 1

    for sentences in dataset:
        sentence = sentences[sentence_idx]
        tokens = tokenizer(sentence)
        yield tokens


def build_vocab(dataset: list, en_tokenizer, fr_tokenizer, min_freq: int):
    """Return two vocabularies, one for each language.
    """
    en_vocab = build_vocab_from_iterator(
        yield_tokens(dataset, en_tokenizer, 'en'),
        min_freq=min_freq,
        specials=SPECIALS,
    )
    en_vocab.set_default_index(en_vocab['<unk>'])  # Default token for unknown words

    fr_vocab = build_vocab_from_iterator(
        yield_tokens(dataset, fr_tokenizer, 'fr'),
        min_freq=min_freq,
        specials=SPECIALS,
    )
    fr_vocab.set_default_index(fr_vocab['<unk>'])

    return en_vocab, fr_vocab


def preprocess(
        dataset: list,
        en_tokenizer,
        fr_tokenizer,
        max_words: int,
    ) -> list:
    """Preprocess the dataset.
    Remove samples where at least one of the sentences are too long.
    Those samples takes too much memory.
    Also remove the pending '\n' at the end of sentences.
    """
    filtered = []

    for en_s, fr_s in dataset:
        if len(en_tokenizer(en_s)) >= max_words or len(fr_tokenizer(fr_s)) >= max_words:
            continue

        en_s = en_s.replace('\n', '')
        fr_s = fr_s.replace('\n', '')

        filtered.append((en_s, fr_s))

    return filtered


def build_datasets(
        max_sequence_length: int,
        min_token_freq: int,
        en_tokenizer,
        fr_tokenizer,
        train: list,
        val: list,
    ) -> tuple:
    """Build the training, validation and testing datasets.
    It takes care of the vocabulary creation.
```

```
    Args
    ----
        - max_sequence_length: Maximum number of tokens in each sequences.
            Having big sequences increases dramatically the VRAM taken during training.
        - min_token_freq: Minimum number of occurences each token must have
            to be saved in the vocabulary. Reducing this number increases
            the vocabularies's size.
        - en_tokenizer: Tokenizer for the english sentences.
        - fr_tokenizer: Tokenizer for the french sentences.
        - train and val: List containing the pairs (english, french) sentences.


    Output
    ------
        - (train_dataset, val_dataset): Tuple of the two TranslationDataset objects.
    """
    datasets = [
        preprocess(samples, en_tokenizer, fr_tokenizer, max_sequence_length)
        for samples in [train, val]
    ]

    en_vocab, fr_vocab = build_vocab(datasets[0], en_tokenizer, fr_tokenizer, min_token_freq)

    datasets = [
        TranslationDataset(samples, en_vocab, fr_vocab, en_tokenizer, fr_tokenizer)
        for samples in datasets
    ]

    return datasets
```

In [ ]:
```
def generate_batch(data_batch: list, src_pad_idx: int, tgt_pad_idx: int) -> tuple:
    """Add padding to the given batch so that all
    the samples are of the same size.

    Args
    ----
        data_batch: List of samples.
            Each sample is a tuple of LongTensors of varying size.
        src_pad_idx: Source padding index value.
        tgt_pad_idx: Target padding index value.

    Output
    ------
        en_batch: Batch of tokens for the padded english sentences.
            Shape of [batch_size, max_en_len].
        fr_batch: Batch of tokens for the padded french sentences.
            Shape of [batch_size, max_fr_len].
    """
    en_batch, fr_batch = [], []
    for en_tokens, fr_tokens in data_batch:
        en_batch.append(en_tokens)
        fr_batch.append(fr_tokens)

    en_batch = pad_sequence(en_batch, padding_value=src_pad_idx, batch_first=True)
    fr_batch = pad_sequence(fr_batch, padding_value=tgt_pad_idx, batch_first=True)
    return en_batch, fr_batch
```

Models architecture

This is where you have to code the architectures.

In a machine translation task, the model takes as input the whole source sentence along with the current known tokens of the target, and predict the next token in the target sequence. This means that the target tokens are predicted in an autoregressive manner, starting from the first token (right after the \<*bos*> token) and producing tokens one by one until the last \<*eos*> token.

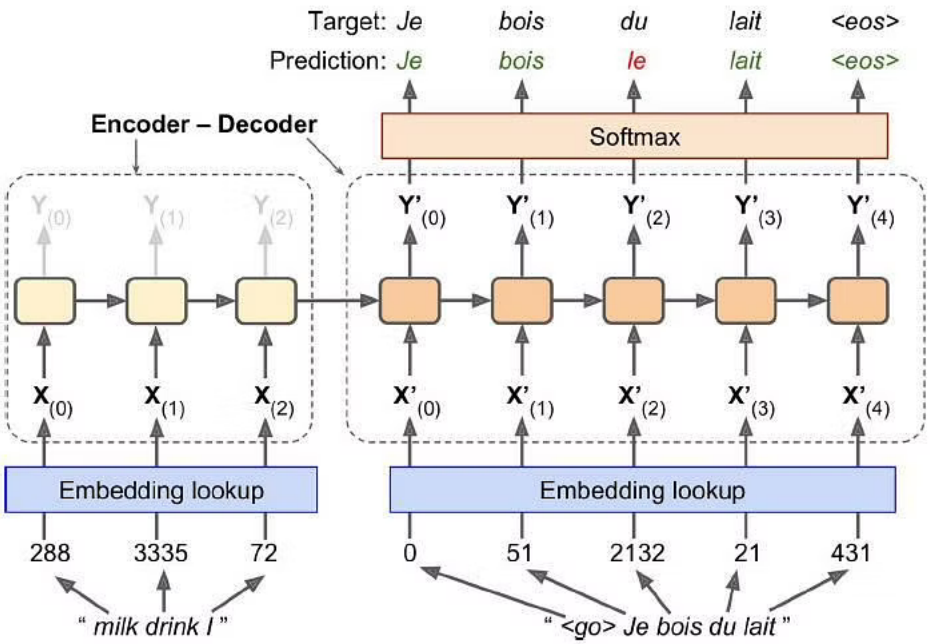
Formally, we define $s = [s_1, \ldots, s_{N_s}]$ as the source sequence made of N_s tokens. We also define $t^i = [t_1, \ldots, t_i]$ as the target sequence at the beginning of the step i.

The output of the model parameterized by θ is:

$$T_{i+1} = p(t_{i+1} | s, t^i; \theta)$$

Where T_{i+1} is the distribution of the next token t_{i+1}.

The loss is simply a *cross entropy loss* over the whole steps, where each class is a token of the vocabulary.



Note that in this image the english sentence is provided in reverse.

In pytorch, there is no dinstinction between an intermediate layer or a whole model having multiple layers in itself. Every layers or models inherit from the `torch.nn.Module`. This module needs to define the `__init__` method where you instanciate the layers, and the `forward` method where you decide how the inputs and the layers of the module interact between them. Thanks to the autograd computations of pytorch, you do not have to implement any backward method!

A really important advice is to **always look at the shape of your input and your output.** From that, you can often guess how the layers should interact with the inputs to produce the right output. You can also easily detect if there's something wrong going on.

You are more than advised to use the `einops` library and the `torch.einsum` function. This will require less operations than 'classical' code, but note that it's a bit trickier to use. This is a way of describing tensors manipulation with strings, bypassing the multiple tensor methods executed in the background. You can find a nice presentation of `einops` here. A paper has just been released about einops here.

A great tutorial on pytorch can be found here. Spending 3 hours on this tutorial is *no* waste of time.

RNN models

RNN and GRU

In [ ]:
```
class RNNCell(nn.Module):
    """A single RNN layer.
```

```
    Parameters
    ----------
        input_size: Size of each input token.
        hidden_size: Size of each RNN hidden state.
        dropout: Dropout rate.
    """
    def __init__(
            self,
            input_size: int,
            hidden_size: int,
            dropout: float,
        ):
        super().__init__()
        self.device = config['device']
        self.Wih = nn.Linear(input_size, hidden_size,device=self.device)
        self.Whh = nn.Linear(hidden_size, hidden_size,device=self.device)
        self.Dropout = nn.Dropout(dropout)

    def forward(self, x: torch.FloatTensor, h: torch.FloatTensor) -> tuple:
        """Go through all the sequence in x, iteratively updatating
        the hidden state h.

        Args
        ----
            x: Input sequence.
                Shape of [batch_size, seq_len, input_size].
            h: Initial hidden state.
                Shape of [batch_size, hidden_size].

        Output
        ------
            y: Token embeddings.
                Shape of [batch_size, seq_len, hidden_size].
            h: Last hidden state.
                Shape of [batch_size, hidden_size].
        """

        seq_len = x.shape[1]
        y_t = []

        for idx in range(seq_len):
          # RNN cell propagation
          h = torch.tanh(self.Wih(x[:,idx]) + self.Whh(h))

          # Add dropout on the outputs of each RNNCell except for the last one
          if idx != (seq_len-1):
            h = self.Dropout(h)

          # Keep the output for all indexes in the sequence.
          y_t.append(h)

        y = torch.stack(y_t,dim=1)
        return y,h

class GRUCell(nn.Module):
    """A single GRU layer.

    Parameters
    ----------
        input_size: Size of each input token.
        hidden_size: Size of each RNN hidden state.
        dropout: Dropout rate.
    """
    def __init__(
            self,
            input_size: int,
            hidden_size: int,
            dropout: float,
```

```
        ):
        super().__init__()
        self.device = config['device']

        # all in one nn.Linear, r , u , n  will be split after propagation
        self.Wih = nn.Linear(input_size, 3*hidden_size,device=self.device)
        self.Whh = nn.Linear(hidden_size,3*hidden_size,device=self.device)
        self.Dropout = nn.Dropout(dropout)

    def forward(self, x: torch.FloatTensor, h: torch.FloatTensor) -> tuple:
        """
        Args
        ----
            x: Input sequence.
                Shape of [batch_size, seq_len, input_size].
            h: Initial hidden state.
                Shape of [batch_size, hidden_size].

        Output
        ------
            y: Token embeddings.
                Shape of [batch_size, seq_len, hidden_size].
            h: Last hidden state.
                Shape of [batch_size, hidden_size].
        """
        seq_len = x.shape[1]
        y_t = []

        for idx in range(seq_len):
          ### GRU cell propagation

          # propagation of input
          x_input = self.Wih(x[:,idx])
          x_r,x_u,x_n = x_input.chunk(3,1) #Split r,u,n

          # propagation of hidden
          hid = self.Whh(h)
          h_r,h_u,h_n = hid.chunk(3,1) #Split r,u,n

          # Gate activation
          reset_g = torch.sigmoid(x_r+h_r)
          update_g = torch.sigmoid(x_u+h_u)
          new_g = torch.tanh(x_n + torch.mul(reset_g,h_n))

          # Output of the GRUCell
          h = torch.mul(new_g,(1-update_g)) + torch.mul(update_g, h)

          # Add dropout on the outputs of each GRUCell except for the last one
          if idx != (seq_len-1) :
            h = self.Dropout(h)

          # Keep the output for all indexes in the sequence.
          y_t.append(h)

        y = torch.stack(y_t,dim=1)
        return y,h


class RNN(nn.Module):
    """Implementation of an RNN based
    on https://pytorch.org/docs/stable/generated/torch.nn.RNN.html.

    Parameters
    ----------
        input_size: Size of each input token.
        hidden_size: Size of each RNN hidden state.
        num_layers: Number of layers (RNNCell or GRUCell).
```

```
        dropout: Dropout rate.
        model_type: Either 'RNN' or 'GRU', to select which model we want.
    """
    def __init__(
            self,
            input_size: int,
            hidden_size: int,
            num_layers: int,
            dropout: float,
            model_type: str,
        ):
        super().__init__()
        self.device = config['device']
        self.num_layers = num_layers
        self.hidden_size = hidden_size
        self._cells = nn.ModuleList()

        ### First layer of the RNN
        if model_type == 'RNN':
          self._cells.append(RNNCell(input_size,hidden_size,dropout))
        elif model_type == 'GRU':
          self._cells.append(GRUCell(input_size,hidden_size,dropout))
        else :
          print(f"model type {self.model_type} is not supported")

        ### Folowing layer if num_layers > 1
        for i in range(num_layers-1):
          if model_type == 'RNN':
            self._cells.append(RNNCell(hidden_size,hidden_size,dropout))
          elif model_type == 'GRU':
            self._cells.append(GRUCell(hidden_size,hidden_size,dropout))
          else :
            print(f"model type {self.model_type} is not supported")

    def forward(self, x: torch.FloatTensor, h: torch.FloatTensor=None) -> tuple:
        """Pass the input sequence through all the RNN cells.
        Returns the output and the final hidden state of each RNN layer

        Args
        ----
            x: Input sequence.
                Shape of [batch_size, seq_len, input_size].
            h: Hidden state for each RNN layer.
                Can be None, in which case an initial hidden state is created.
                Shape of [batch_size, n_layers, hidden_size].

        Output
        ------
            y: Output embeddings for each token after the RNN layers.
                Shape of [batch_size, seq_len, hidden_size].
            h: Final hidden state.
                Shape of [batch_size, n_layers, hidden_size].
        """
        # For the first layer, no input from a previous hidden layer is available
        if h is None :
          h = torch.zeros(x.shape[0],self.hidden_size,device=self.device)
        else:
          h = h[:,self.num_layers-1]

        # Propagate one cell at a time
        h_out=[]
        for layer,cell in enumerate(self._cells):
          if layer == 0 :
            y,h = cell(x,h)
          else:
            y,h = cell(y,h)
          h_out.append(h)
```

```
        h_out = torch.stack(h_out,dim=1)

        return y,h_out
```

Translation RNN

This module instanciates a vanilla RNN or a GRU-RNN and performs the translation task. You have to:

- Encode the source and target sequence
- Pass the final hidden state of the encoder to the decoder (one for each layer)
- Decode the hidden state into the target sequence

We use teacher forcing for training, meaning that when the next token is predicted, that prediction is based on the previous true target tokens.

In [ ]:
```
class TranslationRNN(nn.Module):
    """Basic RNN encoder and decoder for a translation task.
    It can run as a vanilla RNN or a GRU-RNN.

    Parameters
    ----------
        n_tokens_src: Number of tokens in the source vocabulary.
        n_tokens_tgt: Number of tokens in the target vocabulary.
        dim_embedding: Dimension size of the word embeddings (for both language).
        dim_hidden: Dimension size of the hidden layers in the RNNs
            (for both the encoder and the decoder).
        n_layers: Number of layers in the RNNs.
        dropout: Dropout rate.
        src_pad_idx: Source padding index value.
        tgt_pad_idx: Target padding index value.
        model_type: Either 'RNN' or 'GRU', to select which model we want.
        torch_fct: If true use torch RNN or GRU else our model
    """

    def __init__(
            self,
            n_tokens_src: int,
            n_tokens_tgt: int,
            dim_embedding: int,
            dim_hidden: int,
            n_layers: int,
            dropout: float,
            src_pad_idx: int,
            tgt_pad_idx: int,
            model_type: str,
            torch_fct_translation: bool,
        ):
        super().__init__()
        self.device = config['device']

        # Source and Target Embeddings
        self.embedding_encoder = nn.Embedding(n_tokens_src, dim_embedding, padding_idx=src_pad_idx)
        self.embedding_decoder = nn.Embedding(n_tokens_tgt, dim_embedding, padding_idx=tgt_pad_idx)

        if torch_fct_translation : # Use torch functions
          if model_type  == 'RNN' :
            self.model_encoder = nn.RNN(dim_embedding, dim_hidden, n_layers, dropout=dropout, batch_first=True)
            self.model_decoder = nn.RNN(dim_embedding, dim_hidden, n_layers, dropout=dropout, batch_first=True)
          elif model_type == 'GRU' :
            self.model_encoder = nn.GRU(dim_embedding, dim_hidden, n_layers, dropout=dropout, batch_first=True)
            self.model_decoder = nn.GRU(dim_embedding, dim_hidden, n_layers, dropout=dropout, batch_first=True)
          else :
            print(f"model type {self.model_type} is not supported")
        else : # Use our functions
          if model_type  == 'RNN' :
            self.model_encoder = RNN(dim_embedding, dim_hidden, n_layers, dropout,model_type='RNN')
```

```
            self.model_decoder = RNN(dim_embedding, dim_hidden, n_layers, dropout,model_type='RNN')
          elif model_type == 'GRU' :
            self.model_encoder = RNN(dim_embedding, dim_hidden, n_layers, dropout,model_type='GRU')
            self.model_decoder = RNN(dim_embedding, dim_hidden, n_layers, dropout,model_type='GRU')
          else :
            print(f"model type {self.model_type} is not supported")

        # Add normalization layer between encoder output and decoder input
        self.LNorm = nn.LayerNorm(dim_hidden,device=self.device)

        ### Uses an MLP for the translator output instead to increase accuracy.
        MLP_param = config['MLP_param_RNN_GRU']
        MLP_act = MLP_param[0] # Activation function to use
        if MLP_act == "LeakyReLU01":
          act = nn.LeakyReLU(0.1)
        if MLP_act == "ELU":
          act = nn.ELU()
        if MLP_act == "Mish":
          act = nn.Mish()

        MLP_layers = MLP_param[1] # Number of layers in our MLP
        MLP_scale = MLP_param[2]
        layers = []
        dropout_l = nn.Dropout(dropout)
        if MLP_layers == 1 :
          layers.append(nn.Linear(dim_hidden, n_tokens_tgt,device=self.device))
        else:
          #First layer
          layers.append(nn.Linear(dim_hidden, dim_hidden*MLP_scale,device=self.device))
          layers.append(dropout_l)
          layers.append(act)
          layers.append(nn.LayerNorm((dim_hidden*MLP_scale),device=self.device))

          for idx in range(MLP_layers-1) :
            if idx == MLP_layers-2 : #last hidden layer
              layers.append(nn.Linear(dim_hidden*MLP_scale, n_tokens_tgt,device=self.device))
            else:
              layers.append(nn.Linear(dim_hidden*MLP_scale, dim_hidden*MLP_scale,device=self.device))
              layers.append(dropout_l)
              layers.append(act)
              layers.append(nn.LayerNorm((dim_hidden*MLP_scale),device=self.device))

        #Create object _sequential to propagate MLP in one call
        self._sequential = nn.Sequential(*layers)

    def forward(
        self,
        source: torch.LongTensor,
        target: torch.LongTensor
    ) -> torch.FloatTensor:
        """Predict the source tokens based on the target tokens.

        Args
        ----
            source: Batch of source sentences.
                Shape of [batch_size, src_seq_len].
            target: Batch of target sentences.
                Shape of [batch_size, tgt_seq_len].

        Output
        ------
            y: Distributions over the next token for all tokens in each sentences.
                Those need to be the logits only, do not apply a softmax because
                it will be done in the loss computation for numerical stability.
                See https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html for more informations.
                Shape of [batch_size, tgt_seq_len, n_tokens_tgt].
        """
        ## ENCODER
```

```
        encoder_embedding = self.embedding_encoder(source)  # [batch_size, src_seq_len, embedding dim])

        ## Application of the RNN or GRU on the encoded source data.
        # If we initialize no "hidden layer", model_encoder automatically initializes the 1st hidden with zeros
        outputs, hidden = self.model_encoder(encoder_embedding) # output: [batch_size, src_seq_len, hidden dim] ;; hidden: [n_layers, batch_size, hidden_dim]
        hidden = self.LNorm(hidden)

        ## DECODER
        decoder_embedding = self.embedding_decoder(target) # [1, batch_size, embeddding dim]
        predict, hidden = self.model_decoder(decoder_embedding, hidden)

        ### returns the logits after a MLP
        return  self._sequential(predict)
```

Transformer model

Here you have to code the Transformer architecture. It is divided in three parts:

- Attention layers
- Encoder and decoder layers
- Main layers (gather the encoder and decoder layers)

The illustrated transformer blog can help you understanding how the architecture works. Once this is done, you can use the annontated transformer to have an idea of how to code this architecture. We encourage you to use `torch.einsum` and the `einops` library as much as you can. It will make your code simpler.

Implementation order

To help you with the implementation, we advise you following this order:

- Implement `TranslationTransformer` and use `nn.Transformer` instead of `Transformer`
- Implement `Transformer` and use `nn.TransformerDecoder` and `nn.TransformerEnocder`
- Implement the `TransformerDecoder` and `TransformerEncoder` and use `nn.MultiHeadAttention`
- Implement `MultiHeadAttention`

Do not forget to add `batch_first=True` when necessary in the `nn` modules.

Attention layers

We use a `MultiHeadAttention` module, that is able to perform self-attention aswell as cross-attention (depending on what you give as queries, keys and values).

Attention

It takes the multiheaded queries, keys and values as input. It computes the attention between the queries and the keys and return the attended values.

The implementation of this function can greatly be improved with *einsums*.

MultiheadAttention

Computes the multihead queries, keys and values and feed them to the `attention` function. You also need to merge the key padding mask and the attention mask into one mask.

The implementation of this module can greatly be improved with *einops.rearrange*.

In [ ]:
```
from einops.layers.torch import Rearrange
from torch._C import device

def attention(
        q: torch.FloatTensor,
        k: torch.FloatTensor,
        v: torch.FloatTensor,
```

```
        mask: torch.BoolTensor=None,
        dropout: nn.Dropout=None,
    ) -> tuple:
    """Computes multihead scaled dot-product attention from the
    projected queries, keys and values.

    Args
    ----
        q: Batch of queries.
            Shape of [batch_size, seq_len_1, n_heads, dim_model].
        k: Batch of keys.
            Shape of [batch_size, seq_len_2, n_heads, dim_model].
        v: Batch of values.
            Shape of [batch_size, seq_len_2, n_heads, dim_model].
        mask: Prevent tokens to attend to some other tokens (for padding or autoregressive attention).
            Attention is prevented where the mask is `True`.
            Shape of [batch_size, n_heads, seq_len_1, seq_len_2],
            or broadcastable to that shape.
        dropout: Dropout layer to use.

    Output
    ------
        y: Multihead scaled dot-attention between the queries, keys and values.
            Shape of [batch_size, seq_len_1, n_heads, dim_model].
        attn: Computed attention mask.
            Shape of [batch_size, n_heads, seq_len_1, seq_len_2].
    """
    # TODO
    # Définition des variables
    batch_size = q.shape[0]
    k_length = k.shape[1]
    d_model = q.shape[3]
    n_heads = q.shape[2]
    d_k = d_model


    # Permutations
    q = q.permute(0,2,1,3)
    k = k.permute(0,2,1,3)
    v = v.permute(0,2,1,3)
    # Scaling pour éviter la saturation de softmax
    scaled_attention = q / np.sqrt(d_k)  # [batch_size, n_heads, seq_len_1, dim_per_head]

    # Calcul des scores qui déterminent l'importance à accorder aux autres parties de la phrase d'entrée lorsqu'on encode un mot à une certaine position
    scores = torch.matmul(scaled_attention, k.transpose(2,3))   # [batch_size, n_heads, seq_len_1, seq_len_2]

    # Application du mask
    if mask is not None:
      scores = scores.masked_fill_(mask==1,float("-inf"))

    # Application de Softmax pour normaliser les scores : déterminer le score pour lequel chaque mot sera exprimé à cette position
    attn = nn.Softmax(dim=-1)(scores)   # [batch_size, n_heads, seq_len_1, seq_len_2]
    # Application d'un dropout
    scores = dropout(scores) if dropout is not None else scores

    # Multiplier "value" par les scores pour obtenir les valeurs des mots sur lesquels on veut se concentrer et réduire les mots moins pertinents
    # Et les sommer pour obtenir la sortie de la couche d'attention
    y = torch.matmul(attn, v)

    y = y.permute(0,2,1,3) # Permutation pour avoir le format souhaité [batch_size, seq_len_1, n_heads, dim_model]
    '''
    scores = torch.einsum("blhk,bthk->bhlt",[q,k])
    if mask is not None:
      scores = scores.masked_fill_(mask==1, float('-inf'))

    attn = torch.softmax(scores/np.sqrt(k.shape[0]), dim=3)
    y = torch.einsum("bhlt,bthk->blhk", [dropout(attn),v])
    '''
```

```
    return y, attn

class MultiheadAttention(nn.Module):
    """Multihead attention module.
    Can be used as a self-attention and cross-attention layer.
    The queries, keys and values are projected into multiple heads
    before computing the attention between those tensors.

    Parameters
    ----------
        dim: Dimension of the input tokens.
        n_heads: Number of heads. `dim` must be divisible by `n_heads`.
        dropout: Dropout rate.
    """
    def __init__(
            self,
            dim: int,
            n_heads: int,
            dropout: float,
        ):
        super().__init__()

        assert dim % n_heads == 0 #"Embedding dimension must be 0 modulo number of heads."

        # TODO
        # Définition des variables
        self.device = config['device']
        self.dim = dim
        self.n_heads = n_heads
        # Définition des projections pour "query","key" et "value"
        self.Wq = nn.Linear(dim, dim,device=self.device)
        self.Wk = nn.Linear(dim, dim,device=self.device)
        self.Wv = nn.Linear(dim, dim,device=self.device)
        # Définition de la couche "fully connected"
        self.fc = nn.Linear(dim, dim,device=self.device)
        # Définition du dropout
        self.dropout = nn.Dropout(dropout)

    def forward(
            self,
            q: torch.FloatTensor,
            k: torch.FloatTensor,
            v: torch.FloatTensor,
            key_padding_mask: torch.BoolTensor = None,
            attn_mask: torch.BoolTensor = None,
        ) -> torch.FloatTensor:
        """Computes the scaled multi-head attention form the input queries,
        keys and values.

        Project those queries, keys and values before feeding them
        to the `attention` function.

        The masks are boolean masks. Tokens are prevented to attends to
        positions where the mask is `True`.

        Args
        ----
            q: Batch of queries.
                Shape of [batch_size, seq_len_1, dim_model].
            k: Batch of keys.
                Shape of [batch_size, seq_len_2, dim_model].
            v: Batch of values.
                Shape of [batch_size, seq_len_2, dim_model].
            key_padding_mask: Prevent attending to padding tokens.
                Shape of [batch_size, seq_len_2].
            attn_mask: Prevent attending to subsequent tokens.
                Shape of [seq_len_1, seq_len_2].
```

```
        Output
        ------
            y: Computed multihead attention.
                Shape of [batch_size, seq_len_1, dim_model].
        """
        # TODO
        batch_size = q.shape[0]
        seq_len_1, seq_len_2 = q.shape[1], k.shape[1]

        # Projections de "query","key" et "value"
        q = self.Wq(q)  # [batch_size, seq_len1, d_model]
        k = self.Wk(k)  # [batch_size, seq_len2, d_model]
        v = self.Wv(v)  # [batch_size, seq_len2, d_model]

        # Reshape
        q = q.view(batch_size, -1, self.n_heads, self.dim//self.n_heads) # [batch_size, n_heads, seq_len_1, depth]
        k = k.view(batch_size, -1, self.n_heads, self.dim//self.n_heads)  # [batch_size, n_heads, seq_len_2, depth]
        v = v.view(batch_size, -1, self.n_heads, self.dim//self.n_heads) # [batch_size, n_heads, seq_len_2, depth]

        # Gestion des masks key padding et attention masks

        if key_padding_mask is not None:
            # Reshape du mask de padding [batch_size, 1, 1, seq_len_2]
            key_padding_mask = key_padding_mask.view(batch_size, 1, 1, seq_len_2)
            if attn_mask is not None: # Si on a mask d'attention et de padding
              attn_mask = attn_mask.view(1, 1, seq_len_1, seq_len_2)
              mask = torch.logical_or(attn_mask>0,key_padding_mask)
            else: # Si on a uniquement mask de padding
              mask = key_padding_mask


        # Appel à la fonction "scaled dot product" pour le calcul de l'attention
        # scaled_attention shape [batch_size, seq_len_1, n_heads, depth]
        # attention_weights shape [batch_size, n_heads, seq_len_1, seq_len_2]
        scaled_attention, attention_weights = attention(q, k, v, mask=mask)
        # Reshape de l'attention : [batch_size, seq_len_1, d_model]
        scaled_attention = scaled_attention.contiguous().view(batch_size, -1, self.n_heads * (self.dim // self.n_heads))

        # Application d'une couche "fully connected"
        y = self.fc(scaled_attention) # y shape [batch_size, seq_len_1, dim_model]

        return y
```

Encoder and decoder layers

TranformerEncoder

Apply self-attention layers onto the source tokens. It only needs the source key padding mask.

TranformerDecoder

Apply masked self-attention layers to the target tokens and cross-attention layers between the source and the target tokens. It needs the source and target key padding masks, and the target attention mask.

In [ ]:
```
from torch.nn.modules.container import ModuleList
from torch._C import device

class TransformerDecoderLayer(nn.Module):
    """Single decoder layer.

    Parameters
    ----------
        d_model: The dimension of decoders inputs/outputs.
        dim_feedforward: Hidden dimension of the feedforward networks.
        nheads: Number of heads for each multi-head attention.
        dropout: Dropout rate.
    """
```

```
    def __init__(
            self,
            d_model: int,
            d_ff: int,
            nhead: int,
            dropout: float,
            torch_fct_transformer: bool,
        ):
        super().__init__()

        # TODO
        # Définition des variables
        self.device = config['device']
        self.torch_fct_transformer = torch_fct_transformer

        # Définition de l'appel à la couche d'attention
        if torch_fct_transformer[3]== True: # Utilisation de Pytorch pour MultiheadAttention
          self.selfAttention = nn.MultiheadAttention(d_model, nhead, dropout=dropout, batch_first = True,device=self.device)
          self.multiheadAttention = nn.MultiheadAttention(d_model, nhead, dropout=dropout, batch_first = True,device=self.device)
        elif torch_fct_transformer[3]== False: # Utilisation du MultiheadAttention "Homemade"
          self.selfAttention = MultiheadAttention(d_model, nhead, dropout)
          self.multiheadAttention = MultiheadAttention(d_model, nhead, dropout)

        # Définition des couches du décodeur
        self.feed_forward = nn.Sequential(
            nn.Linear(d_model, d_ff,device=self.device),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model,device=self.device)
        )
        # Définition de couches de normalisation
        self.layer_norm1 = nn.LayerNorm(d_model,device=self.device)
        self.layer_norm2 = nn.LayerNorm(d_model,device=self.device)
        self.layer_norm3 = nn.LayerNorm(d_model,device=self.device)
        # Définition des dropouts
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)
        self.dropout3 = nn.Dropout(dropout)

    def forward(
            self,
            tgt: torch.FloatTensor,
            src: torch.FloatTensor,
            tgt_mask_attn: torch.BoolTensor,
            src_key_padding_mask: torch.BoolTensor,
            tgt_key_padding_mask: torch.BoolTensor,
        ) -> torch.FloatTensor:
        """Decode the next target tokens based on the previous tokens.

        Args
        ----
            tgt: Batch of target sentences.
                Shape of [batch_size, tgt_seq_len, dim_model].
            src: Batch of source sentences.
                Shape of [batch_size, src_seq_len, dim_model].
            tgt_mask_attn: Mask to prevent attention to subsequent tokens.
                Shape of [tgt_seq_len, tgt_seq_len].
            src_key_padding_mask: Mask to prevent attention to padding in src sequence.
                Shape of [batch_size, src_seq_len].
            tgt_key_padding_mask: Mask to prevent attention to padding in tgt sequence.
                Shape of [batch_size, tgt_seq_len].

        Output
        ------
            y:  Batch of sequence of embeddings representing the predicted target tokens
                Shape of [batch_size, tgt_seq_len, dim_model].
        """
```

```
        # TODO
        y = tgt
        # Appel à la couche de "self attention" (Pytorch ou "Homemade")
        if self.torch_fct_transformer[3]== True : # Utilisation de Pytorch pour MultiheadAttention
          attn1, _ = self.selfAttention(y, y, y, attn_mask=tgt_mask_attn, key_padding_mask=tgt_key_padding_mask)  # (batch_size, target_seq_len, d_model)
          attn1 = self.dropout1(attn1)
        elif self.torch_fct_transformer[3]== False: # Utilisation du MultiheadAttention "Homemade"
          attn1 = self.dropout1(self.selfAttention(y, y, y, attn_mask=tgt_mask_attn, key_padding_mask=tgt_key_padding_mask))  # (batch_size, target_seq_len, d_model)

        # Application d'une normalisation
        y = self.layer_norm1(y + attn1)

        # Appel à la couche d'attention (Pytorch ou "Homemade")
        if self.torch_fct_transformer[3]== True: # Utilisation de Pytorch pour MultiheadAttention
          attn2, _ = self.multiheadAttention(y, src, src, key_padding_mask=src_key_padding_mask)  # (batch_size, target_seq_len, d_model)
          attn2 = self.dropout1(attn2)
        elif self.torch_fct_transformer[3]== False: #Utilisation du MultiheadAttention "Homemade"
          attn2 = self.dropout2(self.multiheadAttention(y, src, src, key_padding_mask=src_key_padding_mask))  # (batch_size, target_seq_len, d_model)
        # Application d'une normalisation
        y = self.layer_norm2(y + attn2)

        # Forward dans les couches du décodeur
        ffn_output = self.dropout3(self.feed_forward(y))
        # Application d'une normalisation
        y = self.layer_norm3(y + ffn_output)
        return y

class TransformerDecoder(nn.Module):
    """Implementation of the transformer decoder stack.

    Parameters
    ----------
        d_model: The dimension of decoders inputs/outputs.
        dim_feedforward: Hidden dimension of the feedforward networks.
        num_decoder_layers: Number of stacked decoders.
        nheads: Number of heads for each multi-head attention.
        dropout: Dropout rate.
    """

    def __init__(
            self,
            d_model: int,
            d_ff: int,
            num_decoder_layer:int ,
            nhead: int,
            dropout: float,
            torch_fct_transformer: bool,
        ):
        super().__init__()

        # TODO
        # Définition des variables
        self.device = config['device']
        self.d_model = d_model
        self.num_decoder_layer = num_decoder_layer
        # Définition des couches du décodeur
        self.dec_layers = [TransformerDecoderLayer(d_model, d_ff, nhead, dropout, torch_fct_transformer)
                           for _ in range(num_decoder_layer)]

        self.layer_norm = nn.LayerNorm(d_model, device=self.device)
    def forward(
            self,
            tgt: torch.FloatTensor,
            src: torch.FloatTensor,
            tgt_mask_attn: torch.BoolTensor,
            src_key_padding_mask: torch.BoolTensor,
            tgt_key_padding_mask: torch.BoolTensor,
        ) -> torch.FloatTensor:
```

```
        """Decodes the source sequence by sequentially passing.
        the encoded source sequence and the target sequence through the decoder stack.

        Args
        ----
            tgt: Batch of taget sentences.
                Shape of [batch_size, tgt_seq_len, dim_model].
            src: Batch of encoded source sentences.
                Shape of [batch_size, src_seq_len, dim_model].
            tgt_mask_attn: Mask to prevent attention to subsequent tokens.
                Shape of [tgt_seq_len, tgt_seq_len].
            src_key_padding_mask: Mask to prevent attention to padding in src sequence.
                Shape of [batch_size, src_seq_len].
            tgt_key_padding_mask: Mask to prevent attention to padding in tgt sequence.
                Shape of [batch_size, tgt_seq_len].

        Output
        ------
            y:  Batch of sequence of embeddings representing the predicted target tokens
                Shape of [batch_size, tgt_seq_len, dim_model].
        """
        # TODO
        y = tgt
        # Appel aux couches du décodeur
        for i in range(self.num_decoder_layer):
          y = self.dec_layers[i](y, src, tgt_mask_attn, src_key_padding_mask,tgt_key_padding_mask)

        y = self.layer_norm(y)

        return y # shape [batch_size, tgt_seq_len, d_model]


class TransformerEncoderLayer(nn.Module):
    """Single encoder layer.

    Parameters
    ----------
        d_model: The dimension of input tokens.
        dim_feedforward: Hidden dimension of the feedforward networks.
        nheads: Number of heads for each multi-head attention.
        dropout: Dropout rate.
    """

    def __init__(
            self,
            d_model: int,
            d_ff: int,
            nhead: int,
            dropout: float,
            torch_fct_transformer: bool,
        ):
        super().__init__()

        # TODO
        # Définition des variables
        self.device = config['device']
        self.torch_fct_transformer = torch_fct_transformer

        # Définition de la couche d'attention (Pytorch ou "Homemade")
        if torch_fct_transformer[3]== True: # Utilisation de Pytorch pour MultiheadAttention
          self.multiheadAttention = nn.MultiheadAttention(d_model, nhead, dropout=dropout, batch_first = True,device=self.device)
        elif torch_fct_transformer[3]== False: # Utilisation du MultiheadAttention "homemade"
          self.multiheadAttention = MultiheadAttention(d_model, nhead, dropout)
        # Définition des couches de l'encodeur
        self.feed_forward = nn.Sequential(
            nn.Linear(d_model, d_ff,device=self.device),
            nn.ReLU(),
            nn.Dropout(dropout),
```

```
            nn.Linear(d_ff, d_model,device=self.device)
        )

        # Définition de la normalisation des couches
        self.layer_norm1 = nn.LayerNorm(d_model,device=self.device)
        self.layer_norm2 = nn.LayerNorm(d_model,device=self.device)
        # Définitions des dropouts
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(
        self,
        src: torch.FloatTensor,
        key_padding_mask: torch.BoolTensor
        ) -> torch.FloatTensor:
        """Encodes the input. Does not attend to masked inputs.

        Args
        ----
            src: Batch of embedded source tokens.
                Shape of [batch_size, src_seq_len, dim_model].
            key_padding_mask: Mask preventing attention to padding tokens.
                Shape of [batch_size, src_seq_len].

        Output
        ------
            y: Batch of encoded source tokens.
                Shape of [batch_size, src_seq_len, dim_model].
        """
        # TODO
        # Appel à la couche d'attention (Pytorch ou "Homemade")
        if self.torch_fct_transformer[3]== True: # Utilisation de Pytorch pour MultiheadAttention
          attn_output, _ = self.multiheadAttention(src, src, src, key_padding_mask=key_padding_mask)  # [batch_size, src_seq_len, d_model]
        elif self.torch_fct_transformer[3]== False: # Utilisation du MultiheadAttention "Homemade"
          attn_output = self.multiheadAttention(src, src, src, key_padding_mask=key_padding_mask)  # [batch_size, src_seq_len, d_model]

        # Application d'un dropout
        attn_output = self.dropout1(attn_output)
        # Application de la couche de normalisation
        out1 = self.layer_norm1(src + attn_output)  # [batch_size, src_seq_len, d_model]

        # Forward dans les couches de l'encodeur
        ffn_output = self.feed_forward(out1)  # [batch_size, src_seq_len, d_model]
        # Application d'un dropout
        ffn_output = self.dropout2(ffn_output)
        # Application de la couche de normalisation pour obtenir le batch de token encodés de sortie de couche d'encodeur
        y = self.layer_norm2(out1 + ffn_output)  # [batch_size, src_seq_len, d_model]

        return y


class TransformerEncoder(nn.Module):
    """Implementation of the transformer encoder stack.

    Parameters
    ----------
        d_model: The dimension of encoders inputs.
        dim_feedforward: Hidden dimension of the feedforward networks.
        num_encoder_layers: Number of stacked encoders.
        nheads: Number of heads for each multi-head attention.
        dropout: Dropout rate.
    """

    def __init__(
            self,
            d_model: int,
            dim_feedforward: int,
            num_encoder_layers: int,
```

```
            nheads: int,
            dropout: float,
            torch_fct_transformer: bool,
        ):
        super().__init__()

        # TODO
        # Définition des variables
        self.device = config['device']
        self.d_model = d_model
        self.num_encoder_layers = num_encoder_layers

        # Définition des couches de l'encodeur
        self.enc_layers = [TransformerEncoderLayer(d_model,dim_feedforward,nheads, dropout, torch_fct_transformer)
                           for _ in range(num_encoder_layers)]
        self.layer_norm = nn.LayerNorm(d_model, device=self.device)
    def forward(
            self,
            src: torch.FloatTensor,
            key_padding_mask: torch.BoolTensor
        ) -> torch.FloatTensor:
        """Encodes the source sequence by sequentially passing.
        the source sequence through the encoder stack.

        Args
        ----
            src: Batch of embedded source sentences.
                Shape of [batch_size, src_seq_len, dim_model].
            key_padding_mask: Mask preventing attention to padding tokens.
                Shape of [batch_size, src_seq_len].

        Output
        ------
            y: Batch of encoded source sequence.
                Shape of [batch_size, src_seq_len, dim_model].
        """
        # TODO
        y = src
        # Appel aux couches de l'encodeur
        for i in range(self.num_encoder_layers) :
          y = self.enc_layers[i](y, key_padding_mask)

        y = self.layer_norm(y)
        return y  # [batch_size, src_seq_len, d_model]
```

Main layers

This section gather the `Transformer` and the `TranslationTransformer` modules.

Transformer

The classical transformer architecture. It takes the source and target tokens embeddings and do the forward pass through the encoder and decoder.

Translation Transformer

Compute the source and target tokens embeddings, and apply a final head to produce next token logits. The output must not be the softmax but just the logits, because we use the `nn.CrossEntropyLoss`.

It also creates the *src_key_padding_mask*, the *tgt_key_padding_mask* and the *tgt_mask_attn*.

In [ ]:
```
## EXPERIMENT - Positional Encoding

class PositionalEncoding_Experiment(nn.Module):
    """
    Compute positional embedding with sinusoid
```

```
    Inspired by : https://github.com/hyunwoongko/transformer/blob/master/README.md
    """
    def __init__(self, d_model, max_len, device):
        """
        d_model: dimension of model
        max_len: max sequence length
        device: device setting
        """
        super(PositionalEncoding_Experiment, self).__init__()

        # Initialization of the positional embedding
        self.encoding = torch.zeros(max_len, d_model, device=device)
        self.encoding.requires_grad = False

        pos = torch.arange(0, max_len, device=device)
        pos = pos.float().unsqueeze(dim=1)

        _2i = torch.arange(0, d_model, step=2, device=device).float()

        # Compute positional embeddings
        self.encoding[:, 0::2] = torch.sin(pos / (10000 ** (_2i / d_model)))
        self.encoding[:, 1::2] = torch.cos(pos / (10000 ** (_2i / d_model)))


    def forward(self, x):
        batch_size, seq_len = x.size()

        return self.encoding[:seq_len, :]
```

In [ ]:

```
from torch._C import device
from IPython.lib.display import YouTubeVideo
class Transformer(nn.Module):
    """Implementation of a Transformer based on the paper: https://arxiv.org/pdf/1706.03762.pdf.

    Parameters
    ----------
        d_model: The dimension of encoders/decoders inputs/ouputs.
        nhead: Number of heads for each multi-head attention.
        num_encoder_layers: Number of stacked encoders.
        num_decoder_layers: Number of stacked encoders.
        dim_feedforward: Hidden dimension of the feedforward networks.
        dropout: Dropout rate.
    """

    def __init__(
            self,
            d_model: int,
            nhead: int,
            num_encoder_layers: int,
            num_decoder_layers: int,
            dim_feedforward: int,
            dropout: float,
            torch_fct_transformer: bool,
        ):
        super().__init__()
        # TODO
        # Définition des variables
        self.device = config['device']
        self.torch_fct_transformer = torch_fct_transformer

        # Définition de l'encodeur (Pytorch ou "Homemade")
        if torch_fct_transformer[1] == True: # Utilisation de Pytorch pour l'encodeur
          encoder_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=nhead, dim_feedforward=dim_feedforward, dropout=dropout, batch_first=True,device=self.device)
          self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_encoder_layers)
        elif torch_fct_transformer[1] == False: # Utilisation de l'encodeur "homemade"
          self.transformer_encoder = TransformerEncoder(d_model, dim_feedforward=dim_feedforward, num_encoder_layers=num_encoder_layers, nheads=nhead, dropout=dropout, torch_fct_transformer=
```

```
        # Définition du décodeur (Pytorch ou "Homemade")
        if torch_fct_transformer[2] == True: # Utilisation de Pytorch pour le decodeur
          decoder_layer = nn.TransformerDecoderLayer(d_model=d_model, nhead=nhead, dim_feedforward=dim_feedforward, dropout=dropout, batch_first=True,device=self.device)
          self.transformer_decoder = nn.TransformerDecoder(decoder_layer, num_layers=num_decoder_layers)
        elif torch_fct_transformer[2] == False: # Utilisation du décodeur "homemade"
          self.transformer_decoder = TransformerDecoder(d_model, d_ff=dim_feedforward, num_decoder_layer=num_decoder_layers, nhead=nhead, dropout=dropout, torch_fct_transformer=torch_fct_trar

    def forward(
            self,
            src: torch.FloatTensor,
            tgt: torch.FloatTensor,
            tgt_mask_attn: torch.BoolTensor,
            src_key_padding_mask: torch.BoolTensor,
            tgt_key_padding_mask: torch.BoolTensor
        ) -> torch.FloatTensor:
        """Compute next token embeddings.

        Args
        ----
            src: Batch of source sequences.
                Shape of [batch_size, src_seq_len, dim_model].
            tgt: Batch of target sequences.
                Shape of [batch_size, tgt_seq_len, dim_model].
            tgt_mask_attn: Mask to prevent attention to subsequent tokens.
                Shape of [tgt_seq_len, tgt_seq_len].
            src_key_padding_mask: Mask to prevent attention to padding in src sequence.
                Shape of [batch_size, src_seq_len].
            tgt_key_padding_mask: Mask to prevent attention to padding in tgt sequence.
                Shape of [batch_size, tgt_seq_len].

        Output
        ------
            y: Next token embeddings, given the previous target tokens and the source tokens.
                Shape of [batch_size, tgt_seq_len, dim_model].
        """
        # TODO
        # Appel à l'encodeur (Pytorch ou "Homemade")
        if self.torch_fct_transformer[1] == True: # Utilisation de l'encoder de Pytorch
          memory = self.transformer_encoder(src, src_key_padding_mask=src_key_padding_mask)
        else: # Utilisation de l'encoder "homemade"
          memory = self.transformer_encoder(src, key_padding_mask=src_key_padding_mask)

        # Appel au decodeur (Pytorch ou "Homemade")
        if self.torch_fct_transformer[2] == True: # Utilisation du décodeur de Pytorch
          y = self.transformer_decoder(tgt, memory, tgt_mask=tgt_mask_attn, tgt_key_padding_mask=tgt_key_padding_mask, memory_key_padding_mask=src_key_padding_mask)
        else: # Utilisation du décodeur "homemade"
          y = self.transformer_decoder(tgt, memory, tgt_mask_attn, src_key_padding_mask, tgt_key_padding_mask)

        return y # y shape [batch_size, tgt_seq_len, dim_model]


class TranslationTransformer(nn.Module):
    """Basic Transformer encoder and decoder for a translation task.
    Manage the masks creation, and the token embeddings.
    Position embeddings can be learnt with a standard `nn.Embedding` layer.

    Parameters
    ----------
        n_tokens_src: Number of tokens in the source vocabulary.
        n_tokens_tgt: Number of tokens in the target vocabulary.
        n_heads: Number of heads for each multi-head attention.
        dim_embedding: Dimension size of the word embeddings (for both language).
        dim_hidden: Dimension size of the feedforward layers
            (for both the encoder and the decoder).
        n_layers: Number of layers in the encoder and decoder.
        dropout: Dropout rate.
        src_pad_idx: Source padding index value.
        tgt_pad_idx: Target padding index value.
```

```
    """
    def __init__(
            self,
            n_tokens_src: int,
            n_tokens_tgt: int,
            n_heads: int,
            dim_embedding: int,
            dim_hidden: int,
            n_layers: int,
            dropout: float,
            src_pad_idx: int,
            tgt_pad_idx: int,
            torch_fct_transformer: bool,
            positional_embeddings_exp: bool
        ):
        super().__init__()

        # TODO
        # Définition des variables
        self.device = config['device']
        self.dim_embedding = dim_embedding
        self.n_heads = n_heads
        self.n_layers = n_layers
        self.tgt_pad_idx = tgt_pad_idx
        self.src_pad_idx = src_pad_idx
        self.torch_fct_transformer = torch_fct_transformer

        # Définition des embeddings
        self.embedding_src = nn.Embedding(n_tokens_src, dim_embedding, padding_idx=src_pad_idx,device=self.device)
        self.embedding_tgt = nn.Embedding(n_tokens_tgt, dim_embedding, padding_idx=tgt_pad_idx,device=self.device)
        self.embedding_pos_src = nn.Embedding(n_tokens_src, dim_embedding, padding_idx=src_pad_idx,device=self.device)
        self.embedding_pos_tgt = nn.Embedding(n_tokens_tgt, dim_embedding, padding_idx=tgt_pad_idx,device=self.device)
        self.dropout_enc = nn.Dropout(dropout)
        self.dropout_dec = nn.Dropout(dropout)
        # Définition de la couche fully connected
        self.fc_linear = nn.Linear(dim_embedding, n_tokens_tgt,device=self.device)
        # Définition du transformer (Pytorch ou "homemade")
        if torch_fct_transformer[0]:
          self.transformer_model = nn.Transformer(d_model=dim_embedding,nhead=n_heads, num_encoder_layers=n_layers,num_decoder_layers=n_layers, dim_feedforward=dim_hidden, dropout=dropout, ba
        else:
          self.transformer_model = Transformer(d_model=dim_embedding, nhead=n_heads, num_encoder_layers=n_layers,num_decoder_layers=n_layers, dim_feedforward=dim_hidden, dropout=dropout, torc

        ### Uses an MLP for the translator output instead to increase accuracy.
        MLP_param = config['MLP_param_transformer']
        MLP_act = MLP_param[0] # Activation function to use
        if MLP_act == "LeakyReLU01":
          act = nn.LeakyReLU(0.1)
        if MLP_act == "ELU":
          act = nn.ELU()
        if MLP_act == "Mish":
          act = nn.Mish()

        MLP_layers = MLP_param[1] # Number of layers in our MLP
        MLP_scale = MLP_param[2]
        layers = []
        dropout_l = nn.Dropout(dropout)

        if MLP_layers == 1 :
          layers.append(nn.Linear(dim_embedding, n_tokens_tgt,device=self.device))
        else:
          #First layer
          layers.append(nn.Linear(dim_embedding, dim_embedding*MLP_scale,device=self.device))
          layers.append(dropout_l)
          layers.append(act)
          layers.append(nn.LayerNorm((dim_embedding*MLP_scale),device=self.device))

          for idx in range(MLP_layers-1) :
            if idx == MLP_layers-2 : #last hidden layer
```

```
                layers.append(nn.Linear(dim_embedding*MLP_scale, n_tokens_tgt,device=self.device))
              else:
                layers.append(nn.Linear(dim_embedding*MLP_scale, dim_embedding*MLP_scale,device=self.device))
                layers.append(dropout_l)
                layers.append(act)
                layers.append(nn.LayerNorm((dim_embedding*MLP_scale),device=self.device))

          #Create object _sequential to propagate MLP in one call
          self._sequential = nn.Sequential(*layers)

          # PARTIE "EXPERIMENT"
          self.positional_embeddings_exp = positional_embeddings_exp

      def forward(
              self,
              source: torch.LongTensor,
              target: torch.LongTensor
          ) -> torch.FloatTensor:
          """Predict the target tokens based on the source tokens.

          Args
          ----
              source: Batch of source sentences.
                  Shape of [batch_size, seq_len_src].
              target: Batch of target sentences.
                  Shape of [batch_size, seq_len_tgt].

          Output
          ------
              y: Batch of predictions of the next token distributions in the target sentences.
                  Shape of [batch_size, seq_len_tgt, n_tokens_tgt].
          """
          # TODO
          batch_size = source.shape[0]
          seq_len_src, seq_len_tgt = source.shape[1], target.shape[1]

          # Embedding & Positional Embedding
          src_embedding = self.embedding_src(source)  # [batch, seq_len_src, dim_embedding]

          tgt_embedding = self.embedding_tgt(target)

          if self.positional_embeddings_exp == True:
            tgt_pos = PositionalEncoding_Experiment(self.dim_embedding,seq_len_tgt,device=self.device)
            src_pos = PositionalEncoding_Experiment(self.dim_embedding,seq_len_src,device=self.device)
            # Embeddings finaux
            src = self.dropout_enc(src_embedding + src_pos(source)) # [batch, seq_len_src, dim_embedding]
            tgt = self.dropout_enc(tgt_embedding + tgt_pos(target)) # [batch, seq_len_tgt, dim_embedding]

          else:
            tgt_pos_embedding = torch.arange(0, seq_len_tgt,device=self.device).expand(batch_size,seq_len_tgt)
            src_pos_embedding = torch.arange(0, seq_len_src,device=self.device).expand(batch_size,seq_len_src)

            # Embeddings finaux
            src = self.dropout_enc(src_embedding + self.embedding_pos_src(src_pos_embedding)) # [batch, seq_len_src, dim_embedding]
            tgt = self.dropout_enc(tgt_embedding + self.embedding_pos_tgt(tgt_pos_embedding)) # [batch, seq_len_tgt, dim_embedding]

          # Création des masks pour la source et target
          src_seq_len = src.shape[1]
          tgt_seq_len = tgt.shape[1]
          # Masks
          tgt_mask = (torch.triu(torch.ones((tgt_seq_len,tgt_seq_len),device=self.device)) == 0).transpose(0, 1)

          # Padding masks
          src_padding_mask = (source == self.src_pad_idx)
          tgt_padding_mask = (target == self.tgt_pad_idx)

          # Appel au transformer
          if self.torch_fct_transformer[0]: # Utilisation du transformer de Pytorch
```

```
            output = self.transformer_model(src, tgt, tgt_mask=tgt_mask,
                                            src_key_padding_mask=src_padding_mask, tgt_key_padding_mask=tgt_padding_mask)
          else: # Utilisation du transformer "homemade"
            output = self.transformer_model(src, tgt, tgt_mask_attn=tgt_mask,
                                            src_key_padding_mask=src_padding_mask, tgt_key_padding_mask=tgt_padding_mask)


          return  self._sequential(output)
```

Greedy search

Here you have to implement a geedy search to generate a target translation from a trained model and an input source string. The next token will simply be the most probable one.

In [ ]:
```
"""
NB : Greedy Search est en fait cas particulier de Beam Search où beam_width = 1 et max_target = 1
"""

def greedy_search(
        model: nn.Module,
        source: str,
        src_vocab: Vocab,
        tgt_vocab: Vocab,
        src_tokenizer,
        device: str,
        max_sentence_length: int,
    ) -> str:
    """Do a beam search to produce probable translations.

    Args
    ----
        model: The translation model. Assumes it produces logits score (before softmax).
        source: The sentence to translate.
        src_vocab: The source vocabulary.
        tgt_vocab: The target vocabulary.
        device: Device to which we make the inference.
        max_sentence_length: Maximum number of tokens for the translated sentence.

    Output
    ------
        sentence: The translated source sentence.
    """
    src_tokens = ['<bos>'] + src_tokenizer(source) + ['<eos>']
    src_tokens = src_vocab(src_tokens)

    tgt_tokens = ['<bos>']
    tgt_tokens = tgt_vocab(tgt_tokens)

    # To tensor and add unitary batch dimension
    src_tokens = torch.LongTensor(src_tokens).to(device)
    tgt_tokens = torch.LongTensor(tgt_tokens).unsqueeze(dim=0).to(device)
    target_probs = torch.FloatTensor([1]).to(device)
    model.to(device)

    EOS_IDX = tgt_vocab['<eos>']
    with torch.no_grad():
      while tgt_tokens.shape[1] < max_sentence_length:
        batch_size, n_tokens = tgt_tokens.shape

        # Get next tokens
        src = einops.repeat(src_tokens, 't -> b t', b=tgt_tokens.shape[0])
        predicted = model.forward(src, tgt_tokens)
        predicted = torch.softmax(predicted, dim=-1)
        probs, predicted = predicted[:, -1].topk(k=1, dim=-1) # On garde le token le plus probable

        tgt_tokens = append_beams(tgt_tokens, predicted) # On l'ajoute à la phrase
```

```
        if tgt_vocab['<eos>'] in tgt_tokens: # Si on prédit le token '<eo>' alors c'est la fin de phrase
          if tgt_tokens.shape[1] < max_sentence_length: # Si la longueur de la phrase est inférieur au minimum requis, on ajoute du padding
            padding = torch.zeros((max_sentence_length-tgt_tokens.shape[1], 1), dtype=torch.long, device=device).T
            tgt_tokens = torch.cat((tgt_tokens, padding), dim=1)
        #print(tgt_tokens)

    for tgt_sentence in tgt_tokens:
        tgt_sentence = list(tgt_sentence)[1:]  # Remove <bos> token
        tgt_sentence = list(takewhile(lambda t: t != EOS_IDX, tgt_sentence))
        tgt_sentence = ' '.join(tgt_vocab.lookup_tokens(tgt_sentence))

    sentence = [tgt_sentence]
    sentence = [beautify(s) for s in sentence]

    # Join the sentence with its likelihood
    sentence = [(s, p.item()) for s, p in zip(sentence, target_probs)]

    return sentence
```

Beam search

Beam search is a smarter way of producing a sequence of tokens from an autoregressive model than just using a greedy search.

The greedy search always choose the most probable token as the unique and only next target token, and repeat this processus until the \\<*eos*> token is predicted.

Instead, the beam search selects the k-most probable tokens at each step. From those k tokens, the current sequence is duplicated k times and the k tokens are appended to the k sequences to produce new k sequences.

You don't have to understand this code, but understanding this code once the TP is over could improve your torch tensors skills.

More explanations

Since it is done at each step, the number of sequences grows exponentially (k sequences after the first step, k² sequences after the second...). In order to keep the number of sequences low, we remove sequences except the top-s most likely sequences. To do that, we keep track of the likelihood of each sequence.

Formally, we define $s = [s_1, \dots, s_{N_s}]$ as the source sequence made of N_s tokens. We also define $t^i = [t_1, \dots, t_i]$ as the target sequence at the beginning of the step i.

The output of the model parameterized by θ is:

$$T_{i+1} = p(t_{i+1}|s, t^i; \theta)$$

Where T_{i+1} is the distribution of the next token t_{i+1}.

Then, we define the likelihood of a target sentence $t = [t_1, \dots, t_{N_t}]$ as:

$$L(t) = \prod_{i=1}^{N_t-1} p(t_{i+1}|s, t_i; \theta)$$

Pseudocode of the beam search:

```
source: [N_s source tokens]  # Shape of [total_source_tokens]
target: [1, <bos> token]  # Shape of [n_sentences, current_target_tokens]
target_prob: [1]  # Shape of [n_sentences]
# We use `n_sentences` as the batch_size dimension

while current_target_tokens <= max_target_length:
    source = repeat(source, n_sentences)  # Shape of [n_sentences, total_source_tokens]
    predicted = model(source, target)[:, -1]  # Predict the next token distributions of all the n_sentences
```

```
        tokens_idx, tokens_prob = topk(predicted, k)

        # Append the `n_sentences * k` tokens to the `n_sentences` sentences
        target = repeat(target, k)  # Shape of [n_sentences * k, current_target_tokens]
        target = append_tokens(target, tokens_idx)  # Shape of [n_sentences * k, current_target_tokens + 1]

        # Update the sentences probabilities
        target_prob = repeat(target_prob, k)  # Shape of [n_sentences * k]
        target_prob *= tokens_prob

        if n_sentences * k >= max_sentences:
            target, target_prob = topk_prob(target, target_prob, k=max_sentences)
        else:
            n_sentences *= k

        current_target_tokens += 1
```

In [ ]:

```
def beautify(sentence: str) -> str:
    """Removes useless spaces.
    """
    punc = {'.', ',', ';'}
    for p in punc:
        sentence = sentence.replace(f' {p}', p)

    links = {'-', "'"}
    for l in links:
        sentence = sentence.replace(f'{l} ', l)
        sentence = sentence.replace(f' {l}', l)

    return sentence
```

In [ ]:

```
def indices_terminated(
        target: torch.FloatTensor,
        eos_token: int
    ) -> tuple:
    """Split the target sentences between the terminated and the non-terminated
    sentence. Return the indices of those two groups.

    Args
    ----
        target: The sentences.
            Shape of [batch_size, n_tokens].
        eos_token: Value of the End-of-Sentence token.

    Output
    ------
        terminated: Indices of the terminated sentences (who's got the eos_token).
            Shape of [n_terminated, ].
        non-terminated: Indices of the unfinished sentences.
            Shape of [batch_size-n_terminated, ].
    """
    terminated = [i for i, t in enumerate(target) if eos_token in t]
    non_terminated = [i for i, t in enumerate(target) if eos_token not in t]
    return torch.LongTensor(terminated), torch.LongTensor(non_terminated)


def append_beams(
        target: torch.FloatTensor,
        beams: torch.FloatTensor
    ) -> torch.FloatTensor:
    """Add the beam tokens to the current sentences.
    Duplicate the sentences so one token is added per beam per batch.

    Args
    ----
```

```
        target: Batch of unfinished sentences.
            Shape of [batch_size, n_tokens].
        beams: Batch of beams for each sentences.
            Shape of [batch_size, n_beams].

    Output
    ------
        target: Batch of sentences with one beam per sentence.
            Shape of [batch_size * n_beams, n_tokens+1].
    """
    batch_size, n_beams = beams.shape
    n_tokens = target.shape[1]

    target = einops.repeat(target, 'b t -> b c t', c=n_beams)  # [batch_size, n_beams, n_tokens]
    beams = beams.unsqueeze(dim=2)  # [batch_size, n_beams, 1]

    target = torch.cat((target, beams), dim=2)  # [batch_size, n_beams, n_tokens+1]
    target = target.view(batch_size*n_beams, n_tokens+1)  # [batch_size * n_beams, n_tokens+1]
    return target


def beam_search(
        model: nn.Module,
        source: str,
        src_vocab: Vocab,
        tgt_vocab: Vocab,
        src_tokenizer,
        device: str,
        beam_width: int,
        max_target: int,
        max_sentence_length: int,
    ) -> list:
    """Do a beam search to produce probable translations.

    Args
    ----
        model: The translation model. Assumes it produces linear score (before softmax).
        source: The sentence to translate.
        src_vocab: The source vocabulary.
        tgt_vocab: The target vocabulary.
        device: Device to which we make the inference.
        beam_width: Number of top-k tokens we keep at each stage.
        max_target: Maximum number of target sentences we keep at the end of each stage.
        max_sentence_length: Maximum number of tokens for the translated sentence.

    Output
    ------
        sentences: List of sentences orderer by their likelihood.
    """
    src_tokens = ['<bos>'] + src_tokenizer(source) + ['<eos>']
    src_tokens = src_vocab(src_tokens)

    tgt_tokens = ['<bos>']
    tgt_tokens = tgt_vocab(tgt_tokens)

    # To tensor and add unitary batch dimension
    src_tokens = torch.LongTensor(src_tokens).to(device)
    tgt_tokens = torch.LongTensor(tgt_tokens).unsqueeze(dim=0).to(device)
    target_probs = torch.FloatTensor([1]).to(device)
    model.to(device)

    EOS_IDX = tgt_vocab['<eos>']
    with torch.no_grad():
        while tgt_tokens.shape[1] < max_sentence_length:
            batch_size, n_tokens = tgt_tokens.shape

            # Get next beams
            src = einops.repeat(src_tokens, 't -> b t', b=tgt_tokens.shape[0])
```

```
        predicted = model.forward(src, tgt_tokens)
        predicted = torch.softmax(predicted, dim=-1)
        probs, predicted = predicted[:, -1].topk(k=beam_width, dim=-1)

        # Separe between terminated sentences and the others
        idx_terminated, idx_not_terminated = indices_terminated(tgt_tokens, EOS_IDX)
        idx_terminated, idx_not_terminated = idx_terminated.to(device), idx_not_terminated.to(device)

        tgt_terminated = torch.index_select(tgt_tokens, dim=0, index=idx_terminated)
        tgt_probs_terminated = torch.index_select(target_probs, dim=0, index=idx_terminated)

        filter_t = lambda t: torch.index_select(t, dim=0, index=idx_not_terminated)
        tgt_others = filter_t(tgt_tokens)
        tgt_probs_others = filter_t(target_probs)
        predicted = filter_t(predicted)
        probs = filter_t(probs)

        # Add the top tokens to the previous target sentences
        tgt_others = append_beams(tgt_others, predicted)

        # Add padding to terminated target
        padd = torch.zeros((len(tgt_terminated), 1), dtype=torch.long, device=device)
        tgt_terminated = torch.cat(
            (tgt_terminated, padd),
            dim=1
        )

        # Update each target sentence probabilities
        tgt_probs_others = torch.repeat_interleave(tgt_probs_others, beam_width)
        tgt_probs_others *= probs.flatten()
        tgt_probs_terminated *= 0.999  # Penalize short sequences overtime

        # Group up the terminated and the others
        target_probs = torch.cat(
            (tgt_probs_others, tgt_probs_terminated),
            dim=0
        )
        tgt_tokens = torch.cat(
            (tgt_others, tgt_terminated),
            dim=0
        )

        # Keep only the top `max_target` target sentences
        if target_probs.shape[0] <= max_target:
            continue

        target_probs, indices = target_probs.topk(k=max_target, dim=0)
        tgt_tokens = torch.index_select(tgt_tokens, dim=0, index=indices)

    sentences = []
    for tgt_sentence in tgt_tokens:
        tgt_sentence = list(tgt_sentence)[1:]  # Remove <bos> token
        tgt_sentence = list(takewhile(lambda t: t != EOS_IDX, tgt_sentence))
        tgt_sentence = ' '.join(tgt_vocab.lookup_tokens(tgt_sentence))
        sentences.append(tgt_sentence)

    sentences = [beautify(s) for s in sentences]

    # Join the sentences with their likelihood
    sentences = [(s, p.item()) for s, p in zip(sentences, target_probs)]
    # Sort the sentences by their likelihood
    sentences = [(s, p) for s, p in sorted(sentences, key=lambda k: k[1], reverse=True)]

    return sentences
```

Training loop

This is a basic training loop code. It takes a big configuration dictionnary to avoid never ending arguments in the functions. We use Weights and Biases to log the trainings. It logs every training informations and model performances in the cloud. You have to create an account to use it. Every accounts are free for individuals or research teams.

In [ ]:
```
## EXPERIMENT  - METRIC

def blue_score(
        real_sentence: str,
        predict_sentence: str,
        max_n: int,
    ) -> float:
    """Compute the blue score accuracy.

    Args
    ----
        real_sentence: String of the real sentence.
        predict_sentence: String of the predicted sentence.
        max_n :  the maximum n-gram we want to use. E.g. if max_n=3, we will use unigrams, bigrams and trigrams

    Output
    ------
        blue: Scalar of blue score value.
    """

    real_sentence = [real_sentence[:-1].split(" ")]
    predict_sentence = [[predict_sentence[:-1].split(" ")]]
    weights = np.ones(max_n)/max_n

    blue = torchtext.data.metrics.bleu_score(real_sentence, predict_sentence, max_n=max_n, weights=weights)
    return blue


def loop_blue_score(
        model   : nn.Module,
        config  : dict,
        dataset : list,
        sentence_idx : list
    ) -> list:
    """Compute the blue score on several sentences.

    Args
    ----
        model: trained model
        config: Additional parameters from config
        dataset : list of tuples of sentences (source (EN), target(FR))
        sentence_idx : list of indices
    Output
    ------
        list_blue_score: list of blue score values
    """
    max_n = [1,2,3]
    blue = np.zeros(len(max_n))
    for i in range(0,len(sentence_idx)):
      source, target = dataset[sentence_idx[i]]

      if config['search'] == 'beam_search' :
          pred, prob = beam_search(
            model,
            source,
            config['src_vocab'],
            config['tgt_vocab'],
            config['src_tokenizer'],
            config['device'],  # It can take a lot of VRAM
            beam_width=10,
            max_target=100,
            max_sentence_length=config['max_sequence_length'],
        )[0]
      elif config['search'] == 'greedy_search' :
```

```
            pred, prob = greedy_search(
                model,
                source,
                config['src_vocab'],
                config['tgt_vocab'],
                config['src_tokenizer'],
                config['device'],  # It can take a lot of VRAM
                max_sentence_length=config['max_sequence_length'],
            )[0]

        for j in range(0, len(max_n)):
            blue[j] += blue_score(target,pred,max_n[j])

    blue = blue/len(sentence_idx)
    list_blue_score = [(max_n[i], blue[i]) for i in range(0, len(max_n))]
    return list_blue_score
```

In [ ]:

```
def print_logs(dataset_type: str, logs: dict):
    """Print the logs.

    Args
    ----
        dataset_type: Either "Train", "Eval", "Test" type.
        logs: Containing the metric's name and value.
    """
    desc = [
        f'{name}: {value:.2f}'
        for name, value in logs.items()
    ]
    desc = '\t'.join(desc)
    desc = f'{dataset_type} -\t' + desc
    desc = desc.expandtabs(5)
    print(desc)


def topk_accuracy(
        real_tokens: torch.FloatTensor,
        probs_tokens: torch.FloatTensor,
        k: int,
        tgt_pad_idx: int,
    ) -> torch.FloatTensor:
    """Compute the top-k accuracy.
    We ignore the PAD tokens.

    Args
    ----
        real_tokens: Real tokens of the target sentence.
            Shape of [batch_size * n_tokens].
        probs_tokens: Tokens probability predicted by the model.
            Shape of [batch_size * n_tokens, n_target_vocabulary].
        k: Top-k accuracy threshold.
        src_pad_idx: Source padding index value.

    Output
    ------
        acc: Scalar top-k accuracy value.
    """
    total = (real_tokens != tgt_pad_idx).sum()

    _, pred_tokens = probs_tokens.topk(k=k, dim=-1)  # [batch_size * n_tokens, k]
    real_tokens = einops.repeat(real_tokens, 'b -> b k', k=k)  # [batch_size * n_tokens, k]

    good = (pred_tokens == real_tokens) & (real_tokens != tgt_pad_idx)
    acc = good.sum() / total
    return acc
```

```python
def loss_batch(
        model: nn.Module,
        source: torch.LongTensor,
        target: torch.LongTensor,
        config: dict,
    )-> dict:
    """Compute the metrics associated with this batch.
    The metrics are:
        - loss
        - top-1 accuracy
        - top-5 accuracy
        - top-10 accuracy

    Args
    ----
        model: The model to train.
        source: Batch of source tokens.
            Shape of [batch_size, n_src_tokens].
        target: Batch of target tokens.
            Shape of [batch_size, n_tgt_tokens].
        config: Additional parameters.

    Output
    ------
        metrics: Dictionnary containing evaluated metrics on this batch.
    """
    device = config['device']
    loss_fn = config['loss'].to(device)
    metrics = dict()

    source, target = source.to(device), target.to(device)
    target_in, target_out = target[:, :-1], target[:, 1:]

    # Loss
    pred = model(source, target_in)  # [batch_size, n_tgt_tokens-1, n_vocab]
    pred = pred.view(-1, pred.shape[2])  # [batch_size * (n_tgt_tokens - 1), n_vocab]
    target_out = target_out.flatten()  # [batch_size * (n_tgt_tokens - 1),]
    metrics['loss'] = loss_fn(pred, target_out)

    # Accuracy - we ignore the padding predictions
    for k in [1, 5, 10]:
        metrics[f'top-{k}'] = topk_accuracy(target_out, pred, k, config['tgt_pad_idx'])
    return metrics


def eval_model(model: nn.Module, dataloader: DataLoader, config: dict) -> dict:
    """Evaluate the model on the given dataloader.
    """
    device = config['device']
    logs = defaultdict(list)

    model.to(device)
    model.eval()

    with torch.no_grad():
        for source, target in dataloader:
            metrics = loss_batch(model, source, target, config)
            for name, value in metrics.items():
                logs[name].append(value.cpu().item())

    for name, values in logs.items():
        logs[name] = np.mean(values)
    return logs


def train_model(model: nn.Module, config: dict):
    """Train the model in a teacher forcing manner.
    """
```

```
    train_loader, val_loader = config['train_loader'], config['val_loader']
    train_dataset, val_dataset = train_loader.dataset.dataset, val_loader.dataset.dataset
    optimizer = config['optimizer']
    clip = config['clip']
    device = config['device']

    columns = ['epoch']
    for mode in ['train', 'validation']:
        columns += [
            f'{mode} - {colname}'
            for colname in ['source', 'target', 'predicted', 'likelihood']
        ]
    log_table = wandb.Table(columns=columns)


    print(f'Starting training for {config["epochs"]} epochs, using {device}.')
    for e in range(config['epochs']):
        print(f'\nEpoch {e+1}')

        model.to(device)
        model.train()
        logs = defaultdict(list)

        for batch_id, (source, target) in enumerate(train_loader):
            optimizer.zero_grad()

            metrics = loss_batch(model, source, target, config)
            loss = metrics['loss']

            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
            optimizer.step()

            for name, value in metrics.items():
                logs[name].append(value.cpu().item())  # Don't forget the '.item' to free the cuda memory

            if batch_id % config['log_every'] == 0:
                for name, value in logs.items():
                    logs[name] = np.mean(value)

                train_logs = {
                    f'Train - {m}': v
                    for m, v in logs.items()
                }
                wandb.log(train_logs)
                logs = defaultdict(list)

        # Logs
        if len(logs) != 0:
            for name, value in logs.items():
                logs[name] = np.mean(value)
            train_logs = {
                f'Train - {m}': v
                for m, v in logs.items()
            }
        else:
            logs = {
                m.split(' - ')[1]: v
                for m, v in train_logs.items()
            }

        print_logs('Train', logs)

        logs = eval_model(model, val_loader, config)
        print_logs('Eval', logs)
        val_logs = {
            f'Validation - {m}': v
            for m, v in logs.items()
```

```
        }

        val_source, val_target = val_dataset[ torch.randint(len(val_dataset), (1,)) ]
        if config['search'] == 'beam_search' :
          val_pred, val_prob = beam_search(
              model,
              val_source,
              config['src_vocab'],
              config['tgt_vocab'],
              config['src_tokenizer'],
              device,  # It can take a lot of VRAM
              beam_width=10,
              max_target=100,
              max_sentence_length=config['max_sequence_length'],
          )[0]
        elif config['search'] == 'greedy_search' :
          val_pred, val_prob = greedy_search(
              model,
              val_source,
              config['src_vocab'],
              config['tgt_vocab'],
              config['src_tokenizer'],
              device,  # It can take a lot of VRAM
              max_sentence_length=config['max_sequence_length'],
          )[0]
          val_prob = None
        else :
          print (f"Type of search ({config['search']}) not supported")

        print(val_source)
        print(val_pred)
        logs = {**train_logs, **val_logs}  # Merge dictionnaries
        wandb.log(logs)  # Upload to the WandB cloud

        # Table logs
        train_source, train_target = train_dataset[ torch.randint(len(train_dataset), (1,)) ]
        if config['search'] == 'beam_search' :
          train_pred, train_prob = beam_search(
              model,
              train_source,
              config['src_vocab'],
              config['tgt_vocab'],
              config['src_tokenizer'],
              device,  # It can take a lot of VRAM
              beam_width=10,
              max_target=100,
              max_sentence_length=config['max_sequence_length'],
          )[0]
        elif config['search'] == 'greedy_search' :
          train_pred, train_prob = greedy_search(
              model,
              train_source,
              config['src_vocab'],
              config['tgt_vocab'],
              config['src_tokenizer'],
              device,  # It can take a lot of VRAM
              max_sentence_length=config['max_sequence_length'],
          )[0]
          train_prob = None
        else :
          print (f"Type of search ({config['search']}) not suported")

        ## Blue Score Train
        if config['exp_metric'] == 'blue_score':
          train_blue_score = loop_blue_score(model,config,train_dataset, config['train_blue_idx'])
          val_blue_score = loop_blue_score(model,config,val_dataset, config['val_blue_idx'])

          blue_train_logs = {
```

```
                f'Blue Score ({m}-grams)': v
                 for m, v in train_blue_score
            }
        blue_val_logs = {
                f'Blue Score ({m}-grams)': v
                 for m, v in val_blue_score
            }
        # Print Blue Score
        print('\nExperiment - Metric')
        print_logs('Train', blue_train_logs)
        print_logs('Eval', blue_val_logs)

        logs = {**blue_train_logs, **blue_val_logs}  # Merge dictionnaries
        wandb.log(logs)  # Upload to the WandB cloud
      ##

      data = [
          e + 1,
          train_source, train_target, train_pred, train_prob,
          val_source, val_target, val_pred, val_prob,
      ]
      log_table.add_data(*data)

  # Log the table at the end of the training
  wandb.log({'Model predictions': log_table})
```

Training the models

We can now finally train the models. Choose the right hyperparameters, play with them and try to find ones that lead to good models and good training curves. Try to reach a loss under 1.0.

So you know, it is possible to get descent results with approximately 20 epochs. With CUDA enabled, one epoch, even on a big model with a big dataset, shouldn't last more than 10 minutes. A normal epoch is between 1 to 5 minutes.

This is considering Colab Pro, we should try using free Colab to get better estimations.

To test your implementations, it is easier to try your models in a CPU instance. Indeed, Colab reduces your GPU instances priority with the time you recently past using GPU instances. It would be sad to consume all your GPU time on implementation testing. Moreover, you should try your models on small datasets and with a small number of parameters. For exemple, you could set:

```
MAX_SEQ_LEN = 10
MIN_TOK_FREQ = 20
dim_embedding = 40
dim_hidden = 60
n_layers = 1
```

You usually don't want to log anything onto WandB when testing your implementation. To deactivate WandB without having to change any line of code, you can type `!wandb offline` in a cell.

Once you have rightly implemented the models, you can train bigger models on bigger datasets. When you do this, do not forget to change the runtime as GPU (and use `!wandb online`)!

In [ ]:
```
# Checking GPU and logging to wandb

!wandb login

!nvidia-smi
# 0c01e599de3ef5e5a801b3b02166cb76eee87eda : Renaud

# b2208215d33eed3434e8409697c7ba75b44bf9e8 : Morgan
```

```
wandb: You can find your API key in your browser here: https://wandb.ai/authorize
wandb: Paste an API key from your profile and hit enter, or press ctrl+c to quit:
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
Thu Apr  7 20:29:05 2022
+-----------------------------------------------------------------------------+
```

```
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   43C    P0    27W / 250W |      0MiB / 16280MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

In [ ]:
```
# Instanciate the datasets

# MAX_SEQ_LEN = 10    # Original 60
# MIN_TOK_FREQ = 20   # Original 2
MAX_SEQ_LEN = 60
MIN_TOK_FREQ = 2
train_dataset, val_dataset = build_datasets(
    MAX_SEQ_LEN,
    MIN_TOK_FREQ,
    en_tokenizer,
    fr_tokenizer,
    train,
    valid,
)


print(f'English vocabulary size: {len(train_dataset.en_vocab):,}')
print(f'French vocabulary size: {len(train_dataset.fr_vocab):,}')

print(f'\nTraining examples: {len(train_dataset):,}')
print(f'Validation examples: {len(val_dataset):,}')
```

```
English vocabulary size: 11,196
French vocabulary size: 16,970

Training examples: 173,104
Validation examples: 19,235
```

In [ ]:
```
# Build the model, the dataloaders, optimizer and the loss function
# Log every hyperparameters and arguments into the config dictionnary

config = {
    # General parameters
    'epochs': 25,          # Original 5
    'batch_size': 128,
    'lr': 1e-3,
    'betas': (0.9, 0.99),
    'clip': 5,
    'device': 'cuda' if torch.cuda.is_available() else 'cpu',
    'search' : 'beam_search', #'greedy_search'
    #'search' : 'greedy_search',
    'exp_metric' : None, # 'blue_score' or None
    'train_blue_idx' : [torch.randint(len(train_dataset), (1,)) for i in range(0,20)],
    'val_blue_idx': [torch.randint(len(val_dataset), (1,)) for i in range(0,20)],

    # Model parameters
    'n_tokens_src': len(train_dataset.en_vocab),
    'n_tokens_tgt': len(train_dataset.fr_vocab),
    'n_heads': 4,
    # 'dim_embedding': 40,  # Original 196
    # 'dim_hidden': 60,     # Original 256
```

```
    # 'n_layers': 3,        # Original 3
    'dim_embedding': 300,  # Original 196
    'dim_hidden': 256,     # Original 256
    'n_layers': 3,        # Original 3
    'dropout': 0.1,
    'model_type': 'RNN',  # A modifier
    'torch_fct_translation': False,  # A modifier
    'MLP_param_RNN_GRU' : ['LeakyReLU01',1,1], #Activation fct ("LeakyReLU01","ELU","Mish"), number of layers in the MLP, scale of hidden layer
    'MLP_param_transformer' : ['ELU',2,6], #Activation fct ("LeakyReLU01","ELU","Mish"), number of layers in the MLP, scale of hidden layer
    'torch_fct_transformer' : [False, False, False, False], # Utiliser Pytorch pour [Transformer, Encoder, Decoder, MultiheadAttention]
    'positional_embeddings_exp' : False, # Mettre True si on veut tester d'autres positional Embdeddings dans le Transformer

    # Others
    'max_sequence_length': MAX_SEQ_LEN,
    'min_token_freq': MIN_TOK_FREQ,
    'src_vocab': train_dataset.en_vocab,
    'tgt_vocab': train_dataset.fr_vocab,
    'src_tokenizer': en_tokenizer,
    'tgt_tokenizer': fr_tokenizer,
    'src_pad_idx': train_dataset.en_vocab['<pad>'],
    'tgt_pad_idx': train_dataset.fr_vocab['<pad>'],
    'seed': 0,
    'log_every': 50,  # Number of batches between each wandb logs
}

torch.manual_seed(config['seed'])

config['train_loader'] = DataLoader(
    train_dataset,
    batch_size=config['batch_size'],
    shuffle=True,
    collate_fn=lambda batch: generate_batch(batch, config['src_pad_idx'], config['tgt_pad_idx'])
)

config['val_loader'] = DataLoader(
    val_dataset,
    batch_size=config['batch_size'],
    shuffle=True,
    collate_fn=lambda batch: generate_batch(batch, config['src_pad_idx'], config['tgt_pad_idx'])
)
"""
model = TranslationRNN(
    config['n_tokens_src'],
    config['n_tokens_tgt'],
    config['dim_embedding'],
    config['dim_hidden'],
    config['n_layers'],
    config['dropout'],
    config['src_pad_idx'],
    config['tgt_pad_idx'],
    config['model_type'],
    config['torch_fct_translation'],
)
"""

model = TranslationTransformer(
    config['n_tokens_src'],
    config['n_tokens_tgt'],
    config['n_heads'],
    config['dim_embedding'],
    config['dim_hidden'],
    config['n_layers'],
    config['dropout'],
    config['src_pad_idx'],
    config['tgt_pad_idx'],
    config['torch_fct_transformer'],
    config['positional_embeddings_exp']
)
```

```
config['optimizer'] = optim.Adam(
    model.parameters(),
    lr=config['lr'],
    betas=config['betas'],
)

weight_classes = torch.ones(config['n_tokens_tgt'], dtype=torch.float)
weight_classes[config['tgt_vocab']['<unk>']] = 0.1  # Lower the importance of that class
config['loss'] = nn.CrossEntropyLoss(
    weight=weight_classes,
    ignore_index=config['tgt_pad_idx'],  # We do not have to learn those
)

summary(
    model,
    input_size=[
        (config['batch_size'], config['max_sequence_length']),
        (config['batch_size'], config['max_sequence_length'])
    ],
    dtypes=[torch.long, torch.long],
    depth=3,
)
```

Out[ ]:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
TranslationTransformer                   --                        --
├─Embedding: 1-1                         [128, 60, 300]            3,358,800
├─Embedding: 1-2                         [128, 60, 300]            5,091,000
├─Embedding: 1-3                         [128, 60, 300]            3,358,800
├─Dropout: 1-4                           [128, 60, 300]            --
├─Embedding: 1-5                         [128, 60, 300]            5,091,000
├─Dropout: 1-6                           [128, 60, 300]            --
├─Transformer: 1-7                       [128, 60, 300]            --
│    └─TransformerEncoder: 2-1           [128, 60, 300]            --
│    │    └─LayerNorm: 3-1               [128, 60, 300]            600
│    └─TransformerDecoder: 2-2           [128, 60, 300]            --
│    │    └─LayerNorm: 3-2               [128, 60, 300]            600
├─Sequential: 1-8                        [128, 60, 16970]          --
│    └─Linear: 2-3                       [128, 60, 1800]           541,800
│    └─Dropout: 2-4                      [128, 60, 1800]           --
│    └─ELU: 2-5                          [128, 60, 1800]           --
│    └─LayerNorm: 2-6                    [128, 60, 1800]           3,600
│    └─Linear: 2-7                       [128, 60, 16970]          30,562,970
==========================================================================================
Total params: 48,009,170
Trainable params: 48,009,170
Non-trainable params: 0
Total mult-adds (G): 6.15
==========================================================================================
Input size (MB): 0.12
Forward/backward pass size (MB): 1374.41
Params size (MB): 192.04
Estimated Total Size (MB): 1566.57
==========================================================================================
```

In [ ]:

```
!wandb online  # online / offline to activate or deactivate WandB logging

with wandb.init(
        config=config,
        project='INF8225 - TP3 - Transformer Final Test',  # Title of your project
        group='Transformer - Final model',  # In what group of runs do you want this run to be in?
        save_code=True,
    ):
    train_model(model, config)
```

```
W&B online, running your script from this directory will now sync to the cloud.
```

Tracking run with wandb version 0.12.13

Run data is saved locally in `/content/wandb/run-20220407_221333-2vyz2qam`

Syncing run **fast-waterfall-1** to Weights & Biases (docs)

```
Starting training for 25 epochs, using cuda.

Epoch 1
Train -   loss: 2.76     top-1: 0.48    top-5: 0.68    top-10: 0.75
Eval -    loss: 2.57     top-1: 0.50    top-5: 0.71    top-10: 0.77
Are we allowed to take pictures here?
Est-ce que nous allons ici ?

Epoch 2
Train -   loss: 2.17     top-1: 0.55    top-5: 0.77    top-10: 0.83
Eval -    loss: 2.03     top-1: 0.58    top-5: 0.79    top-10: 0.84
The traffic light changed to red.
Le dîner s'est familier.

Epoch 3
Train -   loss: 1.84     top-1: 0.58    top-5: 0.81    top-10: 0.87
Eval -    loss: 1.79     top-1: 0.62    top-5: 0.82    top-10: 0.87
We can't trust Tom anymore.
Nous ne pouvons plus confiance à Tom.

Epoch 4
Train -   loss: 1.43     top-1: 0.67    top-5: 0.86    top-10: 0.90
Eval -    loss: 1.66     top-1: 0.64    top-5: 0.84    top-10: 0.89
Is he breathing?
Est-ce qu'il ?

Epoch 5
Train -   loss: 1.35     top-1: 0.67    top-5: 0.88    top-10: 0.92
Eval -    loss: 1.57     top-1: 0.66    top-5: 0.86    top-10: 0.90
Our company's showroom was a hit with the ladies.
Notre entreprise a été valeur d'enfance.

Epoch 6
Train -   loss: 1.38     top-1: 0.67    top-5: 0.87    top-10: 0.92
Eval -    loss: 1.52     top-1: 0.67    top-5: 0.87    top-10: 0.90
Tom doesn't blame you for anything.
Tom ne vous reproche rien.

Epoch 7
Train -   loss: 1.30     top-1: 0.68    top-5: 0.88    top-10: 0.92
Eval -    loss: 1.48     top-1: 0.68    top-5: 0.87    top-10: 0.91
Some people think that it is difficult for a native speaker of English to learn Chinese, but I disagree.
Certaines personnes pensent que ça ne pense qu'une langue maternelle.

Epoch 8
Train -   loss: 1.14     top-1: 0.71    top-5: 0.90    top-10: 0.95
Eval -    loss: 1.46     top-1: 0.69    top-5: 0.88    top-10: 0.91
I asked what he was going to do.
J'ai demandé ce qu'il allait faire.

Epoch 9
Train -   loss: 1.13     top-1: 0.71    top-5: 0.91    top-10: 0.95
Eval -    loss: 1.44     top-1: 0.69    top-5: 0.88    top-10: 0.91
This watch is a real bargain.
Cette montre est une bonne affaire.

Epoch 10
Train -   loss: 0.96     top-1: 0.72    top-5: 0.94    top-10: 0.96
Eval -    loss: 1.43     top-1: 0.70    top-5: 0.88    top-10: 0.92
I thought I was being smart.
Je pensais que j'étais intelligente.

Epoch 11
Train -   loss: 1.08     top-1: 0.72    top-5: 0.90    top-10: 0.95
Eval -    loss: 1.41     top-1: 0.70    top-5: 0.89    top-10: 0.92
Everyone is very proud of you.
Tout le monde est très fier.

Epoch 12
Train -   loss: 0.93     top-1: 0.75    top-5: 0.93    top-10: 0.96
Eval -    loss: 1.40     top-1: 0.70    top-5: 0.89    top-10: 0.92
I have a big house.
```

```
J'ai une grande maison.

Epoch 13
Train -   loss: 1.11     top-1: 0.71    top-5: 0.92    top-10: 0.95
Eval -    loss: 1.39     top-1: 0.71    top-5: 0.89    top-10: 0.92
It's important that I hear this.
C'est important que j'entendre ça.

Epoch 14
Train -   loss: 0.97     top-1: 0.74    top-5: 0.93    top-10: 0.96
Eval -    loss: 1.38     top-1: 0.71    top-5: 0.89    top-10: 0.92
This one is the worst.
Celle-ci est la pire.

Epoch 15
Train -   loss: 0.93     top-1: 0.74    top-5: 0.94    top-10: 0.97
Eval -    loss: 1.38     top-1: 0.71    top-5: 0.89    top-10: 0.92
I want you to be prepared.
Je veux que tu sois préparé.

Epoch 16
Train -   loss: 0.84     top-1: 0.76    top-5: 0.94    top-10: 0.97
Eval -    loss: 1.37     top-1: 0.72    top-5: 0.89    top-10: 0.92
The door's locked.
La porte est verrouillée.

Epoch 17
Train -   loss: 0.84     top-1: 0.76    top-5: 0.94    top-10: 0.97
Eval -    loss: 1.37     top-1: 0.72    top-5: 0.90    top-10: 0.93
You have our respect.
Tu as notre respect.

Epoch 18
Train -   loss: 0.77     top-1: 0.76    top-5: 0.95    top-10: 0.98
Eval -    loss: 1.37     top-1: 0.72    top-5: 0.90    top-10: 0.93
She decided to resign her job.
Elle décida de démissionner son emploi.

Epoch 19
Train -   loss: 0.72     top-1: 0.78    top-5: 0.96    top-10: 0.98
Eval -    loss: 1.37     top-1: 0.72    top-5: 0.90    top-10: 0.93
I play with my son every night.
Je joue avec mon fils avec tous les nuits.

Epoch 20
Train -   loss: 0.80     top-1: 0.77    top-5: 0.95    top-10: 0.97
Eval -    loss: 1.36     top-1: 0.72    top-5: 0.90    top-10: 0.93
That was never our intention.
Ce n'était jamais notre intention.

Epoch 21
Train -   loss: 0.73     top-1: 0.79    top-5: 0.95    top-10: 0.98
Eval -    loss: 1.35     top-1: 0.73    top-5: 0.90    top-10: 0.93
He is often late for school.
Il est souvent en retard à l'école.

Epoch 22
Train -   loss: 0.68     top-1: 0.81    top-5: 0.96    top-10: 0.98
Eval -    loss: 1.35     top-1: 0.73    top-5: 0.90    top-10: 0.93
Is money important to you?
L'argent est important pour vous ?

Epoch 23
Train -   loss: 0.70     top-1: 0.80    top-5: 0.96    top-10: 0.98
Eval -    loss: 1.35     top-1: 0.73    top-5: 0.90    top-10: 0.93
How did you get here so fast?
Comment êtes-vous arrivée ici si rapidement ?

Epoch 24
Train -   loss: 0.68     top-1: 0.80    top-5: 0.97    top-10: 0.98
Eval -    loss: 1.35     top-1: 0.73    top-5: 0.90    top-10: 0.93
This looks like a trap.
Ça a l'air d'être un piège.
```

```
Epoch 25
Train -    loss: 0.79     top-1: 0.78    top-5: 0.95    top-10: 0.98
Eval -     loss: 1.35     top-1: 0.73    top-5: 0.90    top-10: 0.93
He can drive a car.
Il peut conduire une voiture.
```

Waiting for W&B process to finish... **(success).**

Run history:

Run summary:

Train - loss	0.79233
Train - top-1	0.78063
Train - top-10	0.97631
Train - top-5	0.94933
Validation - loss	1.3487
Validation - top-1	0.7285
Validation - top-10	0.93023
Validation - top-5	0.90208

Synced **fast-waterfall-1**: https://wandb.ai/morgiizz/INF8225%20-%20TP3%20-%20Transformer%20Final%20Test/runs/2vyz2qam
Synced 5 W&B file(s), 1 media file(s), 3 artifact file(s) and 1 other file(s)
Find logs at: `./wandb/run-20220407_221333-2vyz2qam/logs`

In [ ]:

```python
#sentence = "It is possible to try your work here."
sentence = "I'm too tired to walk."

preds = beam_search(
    model,
    sentence,
    config['src_vocab'],
    config['tgt_vocab'],
    config['src_tokenizer'],
    config['device'],
    beam_width=10,
    max_target=100,
    max_sentence_length=config['max_sequence_length']
)[:10]

for i, (translation, likelihood) in enumerate(preds):
    print(f'{i}. ({likelihood*100:.5f}%) \t {translation}')
```

```
0. (58.20621%)   Je suis trop fatiguée pour marcher.
1. (29.44964%)   Je suis trop fatigué pour marcher.
2. (2.80672%)    Je suis trop fatiguée pour continuer.
3. (0.75601%)    Je suis trop fatiguée pour continuer à marcher.
4. (0.36998%)    J'trop fatiguée pour marcher.
5. (0.31987%)    Je suis trop fatiguée pour marcher fatigué.
6. (0.19011%)    J'ai trop fatiguée pour marcher.
7. (0.15915%)    Je vais trop fatiguée pour marcher.
8. (0.13635%)    Je suis trop fatiguée pour continuer de marcher.
9. (0.13133%)    Je suis trop fatigués pour marcher.
```

Questions

Question 1

1. Explain the differences between Vanilla RNN, GRU-RNN, and Transformers.

First of all, here's a recap of the mathematical concepts used in vanilla RNN and GRU.

RNN Description from : https://pytorch.org/docs/master/generated/torch.nn.RNN.html?highlight=rnn#torch.nn.RNN) \ For each element in the input sequence, each layer computes the following

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

where : h_t is the hidden state at time t, x_t is the input at time t, and $h_{(t-1)}$ is the hidden state of the previous layer at time $t-1$ or the initial hidden state at time 0.

GRU Description from : https://pytorch.org/docs/master/_modules/torch/nn/modules/rnn.html#GRU \

For each element in the input sequence, each layer computes the following

$$\begin{aligned} r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\ z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\ n_t &= \tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-1)} + b_{hn})) \\ h_t &= (1 - z_t) * n_t + z_t * h_{(t-1)} \end{aligned}$$

where :h_t is the hidden state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the layer at time $t-1$ or the initial hidden state at time 0, and r_t, z_t, n_t are the reset, update, and new gates, respectively. σ is the sigmoid function, and $*$ is the element-wise product.

As we can see, on the contrary to RNN, GRU uses a concept of gates and we'll see why in our explanation below :

- RNN :
  - One problem with vanilla RNN is that it faces a short-term memory problem. This is due to the "vanishing gradient" and gradient explosion. Indeed, during backpropagation the gradient is used to update the weights but it depends on the influence of the previous layer. If the previous gradient is small, the next one will be even smaller. (For the exploding gradient, it is the contrary.). Thus with a small gradient, the effect on the weights' update will be low or will have no effect at all. Therefore, it has an impact on the learning capability of the model.
- GRU:
  - GRU inherits of the structure of RNNs. However, it adds different gates to balance the hidden states. The major difference with vanilla RNN is GRU's ability to update a memory cell using the R (reset), Z (update) and N (new gate) gates. The R gate allows to reset the state of the cell. The Z gate allows to update the state of the cell and N allows to create a new temporary output which considers the previous hidden layer and the value of R. We then obtain an output h which is a linear combination of Z and N. GRU is a much more flexible model, thus approaching LSTM. This has the benefit of increasing the memory capacity of the model. Indeed, it is capable of forgetting or focusing on previous/current hidden states and the input. However, if the number of GRU cells is too high, it is still possible to face the "vanishing gradient" problem. Thus, the GRU is not always very good at retaining context. This leads us to concept of "attention" for Transformers.
- Transformers:
  - Finally, in a Transformer, sentences are processed entirely rather than word by word. By doing so, there is no longer the risk of losing past information as was the case with previous architectures. Moreover, as mentioned before, the "Attention" mechanism allows the Transformer to compute similarity scores between words in a sentence and thus give the model information about the relationships between words to know which words to focus on. As we will see in the next question, there is also the need to add positional information of the words as their not processed sequentially.

In [ ]:

Question 2

1. Why is positionnal encoding necessary in Transformers and not in RNNs?

The RNN/GRU intrinsically take into account the word order. Indeed, for an input sentence, words enter one by one in a sequential way in the RNN allowing to take into account their positional information.

However with a Transformer, the input is not sequential words but the whole sequence is directly introduced in the model. Then the "Attention" concept allows to tell the model where to focus but there is then no information about the position. This is why positional embeddings are necessary for the Transformer : it gives positional information of each word to the model.

In [ ]:

Question 3

1. Describe the preprocessing process. Detail how the initial dataset is processed before being fed to the translation models.

First, we download the data and create a dataframe with the sentences in English and the corresponding sentences in French. Then, we split the data as follows: 90% training data, 10% validation data thanks to the function train_test_split(). Afterwards, we use the function get_tokenizer() to transform words of the sentences into tokens and we also define special tokens such as :

- unk --> for an unknown word
- pad --> token for padding
- bos --> token for the beggining of sentence
- eos --> token for the end of sentence

Then, here's how the data are preprocessed through the call of the function build_datasets():

1. We use the "preprocess" function in order to filter the dataset :
   - preprocess() removes the break line tag ('\n') and removes from the dataset the examples that contain at least one sentence whose length exceeds the set limit (for memory reasons).
2. We use the "build_vocab" function to create vocabularies based on the sample of sentence that we kept:
   - build_vocab() allows to build vocabularies (english and french) with a minimum occurrence condition on words to be integrated in the vocabulary. It also adds the token "unk" for unknown words.
3. We use "TranslationDataset" class to tokenize each sentence, ddd start("bos")/end("eos") tokens and put the results in an "english" tensor and a "french" tensor:
   - The function **getitem**() tokenizes sentences and adds start-of-sentence ("bos") and end-of-sentence ("eos") tokens for each of the sentences and saves it in two tensors (1 English, 1 French)

Finally, in the model configuration ("config" dictionary), we use the generate_batch() function to add padding so that each sentence of a batch has the same length.

In [ ]:

```

```

Small report - experiments

Once everything is working fine, you can explore and do some little research work.

For exemple, you can experiment with the hyperparameters. What are the effect of the differents hyperparameters with the final model performance? What about training time?

What are some other metrics you could have for machine translation? Can you compute them and add them to your WandB report?

Those are only examples, you can do whatever you think will be interesting. This part account for many points, *feel free to go wild!*

Make a small report about your experiments here.

Our experiment plan

To improve our models, we performed several tests to assess and tune hyper-parameters and methods used for the translation. We decided to perform the following tests :

EXPERIMENT 1 : RNN/GRU comparison (2 tests)

We wanted to compare RNN and GRU with the following parameters:

- Basic Parameter, 10 epochs, fully connected output.

EXPERIMENT 2 : Tuning of the MLP (multilayer perceptron) for RNN/GRU (9 tests)

We performed a fine tuning of the MLP for the best model obtained in the first test. We did the following tests:

- Comparison of the activation functions : leakyReLU, ELU, Mish - with MLP (4 layers, scale 1) on 5 epochs (3 tests)

- With the best activation function, tuning of the MLP on 3 epochs: (6 tests)
  - Scale : 2, 6 and 8
  - Number of layers : 2 and 3

EXPERIMENT 3 : Transformer's hyper-parameters (7 tests)

We performed tests on the transformer to tune the hyperparameters with a fully connected output :

- Comparison on the batch size: 64, 128, 256 - 5 epochs
- Comparison on the dimension of the embeddings: 100, 200, 300 - 5 epochs

We assumed that we need to fine a balance on the embedding dimension. On one hand, not enough embeddings will lead to a small space of embedding and thus bad translations. On the other hand, to many embedding dimensions will lead to a too large space and the model will probably struggle to create coherent translations.

- Comparison on the positional embedding initialization - 5 epoch

We wanted to compare the classic positional embedding (with nn.Embeddding) with another method to initiliaze them. (see class PositionalEncoding_Experiment)

EXPERIMENT 4 : Tuning of the MLP for the Transformer (4 tests)

We performed a tuning of the transformer's MLP with the best parameters determined before. As you we'll see in the following tests reports, those best parameters are :

- Batch size : 128
- Dim Embedding : 300
- Positional Embedding : Classic
- Activation function : ELU

On 5 epochs, tuning of the MLP : (6 tests)

- Scale : 6 and 8
- Number of layers : 2 and 3

EXPERIMENT 5 : Compare Greedy and Beam Search and Blue score (Transformer) (2 tests)

Here, we wanted to compare the Greedy and Beam Search methods. As we expected and because Greedy Search is a special case of Beam Search (with beam_width=1 and max_target=1), Beam Search is better to produce a coherent translation. We also introduced another metric to assess our translated sentences through the training and validation.

BLUE SCORE :
Blue score is a metric for evaluating a generated sentence to a reference sentence. A perfect match results in a score of 1.0. It works by counting matching n-grams in the predicted sentence to n-grams in the reference text, where 1-gram would be each token and a bigram comparison would be each word pair, etc. The comparison is made regardless of word order.

FINAL RUN : Best model (Transformer)

Thanks to all the previous tests, we were able to set our best model. The output can be seen above. We ran it on 25 epochs, it took 1h35 (~3min50 per epoch) with the following parameters :

- Batch size : 128
- Dim Embedding : 300
- Positional Embedding : Classic
- Activation function : ELU
- MLP : Scale = 6, Layers = 2
- Beam Search

You will find below our results for each tests and our analysis.

In [ ]:
```
%%html
<a href="https://wandb.ai/renaudlesperance/INF8225%20-%20TP3%20-%20Final%20Run/reports/RNN-vs-GRU--VmlldzoxODEwNjg5" target="_blank" > Experiment 1 - RNN vs GRU </a> </br>
<a href="https://wandb.ai/renaudlesperance/INF8225%20-%20TP3%20-%20Final%20Run/reports/GRU-Tuning-of-the-MLP--VmlldzoxODEwODEy" target="_blank" > Experiment 2 - MLP Tuning (GRU) </a></br>
<a href="https://wandb.ai/renaudlesperance/INF8225%20-%20TP3%20-%20Final%20Run/reports/Transformer-Hyperparameter-Tuning--VmlldzoxODEyNDQ0" target="_blank" > Experiment 3 - Transformer s hyp
<a href="https://wandb.ai/morgiizz/INF8225%20-%20TP3%20-%20Transformer/reports/Transformer-MLP-tuning--VmlldzoxODA1NzUx" target="_blank" > Experiment 4 - Transformer MLP Tuning </a></br>
<a href="https://wandb.ai/morgiizz/INF8225%20-%20TP3%20-%20Transformer%20Greedy%20vs%20Beam/reports/Transformer-Greedy-vs-Beam-Search--VmlldzoxODA3NDAz" target="_blank" > Experiment 5 - Gree
<a href="https://wandb.ai/morgiizz/INF8225%20-%20TP3%20-%20Transformer%20Final%20Test/reports/Transformer-Final-test--VmlldzoxODA3NDM0" target="_blank" > Final Run : Best model (Transformer)
```

Experiment 1 - RNN vs GRU
Experiment 2 - MLP Tuning (GRU)
Experiment 3 - Transformer s hyperparameters
Experiment 4 - Transformer MLP Tuning
Experiment 5 - Greedy vs Beam Search and BLUE score
Final Run : Best model (Transformer)

Experiment 1 - RNN vs GRU

In [ ]:
```
%%html
<iframe src="https://wandb.ai/renaudlesperance/INF8225%20-%20TP3%20-%20Final%20Run/reports/RNN-vs-GRU--VmlldzoxODEwNjg5" style="border:none;height:1024px;width:100%">
```