TP3, INF8225 2025, Machine translation

Gervais Presley Koyaweda (2305686)

This TP will be due on March 27th at 8:30am. 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 (**Implementation provided!**)
- A GRU-RNN (done individually)
- A Transformer (The implementation, testing and experiments with an Encoder-Decoder - done individually, but you may discuss how to do this, ideas for experiments, etc. with any of your colleagues)

You are provided with the code to load and build the pytorch dataset, the implementation for the Vanilla RNN architecture and the code for the training loop. You "only" have to code the architectures (a GRU-RNN and a the missing parts of the Encoder-Decoder Transformer). Of course, the use of built-in torch layers such as `nn.GRU` or `nn.Transformer` is forbidden, as the TP would be much easier and you would learn much less.

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

We hope that this TP also provides you with a basic but realistic machine learning pipeline. We hope you learn a lot from the provided code.

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

```
!pip uninstall -y torchtext
!pip install torchtext==0.17.0
```

```
  Attempting uninstall: nvidia-cuda-nvrtc-cu12
    Found existing installation: nvidia-cuda-nvrtc-cu12 12.5.82
    Uninstalling nvidia-cuda-nvrtc-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-nvrtc-cu12-12.5.82
  Attempting uninstall: nvidia-cuda-cupti-cu12
    Found existing installation: nvidia-cuda-cupti-cu12 12.5.82
    Uninstalling nvidia-cuda-cupti-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-cupti-cu12-12.5.82
  Attempting uninstall: nvidia-cublas-cu12
    Found existing installation: nvidia-cublas-cu12 12.5.3.2
    Uninstalling nvidia-cublas-cu12-12.5.3.2:
      Successfully uninstalled nvidia-cublas-cu12-12.5.3.2
  Attempting uninstall: nvidia-cusolver-cu12
    Found existing installation: nvidia-cusolver-cu12 11.6.3.83
    Uninstalling nvidia-cusolver-cu12-11.6.3.83:
      Successfully uninstalled nvidia-cusolver-cu12-11.6.3.83
  Attempting uninstall: nvidia-cudnn-cu12
    Found existing installation: nvidia-cudnn-cu12 9.3.0.75
    Uninstalling nvidia-cudnn-cu12-9.3.0.75:
      Successfully uninstalled nvidia-cudnn-cu12-9.3.0.75
  Attempting uninstall: torch
    Found existing installation: torch 2.6.0+cu124
    Uninstalling torch-2.6.0+cu124:
      Successfully uninstalled torch-2.6.0+cu124
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the sourc
torchaudio 2.6.0+cu124 requires torch==2.6.0, but you have torch 2.2.0 which is incompatible.
torchvision 0.21.0+cu124 requires torch==2.6.0, but you have torch 2.2.0 which is incompatible.
Successfully installed nvidia-cublas-cu12-12.1.3.1 nvidia-cuda-cupti-cu12-12.1.105 nvidia-cuda-nvrtc-cu12-12.1.105 nvidia-cuda-runtime
```

```
# Note current default torch and cuda was 2.6.0+cu124
# We need to go back to an earlier version compatible with torchtext
# This will generate some dependency issues (incompatible packages), but for things that we will not need for this TP

#!pip install torch==2.1.2+cu121 -f https://download.pytorch.org/whl/torch/ --force-reinstall --no-cache-dir > /dev/null
#!pip install torchtext==0.16.2 --force-reinstall --no-cache-dir > /dev/null
#!pip install numpy==1.23.5 --force-reinstall --no-cache-dir > /dev/null


!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 .config import registry
  File "/usr/local/lib/python3.11/dist-packages/thinc/config.py", line 5, in <module>
    from .types import Decorator
  File "/usr/local/lib/python3.11/dist-packages/thinc/types.py", line 25, in <module>
    from .compat import cupy, has_cupy
  File "/usr/local/lib/python3.11/dist-packages/thinc/compat.py", line 35, in <module>
    import torch
  File "/usr/local/lib/python3.11/dist-packages/torch/__init__.py", line 1471, in <module>
    from .functional import *  # noqa: F403
  File "/usr/local/lib/python3.11/dist-packages/torch/functional.py", line 9, in <module>
    import torch.nn.functional as F
  File "/usr/local/lib/python3.11/dist-packages/torch/nn/__init__.py", line 1, in <module>
    from .modules import *  # noqa: F403
  File "/usr/local/lib/python3.11/dist-packages/torch/nn/modules/__init__.py", line 35, in <module>
    from .transformer import TransformerEncoder, TransformerDecoder, \
  File "/usr/local/lib/python3.11/dist-packages/torch/nn/modules/transformer.py", line 20, in <module>
    device: torch.device = torch.device(torch._C._get_default_device()),  # torch.device('cpu'),
    from thinc.util import copy_array
  File "/usr/local/lib/python3.11/dist-packages/thinc/__init__.py", line 5, in <module>
    from .config import registry
  File "/usr/local/lib/python3.11/dist-packages/thinc/config.py", line 5, in <module>
    from .types import Decorator
  File "/usr/local/lib/python3.11/dist-packages/thinc/types.py", line 25, in <module>
    from .compat import cupy, has_cupy
  File "/usr/local/lib/python3.11/dist-packages/thinc/compat.py", line 35, in <module>
    import torch
  File "/usr/local/lib/python3.11/dist-packages/torch/__init__.py", line 1471, in <module>
    from .functional import *  # noqa: F403
  File "/usr/local/lib/python3.11/dist-packages/torch/functional.py", line 9, in <module>
    import torch.nn.functional as F
  File "/usr/local/lib/python3.11/dist-packages/torch/nn/__init__.py", line 1, in <module>
    from .modules import *  # noqa: F403
  File "/usr/local/lib/python3.11/dist-packages/torch/nn/modules/__init__.py", line 35, in <module>
    from .transformer import TransformerEncoder, TransformerDecoder, \
  File "/usr/local/lib/python3.11/dist-packages/torch/nn/modules/transformer.py", line 20, in <module>
    device: torch.device = torch.device(torch._C._get_default_device()),  # torch.device('cpu'),
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/transformer.py:20: UserWarning: Failed to initialize NumPy: _ARRAY_API not fo
```

```python
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
# cpal
print(torch.__version__)

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

DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
downgrade to 'numpy<2' or try to upgrade the affected module.
We expect that some modules will need time to support NumPy 2.

Traceback (most recent call last):  File "<frozen runpy>", line 198, in _run_module_as_main
  File "<frozen runpy>", line 88, in _run_code
  File "/usr/local/lib/python3.11/dist-packages/colab_kernel_launcher.py", line 37, in <module>
    ColabKernelApp.launch_instance()
  File "/usr/local/lib/python3.11/dist-packages/traitlets/config/application.py", line 992, in launch_instance
    app.start()
  File "/usr/local/lib/python3.11/dist-packages/ipykernel/kernelapp.py", line 712, in start
    self.io_loop.start()
  File "/usr/local/lib/python3.11/dist-packages/tornado/platform/asyncio.py", line 205, in start
    self.asyncio_loop.run_forever()
  File "/usr/lib/python3.11/asyncio/base_events.py", line 608, in run_forever
    self._run_once()
  File "/usr/lib/python3.11/asyncio/base_events.py", line 1936, in _run_once
    handle._run()
  File "/usr/lib/python3.11/asyncio/events.py", line 84, in _run
    self._context.run(self._callback, *self._args)
```

```
      reply_content = await reply_content
    File "/usr/local/lib/python3.11/dist-packages/ipykernel/ipkernel.py", line 383, in do_execute
      res = shell.run_cell(
    File "/usr/local/lib/python3.11/dist-packages/ipykernel/zmqshell.py", line 528, in run_cell
      return super().run_cell(*args, **kwargs)
    File "/usr/local/lib/python3.11/dist-packages/IPython/core/interactiveshell.py", line 2975, in run_cell
      result = self._run_cell(
    File "/usr/local/lib/python3.11/dist-packages/IPython/core/interactiveshell.py", line 3030, in _run_cell
      return runner(coro)
    File "/usr/local/lib/python3.11/dist-packages/IPython/core/async_helpers.py", line 78, in _pseudo_sync_runner
      coro.send(None)
    File "/usr/local/lib/python3.11/dist-packages/IPython/core/interactiveshell.py", line 3257, in run_cell_async
      has_raised = await self.run_ast_nodes(code_ast.body, cell_name,
    File "/usr/local/lib/python3.11/dist-packages/IPython/core/interactiveshell.py", line 3473, in run_ast_nodes
      if (await self.run_code(code, result,  async_=asy)):
    File "/usr/local/lib/python3.11/dist-packages/IPython/core/interactiveshell.py", line 3553, in run_code
      exec(code_obj, self.user_global_ns, self.user_ns)
    File "<ipython-input-3-c2161d2c5b02>", line 8, in <cell line: 0>
      import torch
    File "/usr/local/lib/python3.11/dist-packages/torch/__init__.py", line 1471, in <module>
      from .functional import *  # noqa: F403
    File "/usr/local/lib/python3.11/dist-packages/torch/functional.py", line 9, in <module>
      import torch.nn.functional as F
    File "/usr/local/lib/python3.11/dist-packages/torch/nn/__init__.py", line 1, in <module>
      from .modules import *  # noqa: F403
    File "/usr/local/lib/python3.11/dist-packages/torch/nn/modules/__init__.py", line 35, in <module>
      from .transformer import TransformerEncoder, TransformerDecoder, \
    File "/usr/local/lib/python3.11/dist-packages/torch/nn/modules/transformer.py", line 20, in <module>
      device: torch.device = torch.device(torch._C._get_default_device()),  # torch.device('cpu'),
  /usr/local/lib/python3.11/dist-packages/torch/nn/modules/transformer.py:20: UserWarning: Failed to initialize NumPy: _ARRAY_API not fo
    device: torch.device = torch.device(torch._C._get_default_device()),  # torch.device('cpu'),
```

```python
# Our 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>']
```

```
--2025-03-27 00:57:51--  http://www.manythings.org/anki/fra-eng.zip
Resolving www.manythings.org (www.manythings.org)... 173.254.30.110
Connecting to www.manythings.org (www.manythings.org)|173.254.30.110|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7943074 (7.6M) [application/zip]
Saving to: ‘fra-eng.zip’

fra-eng.zip         100%[===================>]   7.57M  4.32MB/s    in 1.8s

2025-03-27 00:57:54 (4.32 MB/s) - ‘fra-eng.zip’ saved [7943074/7943074]

Archive:  fra-eng.zip
  inflating: _about.txt
  inflating: fra.txt
209462
/usr/local/lib/python3.11/dist-packages/torchtext/data/utils.py:105: UserWarning: Spacy model "en" could not be loaded, trying "en_core_
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/torchtext/data/utils.py:105: UserWarning: Spacy model "fr" could not be loaded, trying "fr_core_
  warnings.warn(
```

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

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.

```
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


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.

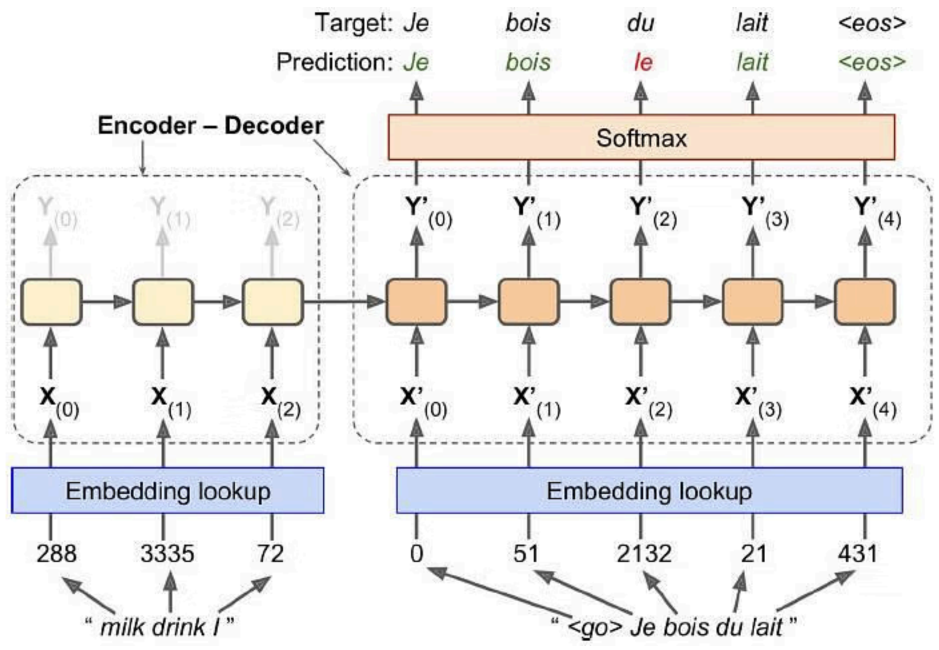
Formally, we define $s = [s_1, \dots, s_{N_s}]$ as the source sequence made of N_s tokens. We also define $t^i = [t_1, \dots, t_i]$ as the target sequence at the beginning of the step i.

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}.

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

Here, the implementation of the RNN is provided as an example. Study this code and use it as an example for the GRU implementation, if needed.

The `RNNCell` layer produce one hidden state vector for each sentence in the batch (useful for the output of the encoder), and also produce one embedding for each token in each sentence (useful for the output of the decoder).

The `RNN` module is composed of a stack of `RNNCell`. Each token embeddings coming out from a previous `RNNCell` is used as an input for the next `RNNCell` layer.

Be careful ! Our `RNNCell` implementation is not exactly the same thing as the PyTorch's `nn.RNNCell`. PyTorch implements only the operations for one token (so you would need to loop through each tokens inside the `RNN` instead).

The same thing apply for the `GRU` and `GRUCell`.

```
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.

    Important note: This layer does not exactly the same thing as nn.RNNCell does.
    PyTorch implementation is only doing one simple pass over one token for each batch.
    This implementation is taking the whole sequence of each batch and provide the
    final hidden state along with the embeddings of each token in each sequence.
    """
    def __init__(
            self,
            input_size: int,
            hidden_size: int,
            dropout: float,
        ):
        super().__init__()

        self.hidden_size = hidden_size

        # See pytorch definition: https://pytorch.org/docs/stable/generated/torch.nn.RNN.html
        self.Wih = nn.Linear(input_size, hidden_size, device=DEVICE)
        self.Whh = nn.Linear(hidden_size, hidden_size, device=DEVICE)
        self.dropout = nn.Dropout(p=dropout)
        self.act = nn.Tanh()

    def forward(self, x: torch.FloatTensor, h: torch.FloatTensor) -> tuple:
        """Go through all the sequence in x, iteratively updating
        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].
        """
        batch_size, seq_len, input_size = x.shape
        y = torch.zeros([batch_size, seq_len, self.hidden_size], device=DEVICE)

        for t in range(seq_len):
          input = x[:, t, :]
          w_input = self.Wih(input)
          w_hidden = self.Whh(h)
          h = self.act(w_input + w_hidden)
          y[:, t, :] = self.dropout(h)

        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.
            This parameter can be removed if you decide to use the module `GRU`.
            Indeed, `GRU` should have exactly the same code as this module,
            but with `GRUCell` instead of `RNNCell`. We let the freedom for you
            to decide at which level you want to specialise the modules (either
            in `TranslationRNN` by creating a `GRU` or a `RNN`, or in `RNN`
            by creating a `GRUCell` or a `RNNCell`).
    """
    def __init__(
            self,
            input_size: int,
            hidden_size: int,
            num_layers: int,
            dropout: float,
            model_type: str,
        ):
        super().__init__()

        self.hidden_size = hidden_size
        model_class = RNNCell if model_type == 'RNN' else GRUCell

        self.layers = nn.ModuleList()
        self.layers.append(model_class(input_size, hidden_size, dropout))
        for i in range(1, num_layers):
          self.layers.append(model_class(hidden_size, hidden_size, dropout))

    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].
        """
        input = x
        h = torch.zeros([x.shape[0], len(self.layers), self.hidden_size], device=x.device) if h is None else h
        final_h = torch.zeros_like(h, device=x.device)
        for l in range(len(self.layers)):
          input, h_out = self.layers[l](input, h[:, l, :])
          final_h[:, l, :] = h_out

        return input, final_h
```

GRU

Here you have to implement a GRU-RNN. This architecture is close to the Vanilla RNN but perform different operations. Look up the pytorch documentation to figure out the differences.

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

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

        # TODO

        # Stocage de num_layers
        self.num_layers = num_layers
        self.hidden_size = hidden_size

        self.gru=nn.GRU(input_size=input_size, # Dimension d'entrée (ex: embeddings de mots)
                        hidden_size=hidden_size, # Taille de l'état caché
                        num_layers=num_layers, # Nombre de couches GRU
                        dropout=dropout if num_layers>1 else 0, # Dropout seulement si plusieurs couches
                        batch_first=True # Permet d'avoir (batch_size, seq_len, input_size)n
                        )

    def forward(self, x: torch.FloatTensor, h: torch.FloatTensor=None) -> tuple:
        """
        Args
        ----
            x: Input sequence
                Shape of [batch_size, seq_len, input_size].
            h: Initial hidden state for each layer.
                If 'None', then an initial hidden state (a zero filled tensor)
                is created.
                Shape of [batch_size, n_layers, hidden_size].

        Output
        ------
            output:
                Shape of [batch_size, seq_len, hidden_size].
            h_n: Final hidden state.
                Shape of [batch_size, n_layers, hidden size].
        """
        # TODO
        if h is None:
            #h = torch.zeros(self.num_layers, x.size(0), self.hidden_size, device=x.device)
            h = torch.zeros(self.num_layers, batch_size, self.hidden_size, device=x.device)
        output, h_n = self.gru(x, h) # Passage dans la couche GRU

        return output, h_n
```

```
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__()
        # TODO

        self.hidden_size = hidden_size

        # Couches linéaires pour les portes du GRU

        # Porte de réinitialisation (Reset Gate)
        self.input_to_r = nn.Linear(input_size, hidden_size)
        self.hidden_to_r = nn.Linear(hidden_size, hidden_size)

        # Porte de mise à jour (Update Gate)
        #self.input_to_z=nn.Lenear(input_size, hidden_size)
        self.input_to_z = nn.Linear(input_size, hidden_size)
        self.hidden_to_z = nn.Linear(hidden_size, hidden_size)

        # Calcul du nouvel état candidat
        self.input_to_n=nn.Linear(input_size, hidden_size)
        self.hidden_to_n=nn.Linear(hidden_size, hidden_size)

        # Dropout pour éviter l'overfitting
        self.dropout = nn.Dropout(p=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].
        """
        # TODO

        """
        x = x.view(-1, x.shape[-1])  #Applatir x si nécessaire

        # Calcul de la porte de réinitialisation r_t
        r_t = torch.sigmoid(self.input_to_r(x) + self.hidden_to_r(h))  # Entre 0 et 1
        print(f"Porte Reset: {r_t}")

        # Calcul de la porte de mise à jour z_t
        z_t = torch.sigmoid(self.input_to_z(x) + self.hidden_to_z(h))  # Entre 0 et 1
        print(f"Porte de Update: {z_t}")

        # Calcul du nouvel état candidat
        n_t = torch.tanh(self.input_to_n(x) + r_t * self.hidden_to_n(h))
        print(f"Nouvel état candidat: {n_t}")

        # Calcul du nouvel état caché
        h_new = (1 - z_t) * n_t + z_t * h # Interpolation entre l'ancien et le nouvel état
        print(f"Nouvel état caché: {h_new}")
        return h_new, h_new

        """
        batch_size, seq_len, _ = x.shape
        y = torch.zeros(batch_size, seq_len, self.hidden_size, device=x.device)

        for t in range(seq_len):
            r_t = torch.sigmoid(self.input_to_r(x[:, t, :]) + self.hidden_to_r(h))
            z_t = torch.sigmoid(self.input_to_z(x[:, t, :]) + self.hidden_to_z(h))
            n_t = torch.tanh(self.input_to_n(x[:, t, :]) + r_t * self.hidden_to_n(h))

            h = (1 - z_t) * n_t + z_t * h
            y[:, t, :] = h

        return y, h


# Test
test_model = GRU(3, 40, 3, 0.5)
dict(test_model.named_parameters()).keys()
```

```
dict_keys(['gru.weight_ih_l0', 'gru.weight_hh_l0', 'gru.bias_ih_l0', 'gru.bias_hh_l0', 'gru.weight_ih_l1', 'gru.weight_hh_l1',
'gru.bias_ih_l1', 'gru.bias_hh_l1', 'gru.weight_ih_l2', 'gru.weight_hh_l2', 'gru.bias_ih_l2', 'gru.bias_hh_l2'])
```

Translation RNN

This module instanciates a vanilla RNN or a GRU-RNN and performs the translation task. This code des the following:

- Encodes the source and target sequence
- Passes the final hidden state of the encoder to the decoder (one for each layer)
- Decodes 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.

```
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.
    """

    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,
        ):
        super().__init__()
        self.src_embeddings = nn.Embedding(n_tokens_src, dim_embedding, src_pad_idx)
        self.tgt_embeddings = nn.Embedding(n_tokens_tgt, dim_embedding, tgt_pad_idx)

        self.dropout_1 = nn.Dropout(dropout)
        self.dropout_2 = nn.Dropout(dropout)

        self.encoder = RNN(dim_embedding, dim_hidden, n_layers, dropout, model_type)
        self.norm = nn.LayerNorm(dim_hidden)
        self.decoder = RNN(dim_embedding, dim_hidden, n_layers, dropout, model_type)
        self.out_layer = nn.Linear(dim_hidden, n_tokens_tgt)
    def forward(
        self,
        source: torch.LongTensor,
        target: torch.LongTensor
    ) -> torch.FloatTensor:
        """Predict the target tokens logits based on the source 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].
        """
        source = torch.fliplr(source)

        src_emb = self.src_embeddings(source)
        out, hidden = self.encoder(src_emb)

        hidden = self.norm(hidden)

        tgt_emb = self.tgt_embeddings(target)
        y, hidden = self.decoder(tgt_emb, hidden)

        y = self.out_layer(y)

        return y
```

Double-cliquez (ou appuyez sur Entrée) pour modifier

```
#Test
test_model = TranslationRNN(10000, 12000, 40, 30, 3, 0.5, 30, 30, 'RNN')
dict(test_model.named_parameters()).keys()
```

```
dict_keys(['src_embeddings.weight', 'tgt_embeddings.weight', 'encoder.layers.0.Wih.weight', 'encoder.layers.0.Wih.bias',
'encoder.layers.0.Whh.weight', 'encoder.layers.0.Whh.bias', 'encoder.layers.1.Wih.weight', 'encoder.layers.1.Wih.bias',
'encoder.layers.1.Whh.weight', 'encoder.layers.1.Whh.bias', 'encoder.layers.2.Wih.weight', 'encoder.layers.2.Wih.bias',
'encoder.layers.2.Whh.weight', 'encoder.layers.2.Whh.bias', 'norm.weight', 'norm.bias', 'decoder.layers.0.Wih.weight',
'decoder.layers.0.Wih.bias', 'decoder.layers.0.Whh.weight', 'decoder.layers.0.Whh.bias', 'decoder.layers.1.Wih.weight',
'decoder.layers.1.Wih.bias', 'decoder.layers.1.Whh.weight', 'decoder.layers.1.Whh.bias', 'decoder.layers.2.Wih.weight',
'decoder.layers.2.Wih.bias', 'decoder.layers.2.Whh.weight', 'decoder.layers.2.Whh.bias', 'out_layer.weight', 'out_layer.bias'])
```

Transformer models

Here you have to code the Full Transformer and Decoder-Only Transformer architectures. It is divided in three parts:

- Attention layers (done individually)
- Encoder and decoder layers (done individually)
- Full Transformer: gather the encoder and decoder layers (done individually)

The Transformer (or "Full Transformer") is presented in the paper: Attention is all you need. 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.

Positional Encoding

```
class PositionalEncoding(nn.Module):
    """
    This PE module comes from:
    Pytorch. (2021). LANGUAGE MODELING WITH NN.TRANSFORMER AND TORCHTEXT. https://pytorch.org/tutorials/beginner/transformer_tutorial.html
    """
    def __init__(self, d_model: int, dropout: float, max_len: int = 5000):
        super().__init__()
        self.dropout = nn.Dropout(dropout)

        position = torch.arange(max_len).unsqueeze(1).to(DEVICE)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model)).to(DEVICE)
        pe = torch.zeros(max_len, 1, d_model).to(DEVICE)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = rearrange(x, "b s e -> s b e")
        """
        Args:
            x: Tensor, shape [seq_len, batch_size, embedding_dim]
        """
        x = x + self.pe[:x.size(0)]
        x = rearrange(x, "s b e -> b s e")
        return self.dropout(x)
```

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*.

```
from einops import rearrange
import math


from pickle import NONE
from einops.layers.torch import Rearrange


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 between the keys and the queries.
                Shape of [batch_size, n_heads, seq_len_1, seq_len_2].
        """
        # TODO
        d_k = q.shape[-1]  # Taille de chaque tête d'attention

        # Calcul du score d'attention : q @ k^T / sqrt(d_k)
        attn_scores = torch.einsum("bqhd,bkhd->bhqk", q, k) / math.sqrt(d_k)

        # Application du masque (si fourni)
        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask.bool(), float('-inf'))

        # Normalisation avec softmax (convertir les scores en probabilités)
        attn_weights = torch.softmax(attn_scores, dim=-1)

        # Application du dropout (si fourni)
        if dropout is not None:
            attn_weights = dropout(attn_weights)

        # Pondération des valeurs `v` par les poids d'attention
        y = torch.einsum("bhqk,bkhd->bqhd", attn_weights, v)

        return y, attn_weights

    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

            # TODO
            self.dim=dim
            self.n_heads=n_heads
            self.dim_head = dim // n_heads # Taille de chaque tête

            # Projections linéaires pour Query, Key, Value
            self.w_q = nn.Linear(dim, dim, bias=False)
            self.w_k = nn.Linear(dim, dim, bias=False)
            self.w_v = nn.Linear(dim, dim, bias=False)

            # Projection linéaire après concaténation des têtes
            self.w_out=nn.Linear(dim, dim, bias=False)

            # 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

            #  Projetion des entrées en Query, Key, Value
            q=self.w_q(q)
            k=self.w_k(k)
            v=self.w_v(v)

            #  Réarrangement pour obtenir (batch, seq_len, n_heads, dim_head)
            q, k, v = map(lambda x: rearrange(x, "b s (h d) -> b s h d", h=self.n_heads), [q, k, v])

            #  Application de l'attention
            y, attn_weights=attention(q,k,v,mask=attn_mask,dropout=self.dropout)

            #  Réorganiser la sortie en (batch, seq_len, dim)
            y= rearrange(y, "b s h d -> b s (h d)")

            #  Application de la projection finale
            y = self.w_out(y)

            return y



    #Test

    # Paramètres
    batch_size = 2
    seq_len = 5
    dim_model = 32
    n_heads = 4
    dropout = 0.1

    # Création de tenseurs aléatoires
    q = torch.randn(batch_size, seq_len, dim_model)
    k = torch.randn(batch_size, seq_len, dim_model)
    v = torch.randn(batch_size, seq_len, dim_model)

    # Initialisation du modèle
    mha = MultiheadAttention(dim_model, n_heads, dropout)

    # Exécution du modèle
    output = mha(q, k, v)
```

```
print("\n=== Sortie du MultiheadAttention ===")
print("Shape:", output.shape)
```

```
=== Sortie du MultiheadAttention ===
Shape: torch.Size([2, 5, 32])
```

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.

```
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
        ):
        super().__init__()

        # TODO
        # Masked Self-Attention (Empêche l'attention sur les futurs tokens)
        self.self_attn=nn.MultiheadAttention(d_model, nhead, dropout=dropout, batch_first=True)

        # Cross-Attention (Regarde la sortie de l'encodeur)
        self.cross_attn=nn.MultiheadAttention(d_model, nhead, dropout=dropout, batch_first=True)

        # Feed-Forward Network
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model)
        )

        # Normalisation & Dropout
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)


    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:
        """Decode the next target tokens based on the previous tokens.

        Args
        ----
            src: Batch of source sentences.
                Shape of [batch_size, src_seq_len, dim_model].
            tgt: Batch of target sentences.
                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:  Batch of sequence of embeddings representing the predicted target tokens
                Shape of [batch_size, tgt_seq_len, dim_model].
        """
        # TODO

        # Masked Self-Attention
        tgt2, _ = self.self_attn(tgt, tgt, tgt, attn_mask=tgt_mask_attn, key_padding_mask=tgt_key_padding_mask)
        tgt = self.norm1(tgt + self.dropout(tgt2))

        #  Cross-Attention avec la sortie de l'encodeur
        tgt2, _ = self.cross_attn(tgt, src, src, key_padding_mask=src_key_padding_mask)
        tgt = self.norm2(tgt + self.dropout(tgt2))

        # Feed-Forward Network
        tgt2 = self.ffn(tgt)
        tgt = self.norm3(tgt + self.dropout(tgt2))

        return tgt


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
        ):
        super().__init__()

        # TODO

        self.layers=nn.ModuleList(
            [
                TransformerDecoderLayer(d_model,d_ff, nhead, dropout)
                for _ in range(num_decoder_layer)
            ]
        )

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

        Args
        ----
            src: Batch of encoded source sentences.
                Shape of [batch_size, src_seq_len, dim_model].
            tgt: Batch of taget sentences.
                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:  Batch of sequence of embeddings representing the predicted target tokens
                Shape of [batch_size, tgt_seq_len, dim_model].
        """
        # TODO
        for layer in self.layers:
            tgt=layer(src, tgt, tgt_mask_attn, src_key_padding_mask, tgt_key_padding_mask)
        return tgt


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,
        ):
        super().__init__()

        # TODO
        self.self_attn = nn.MultiheadAttention(d_model, nhead, dropout=dropout, batch_first=True)

        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model)
        )

        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = 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
        src2, _ = self.self_attn(src, src, src, key_padding_mask=key_padding_mask)
        src = self.norm1(src + self.dropout(src2))

        src2 = self.ffn(src)
        src = self.norm2(src + self.dropout(src2))
        return src

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
        ):
        super().__init__()

        # TODO
        self.layers = nn.ModuleList([
            TransformerEncoderLayer(d_model, dim_feedforward, nheads, dropout)
            for _ in range(num_encoder_layers)
        ])

    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
        for layer in self.layers:
            src = layer(src, key_padding_mask)
        return src
```

Transformer

This section gathers 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.*

```
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,
        ):
        super().__init__()
        # TODO

        # Encodeur Transformer
        self.encoder=TransformerEncoder(
            d_model,
            dim_feedforward,
            num_decoder_layers,
            nhead,
            dropout
        )

        # Décodeur Transformer
        self.decoder=TransformerDecoder(
            d_model,
            dim_feedforward,
            num_decoder_layers,
            nhead,
            dropout
        )

    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

        # Encode la séquence source
        memory=self.encoder(src,
                            src_key_padding_mask)

        # Décode la séquence cible
        output=self.decoder(memory,
                            tgt,
                            tgt_mask_attn,
                            src_key_padding_mask,
                            tgt_key_padding_mask
                            )
        return output


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,
        ):
        super().__init__()

        # TODO

        self.src_pad_idx=src_pad_idx
        self.tgt_pad_idx=tgt_pad_idx

        # Embeddings pour la source et la cible
        self.src_embeddings=nn.Embedding(n_tokens_src, dim_embedding, padding_idx=src_pad_idx)
        self.tgt_embeddings=nn.Embedding(n_tokens_tgt, dim_embedding,padding_idx=tgt_pad_idx)

        # Transformer
        self.transformer=Transformer(
            d_model=dim_embedding,
            nhead=n_heads,
            num_encoder_layers=n_layers,
            num_decoder_layers=n_layers,
            dim_feedforward=dim_hidden,
            dropout=dropout
        )

        # Projette la sortie du Transformer en logits sur le vocabulaire cible
        self.out_layer=nn.Linear(dim_embedding,n_tokens_tgt)

    def forward(
            self,
            source: torch.LongTensor,
            target: torch.LongTensor
        ) -> torch.FloatTensor:
        """Predict the target tokens logites 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: 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, seq_len_tgt, n_tokens_tgt].
        """
        # TODO
        # Conversion en embeddings
        src_emb=self.src_embeddings(source)
        tgt_emb=self.tgt_embeddings(target)

        # Création des masques
        tgt_mask_attn=self.generate_causal_mask(target)
        src_key_padding_mask, tgt_key_padding_mask=self.generate_key_padding_mask(source, target)

        # Passage dans le Transformer
        y = self.transformer(src_emb, tgt_emb, tgt_mask_attn, src_key_padding_mask, tgt_key_padding_mask)

        # Projette la sortie en logits sur le vocabulaire cible
        y = self.out_layer(y)

        return y

    def generate_causal_mask(
            self,
            target: torch.LongTensor,
        ) -> tuple:
        """Generate the masks to prevent attending subsequent 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
        ------
            tgt_mask_attn: Mask to prevent attention to subsequent tokens.
                Shape of [seq_len_tgt, seq_len_tgt].

        """

        seq_len = target.shape[1]

        tgt_mask = torch.ones((seq_len, seq_len), dtype=torch.bool)
        tgt_mask = torch.triu(tgt_mask, diagonal=1).to(target.device)

        return tgt_mask

    def generate_key_padding_mask(
            self,
            source: torch.LongTensor,
            target: torch.LongTensor,
        ) -> tuple:
        """Generate the masks to prevent attending padding 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
        ------
            src_key_padding_mask: Mask to prevent attention to padding in src sequence.
                Shape of [batch_size, seq_len_src].
            tgt_key_padding_mask: Mask to prevent attention to padding in tgt sequence.
                Shape of [batch_size, seq_len_tgt].

        """

        src_key_padding_mask = source == self.src_pad_idx
        tgt_key_padding_mask = target == self.tgt_pad_idx

        return src_key_padding_mask, tgt_key_padding_mask
```

```
# Test

# Paramètres
batch_size = 2
seq_len_src = 6
seq_len_tgt = 6
n_tokens_src = 1000
n_tokens_tgt = 1200
dim_embedding = 128
dim_hidden = 256
n_layers = 4
n_heads = 8
dropout = 0.1
src_pad_idx = 0
tgt_pad_idx = 0

# Création de tokens aléatoires
source = torch.randint(0, n_tokens_src, (batch_size, seq_len_src))
target = torch.randint(0, n_tokens_tgt, (batch_size, seq_len_tgt))

# Initialisation du modèle
model = TranslationTransformer(n_tokens_src, n_tokens_tgt, n_heads, dim_embedding, dim_hidden, n_layers, dropout, src_pad_idx, tgt_pad_idx)

# Passage avant
logits = model(source, target)
print("Shape des logits:", logits.shape)  # [batch_size, seq_len_tgt, n_tokens_tgt]
```

```
Shape des logits: torch.Size([2, 6, 1200])
```

Greedy search

One idea to explore once you have your model working is 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. Compare this strategy of decoding with the beam search strategy below.

```
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_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
    ------
        sentence: The translated source sentence.
    """

    # Tokenisation et conversion en IDs
    tokens=src_tokenizer(source)
    indexed= [src_vocab[token] for token in tokens]
    source_tensor=torch.LongTensor(indexed).unsqueeze(0).to(device)

    # Passage dans le modèle (Encode la source)
    with torch.no_grad():
      encoded_source=model.src_embeddings(source_tensor)
      src_mask=model.generate_key_padding_mask(source_tensor,source_tensor)[0]
      memory=model.transformer.encoder(encoded_source, src_mask)
    # Début de la génération (On commence avec `<bos>`)
    tgt_tokens=[tgt_vocab["<bos>"]]

    for _ in range (max_sentence_length):
      # Convertion des tokens en tenseur
      tgt_tensor=torch.LongTensor(tgt_tokens).unsqueeze(0).to(device)

      # Embeddings et masques
      tgt_emb=model.tgt_embeddings(tgt_tensor)
      tgt_mask=model.generate_causal_mask(tgt_tensor)

      # Passage dans le décodeur
      with torch.no_grad():
          output = model.transformer.decoder(memory, tgt_emb, tgt_mask, src_mask, None)

        # Prédiction du prochain mot
      logits = model.out_layer(output[:, -1, :])  # Dernier token généré
      next_token = torch.argmax(F.softmax(logits, dim=-1), dim=-1).item()

      # Arrêt si `<eos>` est atteint
      if next_token == tgt_vocab["<eos>"]:
          break

      # Ajouter le token prédicté à la séquence cible
      tgt_tokens.append(next_token)

    # Conversion des tokens en phrase
    translated_sentence = " ".join([tgt_vocab.lookup_token(token) for token in tgt_tokens[1:]])  # Exclure `<bos>`

    return translated_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 chooses 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


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


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.

```
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


    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,)) ]
            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]
            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,)) ]
            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]

            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`)!

```
# Checking GPU and logging to wandb

!wandb login

!nvidia-smi
```

```
wandb: Logging into wandb.ai. (Learn how to deploy a W&B server locally: https://wandb.me/wandb-server)
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: No netrc file found, creating one.
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
wandb: Currently logged in as: gpresleyk (gpresleyk-polytechnique-montr-al) to https://api.wandb.ai. Use `wandb login --relogin` to forc
Thu Mar 27 00:59:05 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| 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 T4                       Off |   00000000:00:04.0 Off |                    0 |
| N/A   55C    P0             30W /   70W |     106MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
+-----------------------------------------------------------------------------------------+
```

```
# Instanciate the datasets

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: 12,154
French vocabulary size: 18,340

Training examples: 209,459
Validation examples: 23,274
```

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

```
config = {
    # General parameters
    'epochs': 20,
    'batch_size': 128,
    'lr': 1e-3,
    'betas': (0.9, 0.99),
    'clip': 5,
    'device': 'cuda' if torch.cuda.is_available() else 'cpu',

    # Model parameters
    'n_tokens_src': len(train_dataset.en_vocab),
    'n_tokens_tgt': len(train_dataset.fr_vocab),
    'n_heads': 4,
    'dim_embedding': 196,
    'dim_hidden': 256,
    'n_layers': 3,
    'dropout': 0.1,
    'model_type': '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'])
)

# Uncomment code block to select model to train here!

"""
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'],
)
"""
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['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,
)
```

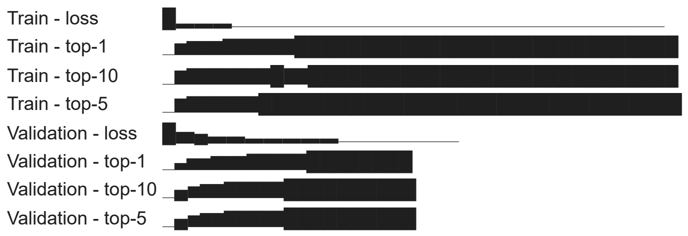
```
====================================================================================================
Layer (type:depth-idx)                             Output Shape              Param #
====================================================================================================
TranslationTransformer                             [128, 60, 18340]          --
├─Embedding: 1-1                                   [128, 60, 196]            2,382,184
├─Embedding: 1-2                                   [128, 60, 196]            3,594,640
├─Transformer: 1-3                                 [128, 60, 196]            --
│    └─TransformerEncoder: 2-1                     [128, 60, 196]            --
│    │    └─ModuleList: 3-1                        --                        768,108
│    └─TransformerDecoder: 2-2                     [128, 60, 196]            --
│    │    └─ModuleList: 3-2                        --                        1,232,628
├─Linear: 1-4                                      [128, 60, 18340]          3,612,980
====================================================================================================
Total params: 11,590,540
Trainable params: 11,590,540
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 1.31
====================================================================================================
Input size (MB): 0.12
Forward/backward pass size (MB): 1498.15
Params size (MB): 40.80
Estimated Total Size (MB): 1539.08
====================================================================================================
```

```
!wandb online  # online / offline / disabled to activate, deactivate or turn off WandB logging

with wandb.init(
        config=config,
        project='INF8225 - TP3',  # Title of your project
        group='Transformer_3_layers',  # 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.
wandb: Using wandb-core as the SDK backend.  Please refer to https://wandb.me/wandb-core for more information.
wandb: Currently logged in as: gpresleyk (gpresleyk-polytechnique-montr-al) to https://api.wandb.ai. Use `wandb login --relogin` to forc
Tracking run with wandb version 0.19.8
Run data is saved locally in /content/wandb/run-20250327_005941-6wyy6te9
Syncing run legendary-darkness-29 to Weights & Biases (docs)
View project at https://wandb.ai/gpresleyk-polytechnique-montr-al/INF8225%20-%20TP3
View run at https://wandb.ai/gpresleyk-polytechnique-montr-al/INF8225%20-%20TP3/runs/6wyy6te9
Starting training for 20 epochs, using cuda.

Epoch 1
Train -    loss: 1.85     top-1: 0.63    top-5: 0.83    top-10: 0.87
Eval -     loss: 1.66     top-1: 0.66    top-5: 0.84    top-10: 0.88
I don't pay attention to gossip.
Je n'ai pas attention à payer.

Epoch 2
Train -    loss: 1.46     top-1: 0.68    top-5: 0.87    top-10: 0.91
Eval -     loss: 1.39     top-1: 0.70    top-5: 0.88    top-10: 0.91
I want to be clear about that.
Je veux être clair à ce sujet.

Epoch 3
Train -    loss: 1.35     top-1: 0.70    top-5: 0.89    top-10: 0.92
Eval -     loss: 1.27     top-1: 0.71    top-5: 0.89    top-10: 0.92
Ah, to be young again.
Je serai jeune, à nouveau jeune.

Epoch 4
Train -    loss: 1.25     top-1: 0.72    top-5: 0.90    top-10: 0.93
Eval -     loss: 1.21     top-1: 0.72    top-5: 0.90    top-10: 0.93
Don't call me so late at night.
Ne m'appelez pas si tard la nuit.

Epoch 5
Train -    loss: 1.16     top-1: 0.73    top-5: 0.91    top-10: 0.94
Eval -     loss: 1.15     top-1: 0.74    top-5: 0.91    top-10: 0.93
It'll soon be time for dinner.
Ce sera bientôt pour le dîner.

Epoch 6
Train -    loss: 1.12     top-1: 0.73    top-5: 0.91    top-10: 0.94
Eval -     loss: 1.12     top-1: 0.74    top-5: 0.91    top-10: 0.94
Even I don't know that.
Même je ne sais pas ça.

Epoch 7
Train -    loss: 1.06     top-1: 0.75    top-5: 0.92    top-10: 0.95
Eval -     loss: 1.11     top-1: 0.74    top-5: 0.91    top-10: 0.94
I want more detailed information.
Je veux davantage d'information.

Epoch 8
Train -    loss: 1.03     top-1: 0.75    top-5: 0.92    top-10: 0.95
Eval -     loss: 1.08     top-1: 0.75    top-5: 0.92    top-10: 0.94
The game will be held even if it rains.
Le match fera s'il pleuvra.

Epoch 9
Train -    loss: 1.00     top-1: 0.76    top-5: 0.93    top-10: 0.95
Eval -     loss: 1.06     top-1: 0.75    top-5: 0.92    top-10: 0.94
Choose your favorite racket.
Choisis ta raquette.

Epoch 10
Train -    loss: 0.97     top-1: 0.76    top-5: 0.93    top-10: 0.95
Eval -     loss: 1.06     top-1: 0.75    top-5: 0.92    top-10: 0.94
One of my suitcases is missing.
L'un de mes valises est manquant.

Epoch 11
Train -    loss: 0.93     top-1: 0.77    top-5: 0.93    top-10: 0.96
Eval -     loss: 1.05     top-1: 0.76    top-5: 0.92    top-10: 0.94
Are you jealous?
Es-tu jaloux ?

Epoch 12
Train -    loss: 0.94     top-1: 0.77    top-5: 0.93    top-10: 0.96
Eval -     loss: 1.04     top-1: 0.76    top-5: 0.92    top-10: 0.94
I don't have time to finish my homework.
Je n'ai pas le temps de finir mes devoirs.

Epoch 13
Train -    loss: 0.90     top-1: 0.77    top-5: 0.94    top-10: 0.96
```

```
Eval -    loss: 1.03     top-1: 0.76    top-5: 0.92    top-10: 0.95
I've never been to a professional baseball game.
Je n'ai jamais été un match de base-ball professionnel.

Epoch 14
Train -   loss: 0.88     top-1: 0.78    top-5: 0.94    top-10: 0.96
Eval -    loss: 1.03     top-1: 0.76    top-5: 0.92    top-10: 0.95
I never told you to quit.
Je ne t'ai jamais dit de démissionner.

Epoch 15
Train -   loss: 0.86     top-1: 0.78    top-5: 0.94    top-10: 0.96
Eval -    loss: 1.03     top-1: 0.76    top-5: 0.92    top-10: 0.95
They deserve more.
Ils méritent davantage.

Epoch 16
Train -   loss: 0.88     top-1: 0.78    top-5: 0.94    top-10: 0.96
Eval -    loss: 1.02     top-1: 0.76    top-5: 0.92    top-10: 0.95
I don't recommend eating in that restaurant. The food is awful.
Le restaurant que je ne recommande pas de nourriture dans ce restaurant.

Epoch 17
Train -   loss: 0.84     top-1: 0.78    top-5: 0.94    top-10: 0.96
Eval -    loss: 1.01     top-1: 0.77    top-5: 0.92    top-10: 0.95
Tom hates Halloween.
Tom déteste Halloween.

Epoch 18
Train -   loss: 0.82     top-1: 0.79    top-5: 0.94    top-10: 0.96
Eval -    loss: 1.01     top-1: 0.77    top-5: 0.93    top-10: 0.95
Figure up how much it amounts to.
Arrête de monter la main.

Epoch 19
Train -   loss: 0.83     top-1: 0.79    top-5: 0.94    top-10: 0.96
Eval -    loss: 1.01     top-1: 0.77    top-5: 0.92    top-10: 0.95
This building belongs to my family.
Ce bâtiment appartient à ma famille.

Epoch 20
Train -   loss: 0.82     top-1: 0.79    top-5: 0.94    top-10: 0.96
Eval -    loss: 1.02     top-1: 0.77    top-5: 0.93    top-10: 0.95
Tom doesn't know anything.
Tom ne sait rien.
```

Run history:



Run summary:

Train - loss	0.81961
Train - top-1	0.78857
Train - top-10	0.96385
Train - top-5	0.94379
Validation - loss	1.01656
Validation - top-1	0.76679
Validation - top-10	0.94783
Validation - top-5	0.925

View run **legendary-darkness-29** at: https://wandb.ai/gpresleyk-polytechnique-montr-al/INF8225%20-%20TP3/runs/6wyy6te9
View project at: https://wandb.ai/gpresleyk-polytechnique-montr-al/INF8225%20-%20TP3
Synced 5 W&B file(s), 1 media file(s), 5 artifact file(s) and 0 other file(s)
Find logs at: ./wandb/run-20250327_005941-6wyy6te9/logs

Comparaison Greedy et Beam Search

```
sentence = "It is possible to try your work here."

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']
)[:5]

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

```
0. (19.92365%)   Il est possible d'essayer ton travail ici.
1. (14.13508%)   C'est possible d'essayer ton travail ici.
2. (12.76833%)   Il est possible d'essayer de travailler ici.
3. (7.20863%)    C'est possible d'essayer de travailler ici.
4. (3.70763%)    Il est possible d'essayer votre travail ici.
```

Commencez à coder ou à générer avec l'IA.

Grading:

Implementations (50 points total)

10 Points for your implementation of the GRU

40 Points for your implementaiton of the Transformer components

Questions (12 points, 1 point each)

1. Explain the differences between Vanilla RNN, GRU-RNN, Encoder-Decoder Transformer and Decoder-Only Transformer.
2. Why is positionnal encoding necessary in Transformers and not in RNNs?
3. Describe the preprocessing process. Detail how the initial dataset is processed before being fed to the translation models.
4. What is teacher forcing, and how is it used in Transformer training? How does the decoder input differ?
5. How are the two types of mask important to the attention mechanism (causal and padding) and how do they work? How do they differ between the encoder and decoder?
6. What is a causal mask, and why is it only used in the decoder?
7. Why does the decoder use both self-attention and encoder-decoder attention?
8. Why is the Transformer model parallelizable, and how does this improve efficiency compared to RNNs?
9. How does multi-head self-attention allow the model to capture different aspects of a sentence?
10. What does the decoder's final output represent before the projection layer? What does the encoder's final output represent?
11. What is the role of the final linear projection layer in the decoder? How does the decoder output differ between training (parallel processing) and inference (sequential generation)?
12. Why does the decoder recompute all outputs at each inference step instead of appending new outputs incrementally?

Réponses aux 12 Questions

---

1. Explain the differences between Vanilla RNN, GRU-RNN, Encoder-Decoder Transformer and Decoder-Only Transformer.

- **Vanilla RNN** : Réseau récurrent simple. Il traite les tokens séquentiellement avec une mémoire limitée, mais il souffre fortement du problème de disparition du gradient.

- **GRU-RNN** : Variante de RNN intégrant des portes (`update` et `reset`) permettant de mieux gérer les dépendances à long terme. Il est plus performant que le Vanilla RNN tout en étant plus léger que le LSTM.
- **Transformer Encodeur-Décodeur** : Architecture fondée sur l'attention. L’encodeur transforme la phrase source en représentations contextuelles, et le décodeur les utilise pour générer la traduction. Il permet un traitement parallèle grâce à l’attention.
- **Transformer Décodeur-Seul** : N’utilise que la partie décodeur avec masquage causal. Utilisé notamment dans la génération de texte (ex. GPT), il ne nécessite pas d’encodeur.

---

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

Les Transformers traitent tous les tokens en parallèle, sans ordre séquentiel implicite. Le **positional encoding** ajoute donc explicitement l’information de position.
Les RNN, quant à eux, traitent les tokens un par un : la position est naturellement encodée par la récursivité.

---

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

- Nettoyage des phrases (suppression des caractères spéciaux, filtrage de la longueur).
- **Tokenisation** en utilisant `spacy` .
- Ajout de tokens spéciaux `<bos>` et `<eos>` .
- Construction du vocabulaire avec un seuil de fréquence minimal (`min_freq`).
- Encodage des tokens en indices pour former les batchs de données.

---

4. What is teacher forcing, and how is it used in Transformer training? How does the decoder input differ?

Le **teacher forcing** consiste à fournir, à chaque étape d’apprentissage, le **vrai mot précédent** au lieu de celui généré par le modèle.
Cela permet un apprentissage plus stable et rapide.

- **En entraînement** : on fournit la séquence cible décalée.
- **En inférence** : on génère les mots **un à un** de manière séquentielle.

---

5. How are the two types of mask important to the attention mechanism (causal and padding) and how do they work? How do they differ between the encoder and decoder?

- **Padding mask** : masque les tokens `<pad>` pour qu’ils ne soient pas pris en compte dans l’attention.
- **Causal mask** : empêche un token de voir les tokens **futurs**, essentiel dans le décodeur.

Encodeur : uniquement le padding mask.
Décodeur : les deux masques sont nécessaires.

---

6. What is a causal mask, and why is it only used in the decoder?

Le **causal mask** est une matrice triangulaire qui bloque les tokens futurs. Il est utilisé dans le décodeur pour éviter que le modèle "triche" en voyant la suite de la phrase lors de la génération.

---

7. Why does the decoder use both self-attention and encoder-decoder attention?

- La **self-attention** permet au décodeur de contextualiser les mots déjà générés.
- L’**encoder-decoder attention** connecte les mots générés avec les représentations de la phrase source.

Ces deux mécanismes sont complémentaires pour une traduction précise.

---

8. Why is the Transformer model parallelizable, and how does this improve efficiency compared to RNNs?

Grâce à l’attention, le Transformer traite **tous les tokens simultanément** (en parallèle), contrairement aux RNNs qui traitent les tokens **séquentiellement**.
Cela permet un **gros gain de vitesse**, surtout sur GPU.

---

9. How does multi-head self-attention allow the model to capture different aspects of a sentence?

Chaque tête d'attention se concentre sur des **relations différentes** (sujet-verbe, dépendances lointaines, etc.).
Le **multi-head** permet donc de capturer plusieurs niveaux de sens en parallèle.

---

10. What does the decoder's final output represent before the projection layer? What does the encoder's final output represent?

- **Encodeur** : vecteurs représentant chaque mot de la phrase source avec son contexte global.
- **Décodeur** : représentations des mots déjà générés, combinées avec le contexte source.

Ces sorties sont projetées sur le vocabulaire via une couche linéaire.

---

11. What is the role of the final linear projection layer in the decoder? How does the decoder output differ between training (parallel processing) and inference (sequential generation)?

- La couche linéaire **projette** les vecteurs sur l'espace du vocabulaire (dimension `hidden → vocab_size`) pour prédire le mot suivant.
- **En entraînement** : on connaît la séquence cible → traitement **parallèle**.
- **En inférence** : on génère un mot à la fois → traitement **séquentiel**.

---

12. Why does the decoder recompute all outputs at each inference step instead of appending new outputs incrementally?

Contrairement aux RNNs, le Transformer **n'a pas de mémoire d'état**.
Il utilise toute la séquence générée à chaque étape pour recalculer les attentions. Cela assure une meilleure précision, mais augmente le **coût computationnel**.

Small report - experiments of your own choice (15 points)

Once everything is working fine, you can explore aspects of these models and do some research of your own into how they behave.

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? If you decide to implement Greedy search to compare with beam search, how much worse is it ?

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 accounts for many points, *feel free to go wild!*

Make a concise report about your experiments here.

Court Rapport – Expérimentations personnelles (TP3)

Ce court rapport présente les grandes lignes des expérimentations que j'ai menées dans le cadre du TP3 sur la traduction automatique EN-FR. Il s'appuie sur une analyse approfondie disponible dans mon rapport final.

Architectures évaluées

- **Vanilla RNN** (1, 2, 3 couches)
- **GRU-RNN** (1, 2, 3 couches)
- **Transformer** (1, 2, 3 couches)

Ajustement des hyperparamètres

- Test du nombre de **couches empilées** sur chaque architecture.
- Analyse de l'**impact sur la performance** (Loss, Top-k accuracy) et le **temps d'entraînement**.
- Utilisation de **20 époques** pour améliorer la convergence (vs 5 par défaut).

Comparaison des méthodes de décodage

- Comparaison qualitative et quantitative des traductions générées.

Autres aspects étudiés

- **Analyse du temps d'exécution** selon l’architecture et le nombre de couches.
- **Visualisation des courbes d’apprentissage** (loss, top-k) via WandB.
- Interprétation détaillée des **tableaux de performances** (entraînement et validation).

ℹ️ **NB :** Ce résumé donne juste un aperçu synthétique. Tous les détails (graphiques, interprétations, explications techniques) sont présentés dans mon **rapport complet**.