A

Transformers are better at handling inputs where context is need to understand what to output

B

The purpose of the encoder is to process the input sequence. it encode the sentances "le chat est noir"

C

The purpose of the decoder is the process the output sequence

D

If there are no positional enmbeddings the sentance/input sequence loses all meaning and the model will be bad.
The encodings are [00 42 82 16 04]

E

i) The output sequence shape will be (B, Lq, Lk)
ii) The attention weights shape will be (B,H,Lq,Lk)
iii) No

F

G

In [1]:
```
!pip install tqdm
```

```
Requirement already satisfied: tqdm in /Library/Frameworks/Python.framework/
Versions/3.12/lib/python3.12/site-packages (4.66.4)
```

In [ ]:
```
# For tips on running notebooks in Google Colab, see
# https://pytorch.org/tutorials/beginner/colab
%matplotlib inline
```

NLP From Scratch: Translation with a Transformer Network

Tutorial adapted from NLP From Scratch with PyTorch by Sean Robertson, and Tensorflow Tutorial on Transformers

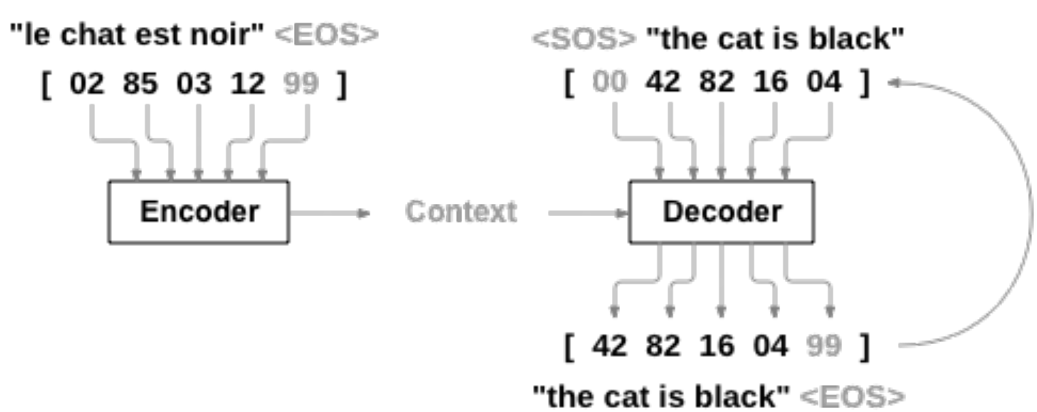
In this tutorial, we will write our own classes and functions to preprocess the data to do our NLP modeling tasks. We hope after you complete this tutorial that you'll proceed to learn how `torchtext` can handle much of this preprocessing for you.

In this project we will be teaching a neural network to translate from French to English.

```
[KEY: > input, = target, < output]

> il est en train de peindre un tableau .
= he is painting a picture .
< he is painting a picture .

> pourquoi ne pas essayer ce vin delicieux ?
= why not try that delicious wine ?
< why not try that delicious wine ?

> elle n est pas poete mais romanciere .
= she is not a poet but a novelist .
< she not not a poet but a novelist .

> vous etes trop maigre .
= you re too skinny .
< you re all alone .
```

... to varying degrees of success.

This is made possible by the simple but powerful idea of the sequence to sequence network_, in which two neural networks work together to transform one sequence to another. An encoder network condenses an input sequence into a vector, and a decoder network unfolds that vector into a new sequence.



To improve upon this model we'll use an attention mechanism_, which lets the decoder learn to focus over a specific range of the input sequence.

Recommended Reading on Sequence to Sequence networks:

- Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation_
- Sequence to Sequence Learning with Neural Networks_
- Neural Machine Translation by Jointly Learning to Align and Translate_
- A Neural Conversational Model_

Import Requirements

In [2]:
```
from __future__ import unicode_literals, print_function, division
from io import open
import unicodedata
import re
import random
import numpy as np
import tqdm
import torch
import torch.nn as nn
from torch import optim

from torch.utils.data import TensorDataset, DataLoader, RandomSampler

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

Loading and preparing the data

The data for this project is a set of many thousands of English to French translation pairs. Let's start by downloading the data to `data/eng-fra.txt`. This file is a tab separated list of translation pairs:

```
I am cold.    J'ai froid.
```

In [3]:
```
import requests
from zipfile import ZipFile
from io import BytesIO

def download_and_unzip(url, extract_to='.'):
    """
    Downloads a ZIP file from the given URL and unzips it to the given direc

    Parameters:
    url (str): The URL to download the ZIP file from.
    extract_to (str): The directory to extract the contents of the ZIP file
    """
```

```python
    try:
        # Send a GET request to the URL
        print(f"Downloading file from {url}")
        response = requests.get(url)
        response.raise_for_status()

        # Extract all the contents of the zip file in the directory 'extract
        with ZipFile(BytesIO(response.content)) as zip_file:
            print(f"Extracting contents to {extract_to}")
            zip_file.extractall(path=extract_to)
            print("Extraction completed.")
    except requests.exceptions.HTTPError as http_err:
        print(f"HTTP error occurred: {http_err}")  # Python 3.6
    except Exception as err:
        print(f"An error occurred: {err}")

download_and_unzip('https://download.pytorch.org/tutorial/data.zip', '.')

print('\nFirst 10 lines of the file:')
!head data/eng-fra.txt
```

```
Downloading file from https://download.pytorch.org/tutorial/data.zip
Extracting contents to .
Extraction completed.

First 10 lines of the file:
Go.     Va !
Run!    Cours !
Run!    Courez !
Wow!    Ça alors !
Fire!   Au feu !
Help!   À l'aide !
Jump.   Saute.
Stop!   Ça suffit !
Stop!   Stop !
Stop!   Arrête-toi !
```

We will now process the data into pairs of french and english sentences. The full process for preparing the data is:

- Read the text file, split each line into pairs (the pairs are tab separated)
- Pre-process all sentences, and filter by length and content
- Split sentences into list of words, create two dictionaries (for english and french).

Simplifications:

- The files are all in Unicode, we will turn Unicode characters to ASCII, make everything lowercase, and trim most punctuation.
- Since we want to train something quickly, we'll trim the data set to only relatively short (10 word maximum) and simple sentences (starting with "I am" or "She is").

In [4]:
```python
# Turn a Unicode string to plain ASCII, thanks to
# https://stackoverflow.com/a/518232/2809427
def unicode2ascii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

# Lowercase, trim, and remove non-letter characters
def preprocess_string(s):
    s = unicode2ascii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z!?]+", r" ", s)
    return s.strip()

MAX_LENGTH = 10
ENG_PREFIXES = (
    "i am ", "i m ",
    "he is", "he s ",
    "she is", "she s ",
    "you are", "you re ",
    "we are", "we re ",
    "they are", "they re "
)

# Filter pairs
def filter_pairs(pairs):
    subset = []
    for fr, en in pairs:
        if len(fr.split(' ')) > MAX_LENGTH:
            continue
        if len(en.split(' ')) > MAX_LENGTH:
            continue
        if not en.startswith(ENG_PREFIXES):
            continue
        subset.append((fr, en))
    return subset


# Read the data
def read_dataset(lang1, lang2, reverse=False):

    # Read the file and split into lines
    print("Reading lines...")
    lines = open('data/%s-%s.txt' % (lang1, lang2), encoding='utf-8').\
        read().strip().split('\n')

    # Split every line into pairs and normalize
    print("Processing lines...")
    pairs = [[preprocess_string(s) for s in l.split('\t')] for l in lines]

    # Reverse pairs
    if reverse:
```

```
        pairs = [list(reversed(p)) for p in pairs]

    # Filter pairs by length and content
    pairs = filter_pairs(pairs)

    print("Finished processing")
    return pairs

corpus_pairs = read_dataset('eng', 'fra', reverse=True)

print(f"\nFound {len(corpus_pairs)} translation pairs.")
print("Here are 10 examples")
for _ in range(10):
    fr, en = random.choice(corpus_pairs)
    print(f"French: {fr} -> English: {en}")
```

```
Reading lines...
Processing lines...
Finished processing

Found 12038 translation pairs.
Here are 10 examples
French: vous etes tres braves -> English: you re very brave
French: elle cherche les cles de sa voiture -> English: she is looking for h
er car keys
French: je ne suis pas autorisee a vous aider -> English: i m not allowed to
help you
French: nous sommes inquiets pour leur securite -> English: we are anxious f
or their safety
French: je perds du poids -> English: i m losing weight
French: je suis un peu ivre -> English: i m a bit drunk
French: vous etes depourvue d ambition -> English: you re unambitious
French: je suis tetu -> English: i m stubborn
French: vous etes lunatique -> English: you re temperamental
French: ils sont en train de temporiser -> English: they re stalling
```

We'll need a unique index per word to use as the inputs and targets of the networks later. To keep track of all this we will use a helper class called `Lang` which has word → index (`word2index`) and index → word (`index2word`) dictionaries, as well as a count of each word `word2count` which will be used to replace rare words later.

In [5]:
```
SOS_token = 0
EOS_token = 1

class Lang:
    def __init__(self, name):
        self.name = name
        self.word2index = {"PAD": 0, "SOS": 1, "EOS": 2, "UNK": 3}
        self.index2word = {0: "PAD", 1: "SOS", 2: "EOS", 3: "UNK"}
        self.word2count = {}
        self.n_words = 4  # Count SOS and EOS

    def addSentence(self, sentence):
```

```python
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1
        else:
            self.word2count[word] += 1

    def tokenize(self, sentence, seq_len=None):
        # Add Start Of Sentence token
        token_seq_idx = [self.word2index["SOS"]]

        # Tokenize each word in sentence
        for tkn in sentence.split():
            token_seq_idx.append(self.word2index[tkn if tkn in self.word2inc

        # Add End Of Sentence token
        token_seq_idx.append(self.word2index["EOS"])

        if seq_len is not None:
            if len(token_seq_idx) < seq_len:
                # Pad to desired lengh
                token_seq_idx += [self.word2index["PAD"]] * (seq_len - len(t
            else:
                # Trim sentence to length
                token_seq_idx = token_seq_idx[:seq_len]

        return token_seq_idx

    def list2sentence(self, seq_ids):
        return " ".join([self.index2word[idx] for idx in seq_ids])


print("Creating French and English dictionaries.")
fr_vocab = Lang('fr')
en_vocab = Lang('en')
for fr, en in corpus_pairs:
    fr_vocab.addSentence(fr)
    en_vocab.addSentence(en)

print(f"French: {fr_vocab.n_words} words found.")
print(f"English: {en_vocab.n_words} words found.")
```

```
Creating French and English dictionaries.
French: 4771 words found.
English: 3111 words found.
```

In [6]:

```python
def create_dataloaders(batch_size):
    # Create two huge tensor with all english and french sentences
    n = len(corpus_pairs)
```

```
    french_seqs_ids = torch.zeros((n, MAX_LENGTH+2)).long()
    english_seqs_ids = torch.zeros((n, MAX_LENGTH+2)).long()

    for idx, (fr, en) in enumerate(corpus_pairs):
        french_seqs_ids[idx] = torch.tensor(fr_vocab.tokenize(fr, seq_len=MA
        english_seqs_ids[idx] = torch.tensor(en_vocab.tokenize(en, seq_len=M

    # Split into training and testing
    train_sample_mask = torch.rand((n,)) > 0.3
    train_french_seqs_ids = french_seqs_ids[train_sample_mask]
    train_english_seqs_ids = english_seqs_ids[train_sample_mask]
    test_french_seqs_ids = french_seqs_ids[~train_sample_mask]
    test_english_seqs_ids = english_seqs_ids[~train_sample_mask]

    # Create train dataloader
    train_data = TensorDataset(train_french_seqs_ids.to(device), train_engli
    train_dataloader = DataLoader(train_data, sampler=RandomSampler(train_da

    # Create test dataloader
    test_data = TensorDataset(test_french_seqs_ids.to(device), test_english_
    # test_dataloader = DataLoader(test_data, sampler=RandomSampler(train_da
    return train_dataloader, test_data

# Test the dataloader
train_dataloader, test_data = create_dataloaders(32)
for fr, en in train_dataloader:
    print('Batch | fr =', fr.shape, '| en =', en.shape)
    print('First sentence in French: ', fr_vocab.list2sentence(fr[0].tolist(
    print('First sentence in English:', en_vocab.list2sentence(en[0].tolist(
    break
```

```
Batch | fr = torch.Size([32, 12]) | en = torch.Size([32, 12])
First sentence in French:  SOS je vais a un concert la semaine prochaine EOS
PAD PAD
First sentence in English: SOS i am going to a concert next week EOS PAD PAD
```

The Seq2Seq Model

This tutorial demonstrates how to create and train a sequence-to-sequence Transformer model to translate French into English. The Transformer was originally proposed in "Attention is all you need" by Vaswani et al. (2017).

Transformers are deep neural networks that replace CNNs and RNNs with self-attention. Self attention allows Transformers to easily transmit information across the input sequences.

As explained in the Google AI Blog post:

> Neural networks for machine translation typically contain an encoder reading the input sentence and generating a representation of it. A

> decoder then generates the output sentence word by word while consulting the representation generated by the encoder. The Transformer starts by generating initial representations, or embeddings, for each word... Then, using self-attention, it aggregates information from all of the other words, generating a new representation per word informed by the entire context, represented by the filled balls. This step is then repeated multiple times in parallel for all words, successively generating new representations.

Figure 1: Applying the Transformer to machine translation. Source: Google AI Blog.

Why Transformers are significant

- Transformers excel at modeling sequential data, such as natural language.
- Unlike the recurrent neural networks (RNNs), Transformers are parallelizable. This makes them efficient on hardware like GPUs and TPUs. The main reasons is that Transformers replaced recurrence with attention, and computations can happen

simultaneously. Layer outputs can be computed in parallel, instead of a series like an RNN.

- Unlike RNNs (like seq2seq, 2014) or convolutional neural networks (CNNs) (for example, ByteNet), Transformers are able to capture distant or long-range contexts and dependencies in the data between distant positions in the input or output sequences. Thus, longer connections can be learned. Attention allows each location to have access to the entire input at each layer, while in RNNs and CNNs, the information needs to pass through many processing steps to move a long distance, which makes it harder to learn.
- Transformers make no assumptions about the temporal/spatial relationships across the data. This is ideal for processing a set of objects (for example, StarCraft units).

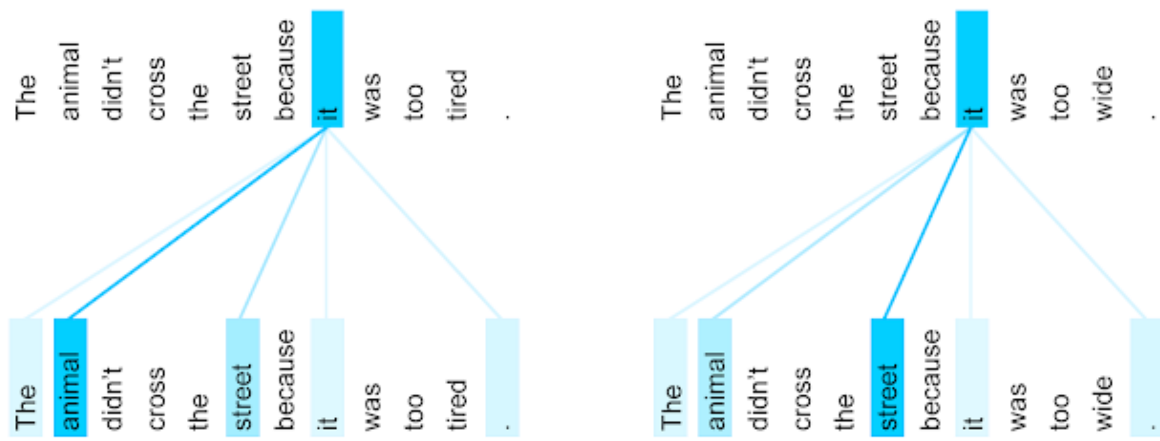
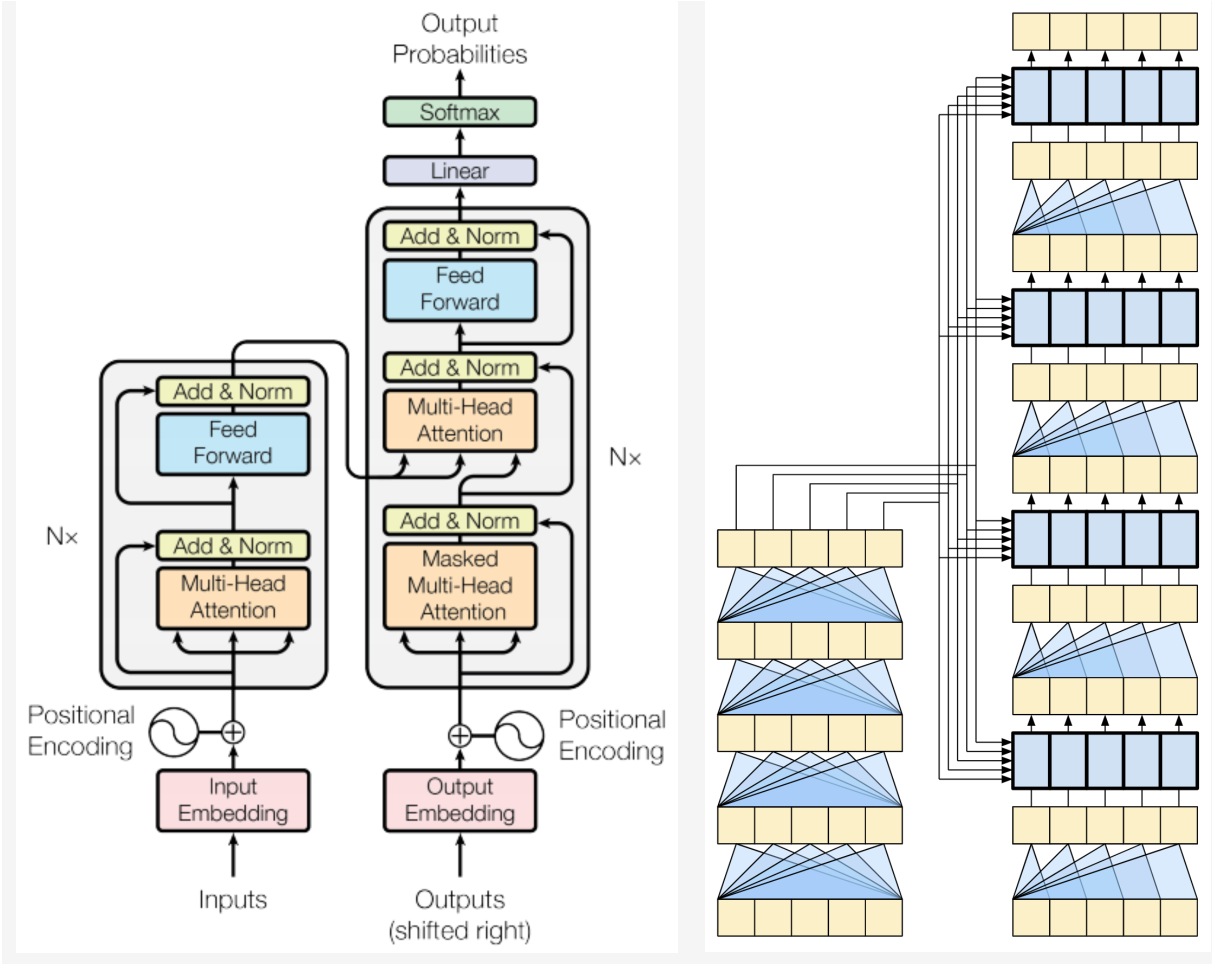


Figure 3: The encoder self-attention distribution for the word “it” from the 5th to the 6th layer of a Transformer trained on English-to-French translation (one of eight attention heads). Source: Google AI Blog.

There's a lot going on inside a Transformer. The important things to remember are:

1. It follows the same general pattern as a standard sequence-to-sequence model with an encoder and a decoder. The encoder processes the input sentence into a set of vector representations (one for each word), and the decoder uses the encoder's outputs to predict the target (ie, translated) sentence.
2. If you work through it step by step it will all make sense.

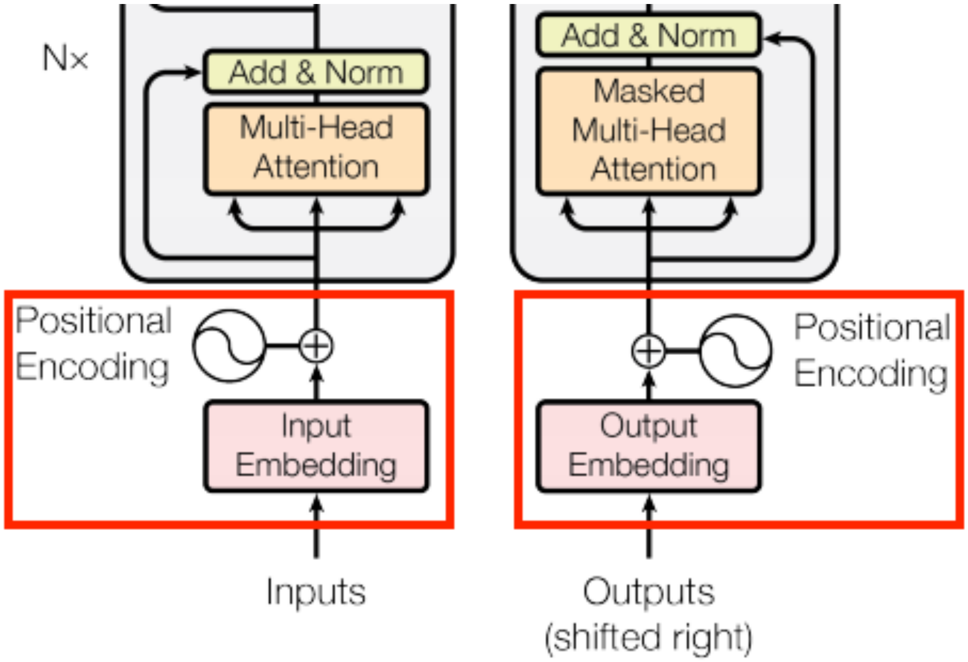
The original Transformer diagram	A representation of a 4-layer Transformer



Each of the components in these two diagrams will be explained next. Namely,

- Embedding and positional encoding layer
- Add & Norm layer
- Multi-Head Attention Layers
- Feed Forward Layers

The embedding and positional encoding layer



The inputs to both the encoder and decoder use the same embedding and positional encodings.

First, given a sequence of tokens, both the input tokens (French) and target tokens (English) have to be converted into vectors using a `nn.Embedding` layer.

Second, since attention layers see their input as an unordered set of vectors, it needs some way to identify word order. Otherwise, sentences like, `how are you`, `how you are`, `you how are`, and so on, would be indistinguishable. A Transformer adds a "Positional Encoding" to the embedding vectors. Positional Encodings are just vectors that uniquely identify word position. Also, ideally, nearby words should have similar position encodings.

The original paper uses a set of sines and cosines with different frequencies (across the sequence) for calculating the positional encoding

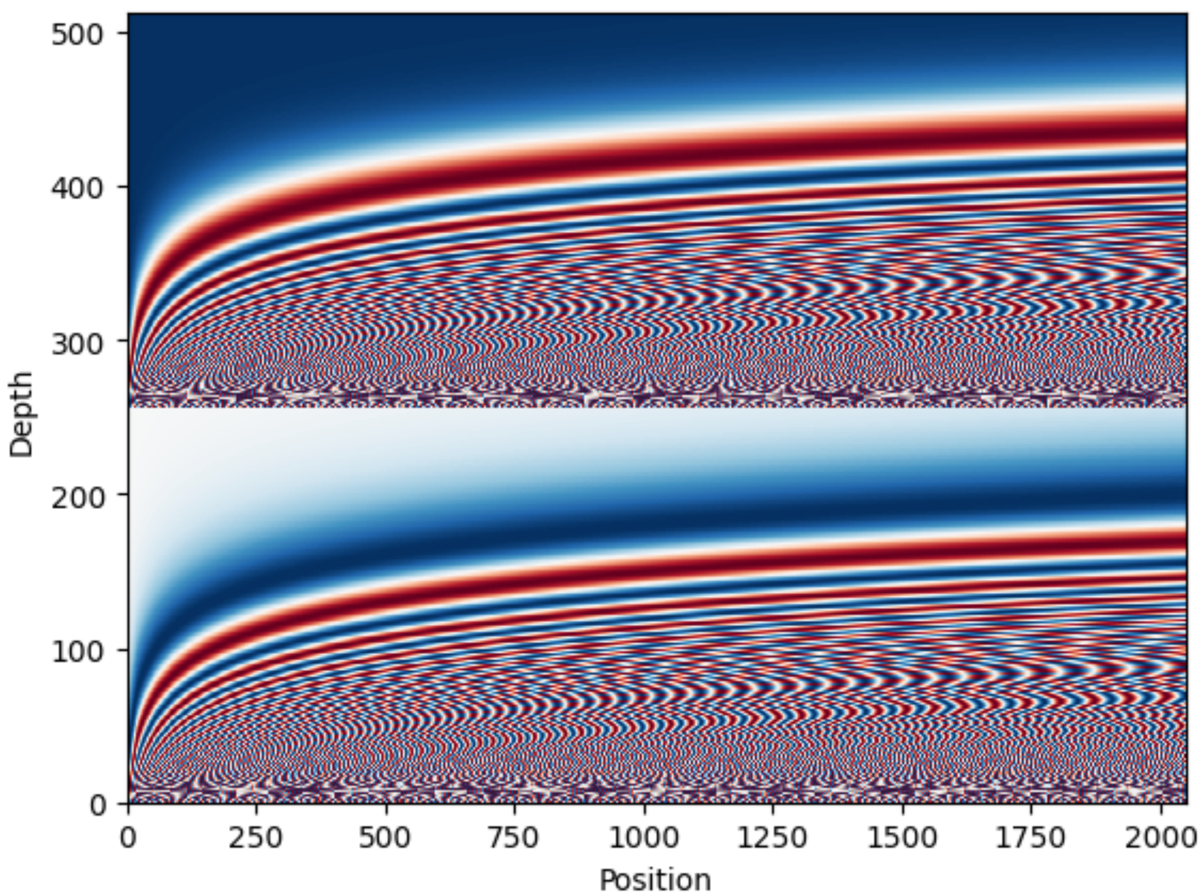
$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

In [17]:
```
def positional_encoding(length, depth):
    depth = depth/2

    positions = np.arange(length)[:, np.newaxis]     # (seq, 1)
    depths = np.arange(depth)[np.newaxis, :]/depth   # (1, depth)

    angle_rates = 1 / (10000**depths)         # (1, depth)
    angle_rads = positions * angle_rates      # (pos, depth)
```

```python
    pos_encoding = np.concatenate(
        [np.sin(angle_rads), np.cos(angle_rads)],
        axis=-1)

    return pos_encoding


pos_encoding = positional_encoding(length=2048, depth=512)

# Visualize Position Embeddings
import matplotlib.pyplot as plt
print("Position Encodings: (Max Position, Embedding Size) =", pos_encoding.s
plt.pcolormesh(pos_encoding.T, cmap='RdBu')
plt.ylabel('Depth')
plt.xlabel('Position')
plt.show()
```

```
Position Encodings: (Max Position, Embedding Size) = (2048, 512)
```



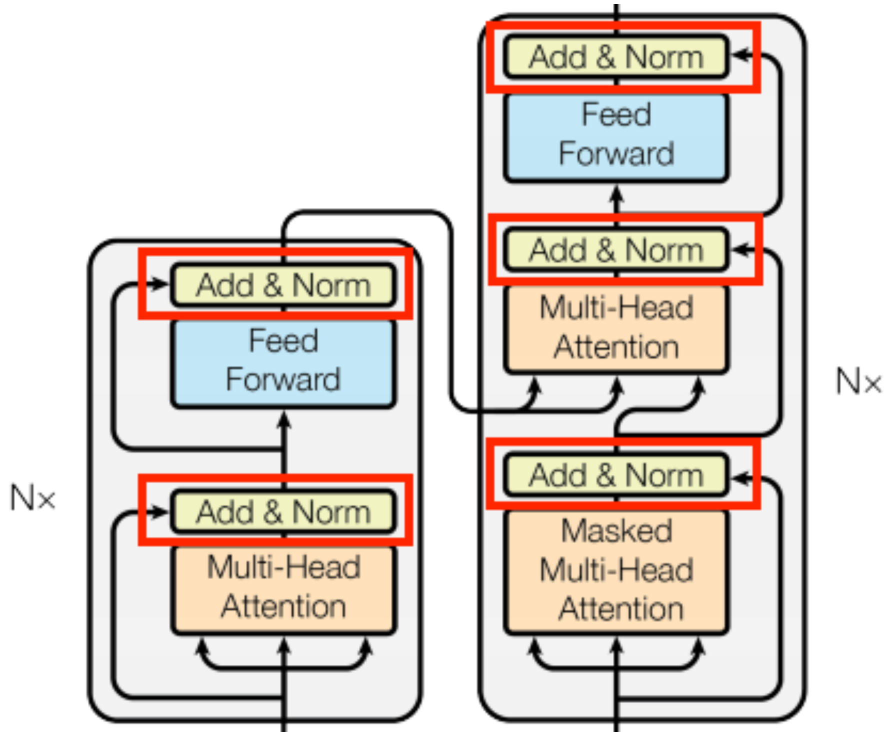
To combine info about the word itself and the word location within the sequence, we create a WordPosEmbedding layer that looks-up a token's embedding vector and adds the position vector. Since we are working with two different languages, we need to use two different token embeddings.

In [ ]:
```python
class WordPosEmbedding(nn.Module):
    def __init__(self, vocab_size, d_model):
```

```
        super().__init__()
        self.d_model = d_model
        self.embedding = nn.Embedding(vocab_size, d_model)
        nn.init.normal_(self.embedding.weight, mean=0, std=0.01)
        self.pos_encoding = torch.Tensor(positional_encoding(length=2048, de
        self.pos_encoding.requires_grad = False

    def compute_mask(self, *args, **kwargs):
        return self.embedding.compute_mask(*args, **kwargs)

    def forward(self, x):
        length = x.shape[1]
        x = self.embedding(x)
        # This factor sets the relative scale of the embedding and positonal
        x *= (self.d_model ** 0.5)
        x = x + self.pos_encoding[None, :length, :]
        return x

embed_fr = WordPosEmbedding(vocab_size=fr_vocab.n_words, d_model=512).to(dev
embed_en = WordPosEmbedding(vocab_size=en_vocab.n_words, d_model=512).to(dev

# Example usage: embed layer receives a batch of sequences of word indexes
# (ie, a matrix of size BxL where B is batch size and L sequence lenght)
en_sentence = 'i am awesome'
en_seq = torch.tensor([en_vocab.word2index[w] for w in en_sentence.split()])
print(en_seq.shape)
en_tkn_seq = embed_en(en_seq.to(device))
print(en_tkn_seq.shape)

fr_sentence = 'je plaisante'
fr_seq = torch.tensor([fr_vocab.word2index[w] for w in fr_sentence.split()])
print(fr_seq.shape)
fr_tkn_seq = embed_fr(fr_seq.to(device))
print(fr_tkn_seq.shape)
```

```
torch.Size([1, 3])
torch.Size([1, 3, 512])
torch.Size([1, 2])
torch.Size([1, 2, 512])
```

Add and normalize



These "Add & Norm" blocks are scattered throughout the model. Each one just implements a residual connection and followed by a `LayerNormalization` layer.

The residual "Add & Norm" blocks are included so that training is efficient. The residual connection provides a direct path for the gradient (and ensures that vectors are **updated** by the attention layers instead of **replaced**), while the normalization maintains a reasonable scale for the outputs.

In [19]:
```
class AddNorm(nn.Module):
    def __init__(self, d_model):
        super().__init__()
        self.norm = nn.LayerNorm(d_model)

    def forward(self, x, res):
        return self.norm(x + res)
```

Attention refresher

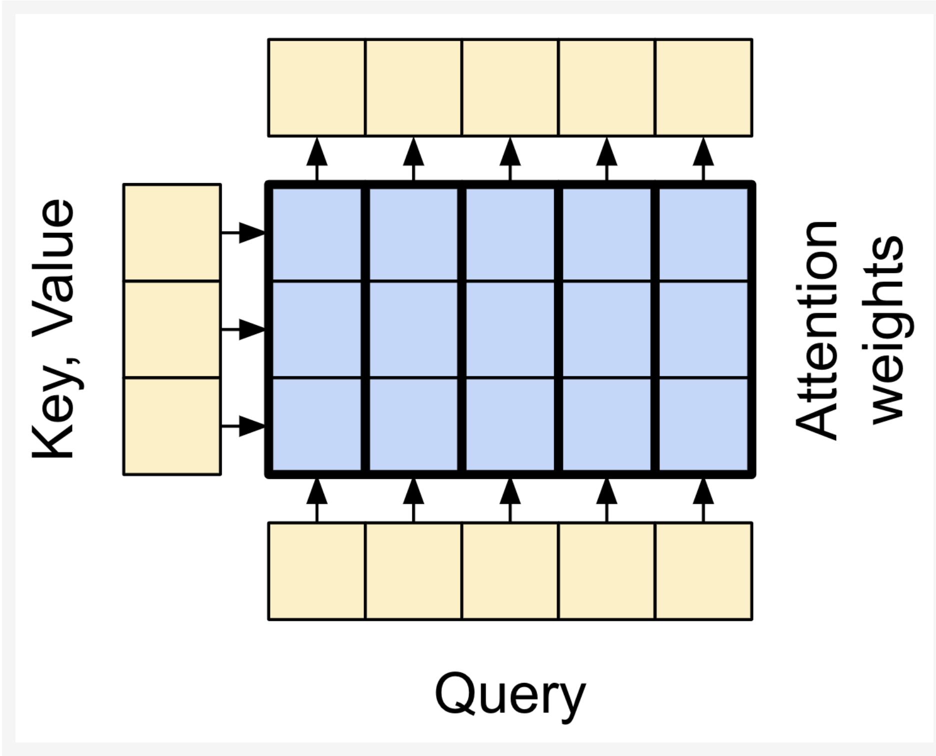
Before you get into the specifics of each usage, here is a quick refresher on how attention works.

There are two inputs:

1. The query sequence; the sequence being processed; the sequence doing the attending (bottom).
2. The context sequence; the sequence being attended to (left).

The output has the same shape as the query-sequence.

The base attention layer



For an intuitive understanding of attention, the common comparison is that this operation is like a dictionary lookup. A **fuzzy**, **differentiable**, **vectorized** dictionary lookup.

Here's a regular python dictionary, with 3 keys and 3 values being passed a single query.

```
d = {'color': 'blue', 'age': 22, 'type': 'pickup'}
result = d['color']
```

- The `query` s is what you're trying to find.
- The `key` s what sort of information the dictionary has.
- The `value` is that information.

When you look up a `query` in a regular dictionary, the dictionary finds the matching `key`, and returns its associated `value`. The `query` either has a matching `key` or it doesn't. You can imagine a **fuzzy** dictionary where the keys don't have to match perfectly. If you looked up `d["species"]` in the dictionary above, maybe you'd want it to return `"pickup"` since that's the best match for the query.

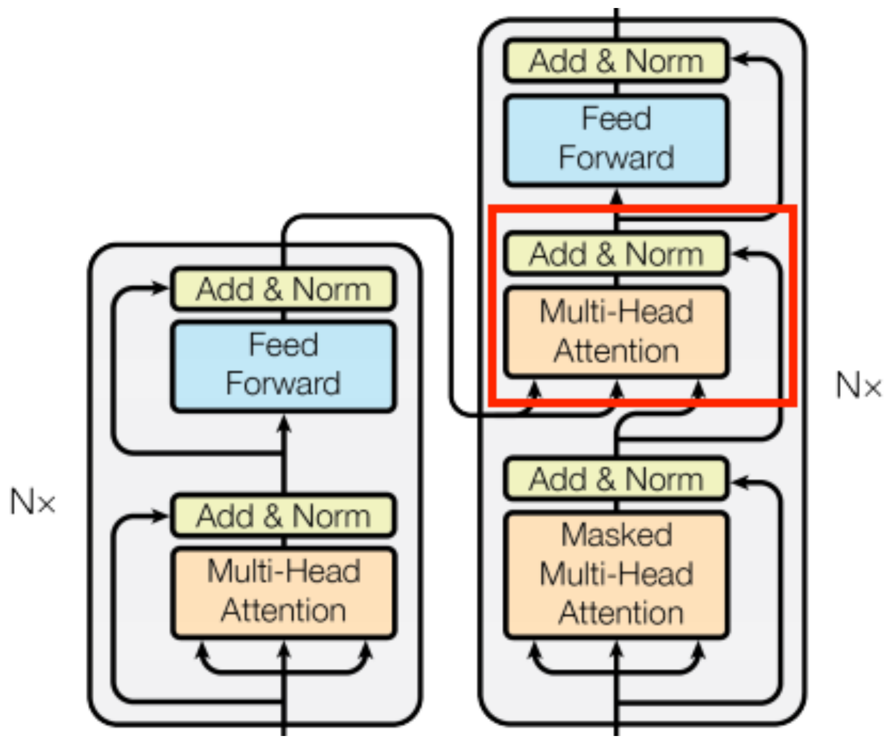
An attention layer does a fuzzy lookup like this, but it's not just looking for the best key. It combines the `values` based on how well the `query` matches each `key`.

How does that work? In an attention layer the `query`, `key`, and `value` are each vectors. Instead of doing a hash lookup the attention layer combines the `query` and `key` vectors to determine how well they match, the "attention score". The layer returns the average across all the `values`, weighted by the "attention scores".

Each location the query-sequence provides a `query` vector. The context sequence acts as the dictionary. At each location in the context sequence provides a `key` and `value` vector. The input vectors are not used directly, the `nn.MultiheadAttention` layer includes `nn.Dense` layers to project the input vectors before using them.

There are three different types of attention layers used throughout the model. These are all identical except for how the attention is configured. Lets take a look at each one of them now.

The cross attention layer



At the literal center of the Transformer is the cross-attention layer. This layer connects the encoder and decoder. It updates the decoder representations by attending to all encoder sequence. To implement this, you pass the target sequence `x` as the `query` and the `context` sequence as the `key/value` when calling the `mha` layer. Furthermore, since the queries can attend to the entire input sequence representation, obtained from the encoder, then no causal mask is applied.

E

In [10]:
```
class CrossAttention(nn.Module):
    def __init__(self, d_model, num_heads, dropout=0.):
        super().__init__()
        self.mha = nn.MultiheadAttention(d_model, num_heads, dropout=dropout
        self.add_norm = AddNorm(d_model)
```

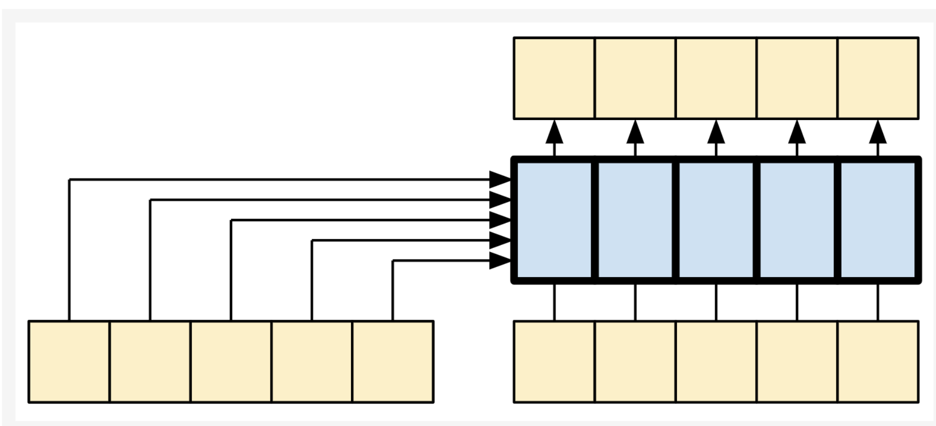
```
    def forward(self, x, context):
        attn_output, attn_scores = self.mha.forward(
            query=x, key=context, value=context)
        x = self.add_norm(x, attn_output)

        # Cache the attention scores for plotting later.
        self.last_attn_scores = attn_scores
        return x

# Example usage
sample_ca = CrossAttention(d_model=512, num_heads=2).to(device)
print('Batch of English Sentences:', en_tkn_seq.shape)
print('Batch of French Sentences:', fr_tkn_seq.shape)
print('Output of Cross-Attention:', sample_ca(en_tkn_seq.to(device), fr_tkn_
```

```
Batch of English Sentences: torch.Size([1, 3, 512])
Batch of French Sentences: torch.Size([1, 2, 512])
Output of Cross-Attention: torch.Size([1, 3, 512])
```

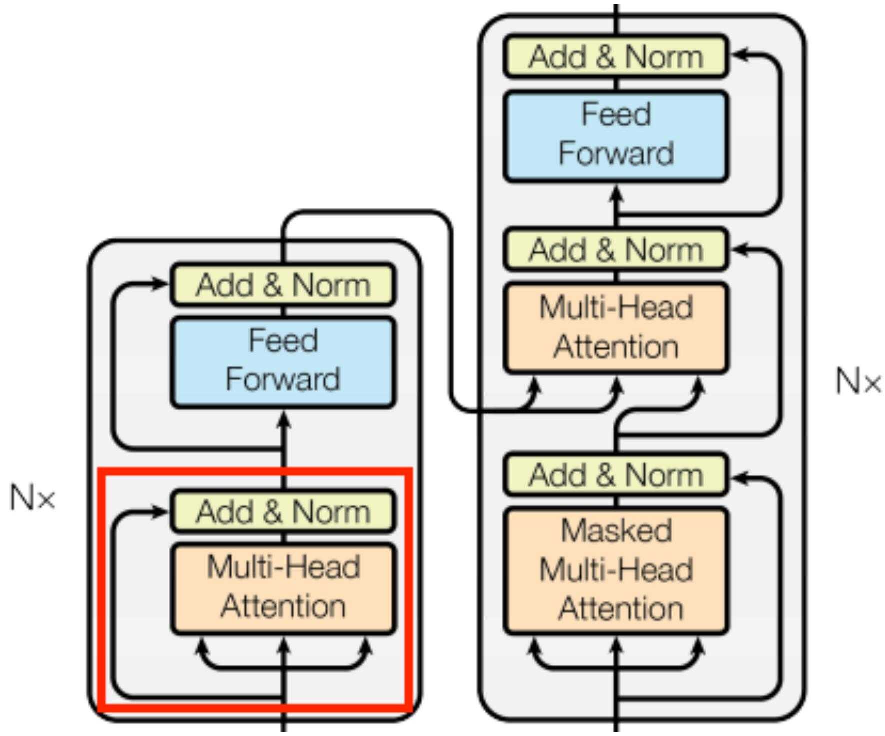
The caricature below shows how information flows through this layer. For simplicity the residual connections are not shown.

The output length is the length of the `query` sequence, and not the length of the context `key/value` sequence. The point is that each `query` location (english words in this example) can see all the `key/value` pairs in the context (input french words), but no information is exchanged between the queries.

Each query sees the whole context.



The global self attention layer



This layer is responsible for processing the context sequence (french sentence), and propagating information along its length.

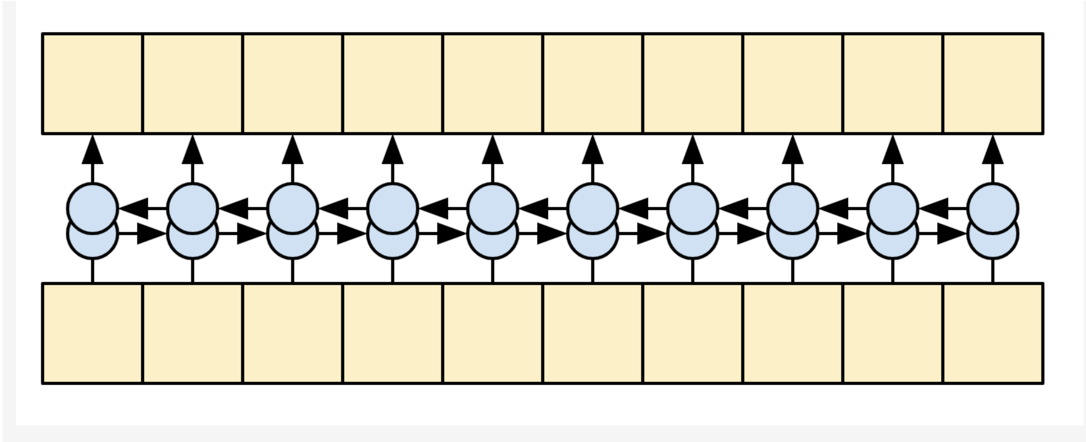
Since the context sequence is fixed while the translation is being generated, information is allowed to flow in both directions.

Before Transformers, models commonly used RNNs or CNNs to do this task. However, RNNs and CNNs have their limitations.

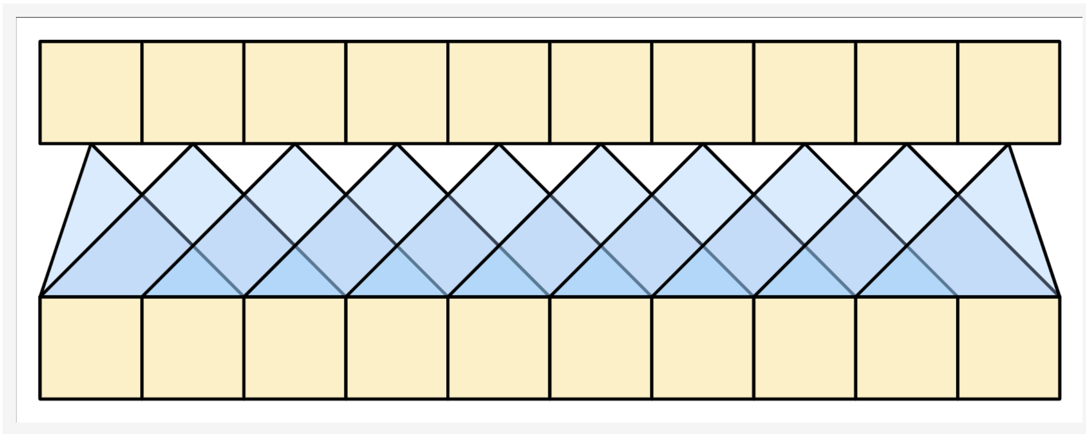
- The RNN allows information to flow all the way across the sequence, but it passes through many processing steps to get there (limiting gradient flow). These RNN steps have to be run sequentially and so the RNN is less able to take advantage of modern parallel devices.
- In the CNN each location can be processed in parallel, but it only provides a limited receptive field. The receptive field only grows linearly with the number of CNN layers, You need to stack a number of Convolution layers to transmit information across the sequence (Wavenet reduces this problem by using dilated convolutions).

The global self attention layer on the other hand lets every sequence element directly access every other sequence element, with only a few operations, and all the outputs can be computed in parallel.

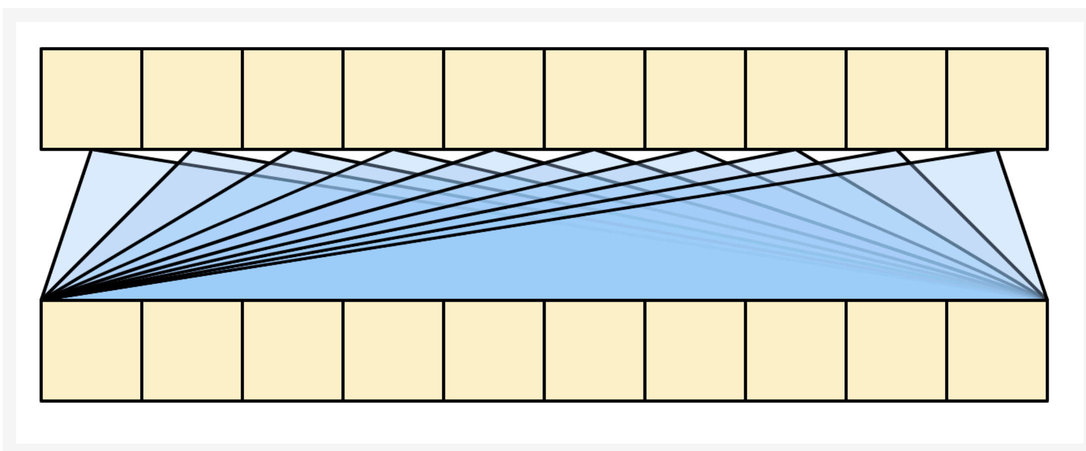
Bidirectional RNNs



CNNs



The global self attention layer



To implement global self-attention you just need to pass the target sequence, `x`, as both the `query`, and `value` arguments to the `mha` layer:

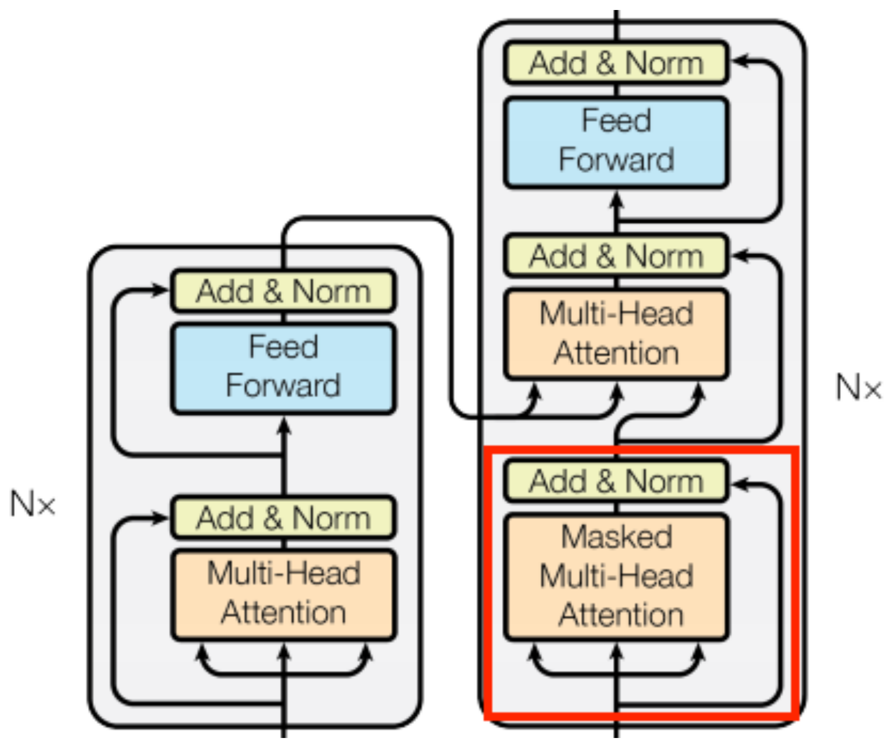
F

In [20]:
```
class GlobalSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads, dropout=0.):
        super().__init__()
        self.mha = nn.MultiheadAttention(d_model, num_heads, dropout=dropout
```

```
        self.add_norm = AddNorm(d_model)

    def forward(self, x):
        attn_output, attn_scores = self.mha(
            query=x,
            key=x,
            value=x)
        x = self.add_norm(x, attn_output)

        # Cache the attention scores for plotting later.
        self.last_attn_scores = attn_scores
        return x


# Example usage
sample_gsa = GlobalSelfAttention(d_model=512, num_heads=2).to(device)
print('Batch of French Sentences:', fr_tkn_seq.shape)
print('Output of Global Self-Attention:', sample_gsa(fr_tkn_seq.to(device)).
```

```
Batch of French Sentences: torch.Size([1, 2, 512])
Output of Global Self-Attention: torch.Size([1, 2, 512])
```

The causal self attention layer



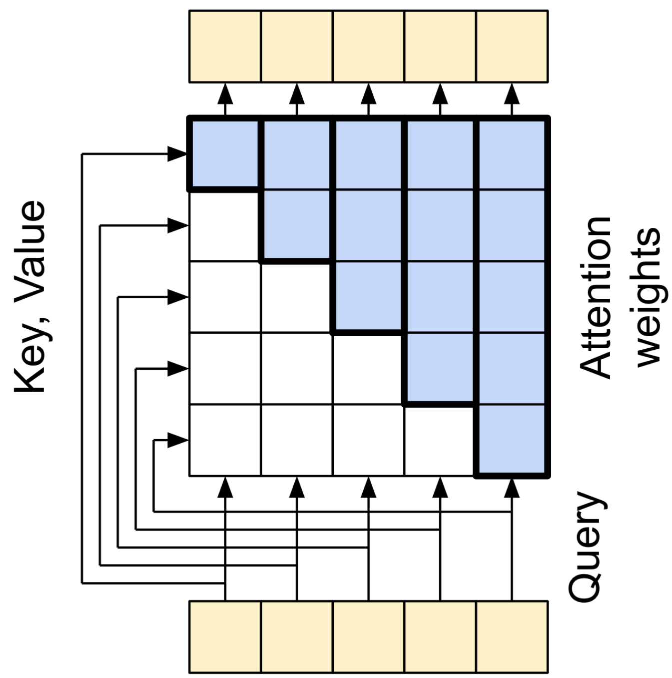
This layer does a similar job as the global self attention layer, for the output sequence. However, since we want to generate the output sequence word-by-word, the query sequence (ie, representing the english translation) can only attend to the previous (already generated) words: the models are "causal".

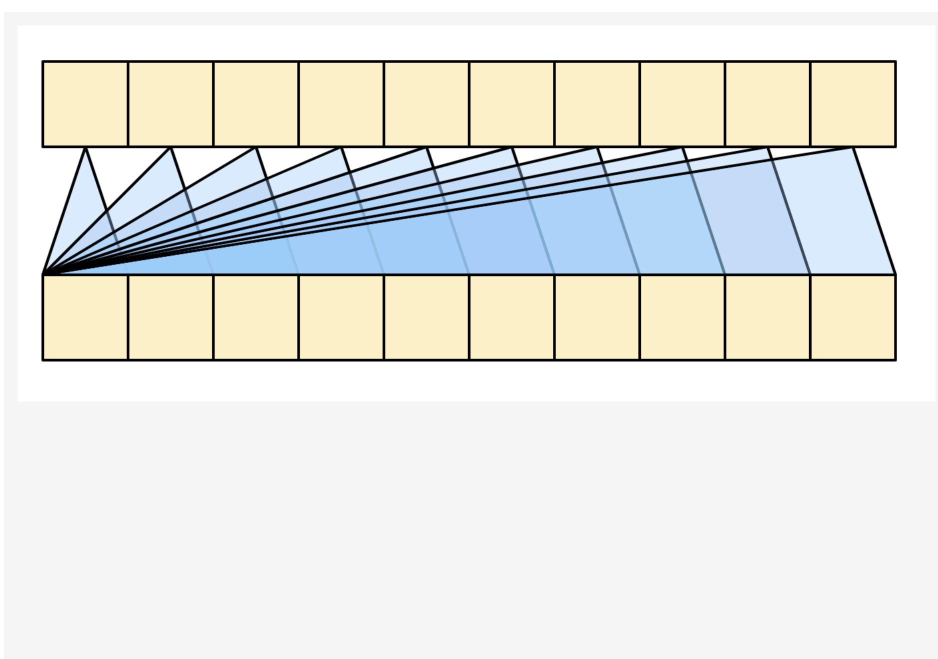
A causal model is efficient in two ways:

2. During training, we can feed the ground truth translation to the decoder input, and have it predict the very next token at all locations. This lets you compute loss for every location in the output sequence while executing the model just once.
3. During inference, for each new token generated you only need to calculate its outputs, the outputs for the previous sequence elements can be reused.

Causal attention is accomplished using a causal mask, which ensures that each location only has access to the locations that come before it:

The causal self attention layer





Luckily, PyTorch MultiheadAttention already implements causal masks with the function `generate_square_subsequent_mask()`.

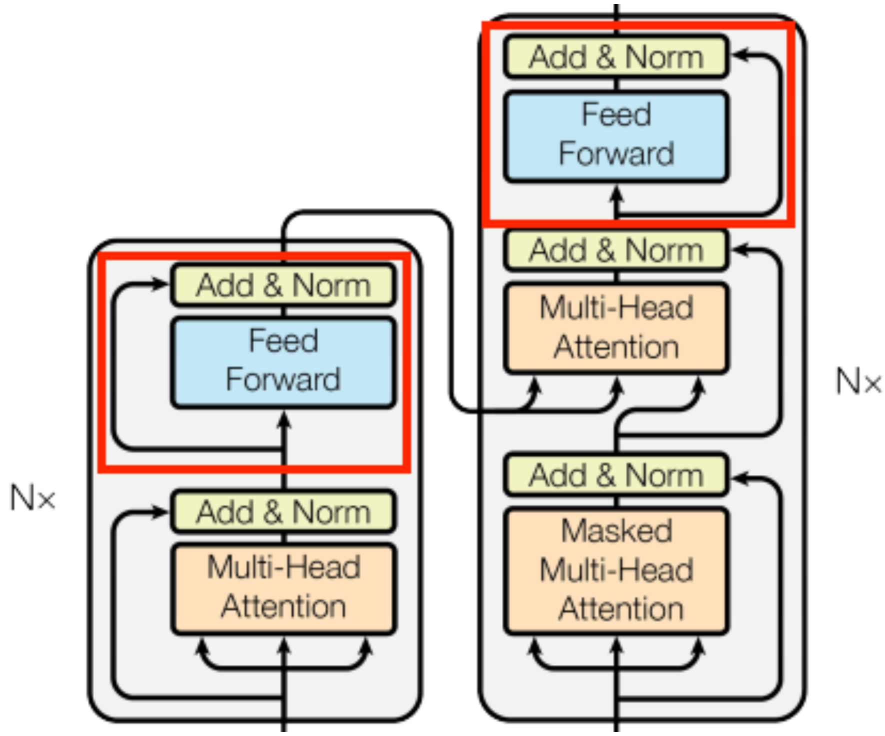
In [21]:
```
from torch.nn import Transformer as TF

class CausalSelfAttention(nn.Module):
    def __init__(self, d_model, num_heads, dropout=0.):
        super().__init__()
        self.mha = nn.MultiheadAttention(d_model, num_heads, dropout=dropout
        self.add_norm = AddNorm(d_model)

    def forward(self, x):
        causal_mask = TF.generate_square_subsequent_mask(x.shape[1], device=
        attn_output, attn_scores = self.mha(
            query=x,
            key=x,
            value=x,
            attn_mask=causal_mask)
        x = self.add_norm(x, attn_output)

        # Cache the attention scores for plotting later.
        self.last_attn_scores = attn_scores
        return x

# Example usage
sample_csa = CausalSelfAttention(d_model=512, num_heads=2).to(device)
print('Batch of English Sentences:', en_tkn_seq.shape)
print('Output of Causal Self-Attention:', sample_csa(en_tkn_seq.to(device)).
```

```
Batch of English Sentences: torch.Size([1, 3, 512])
Output of Causal Self-Attention: torch.Size([1, 3, 512])
```

G

We need it because we shouldnt predic on stuff that happens before time starts

The feed forward network



The transformer also includes point-wise feed-forward networks, which process each token independently (no interactions between words), in both the encoder and decoder.

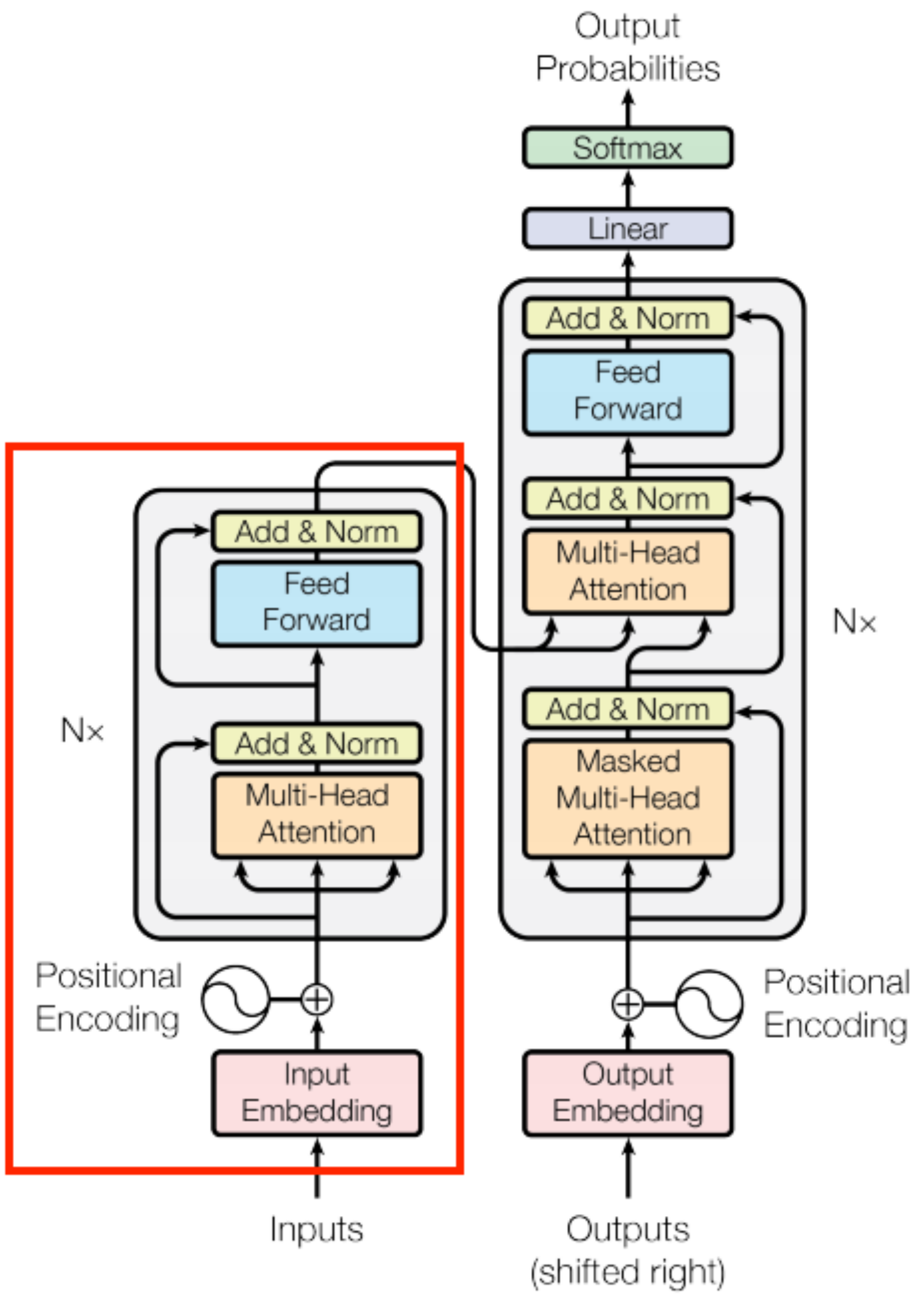
In [22]:
```
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout_rate=0.1):
        super().__init__()
        self.ffnet = nn.Sequential(
            nn.Linear(d_model, d_ff*d_model),
            nn.ReLU(inplace=True),
            nn.Linear(d_ff*d_model, d_model),
            nn.Dropout(dropout_rate)
        )
        self.add_norm = AddNorm(d_model)

    def forward(self, x):
        x = self.add_norm(x, self.ffnet(x))
        return x

# Example usage
sample_ffnet = FeedForward(d_model=512, d_ff=4).to(device)
print('Batch of English Sentences:', en_tkn_seq.shape)
print('Output of Causal Self-Attention:', sample_ffnet(en_tkn_seq.to(device)
```

```
Batch of English Sentences: torch.Size([1, 3, 512])
Output of Causal Self-Attention: torch.Size([1, 3, 512])
```

The Encoder and Decoder

Now, that we know how each component of a Transformer model work, lets put them all together.

The encoder contains a `WordPosEmbedding` layer at the input and a stack of `N` encoder layers. Each `EncoderLayer` contains a `GlobalSelfAttention` and `FeedForward` layer.

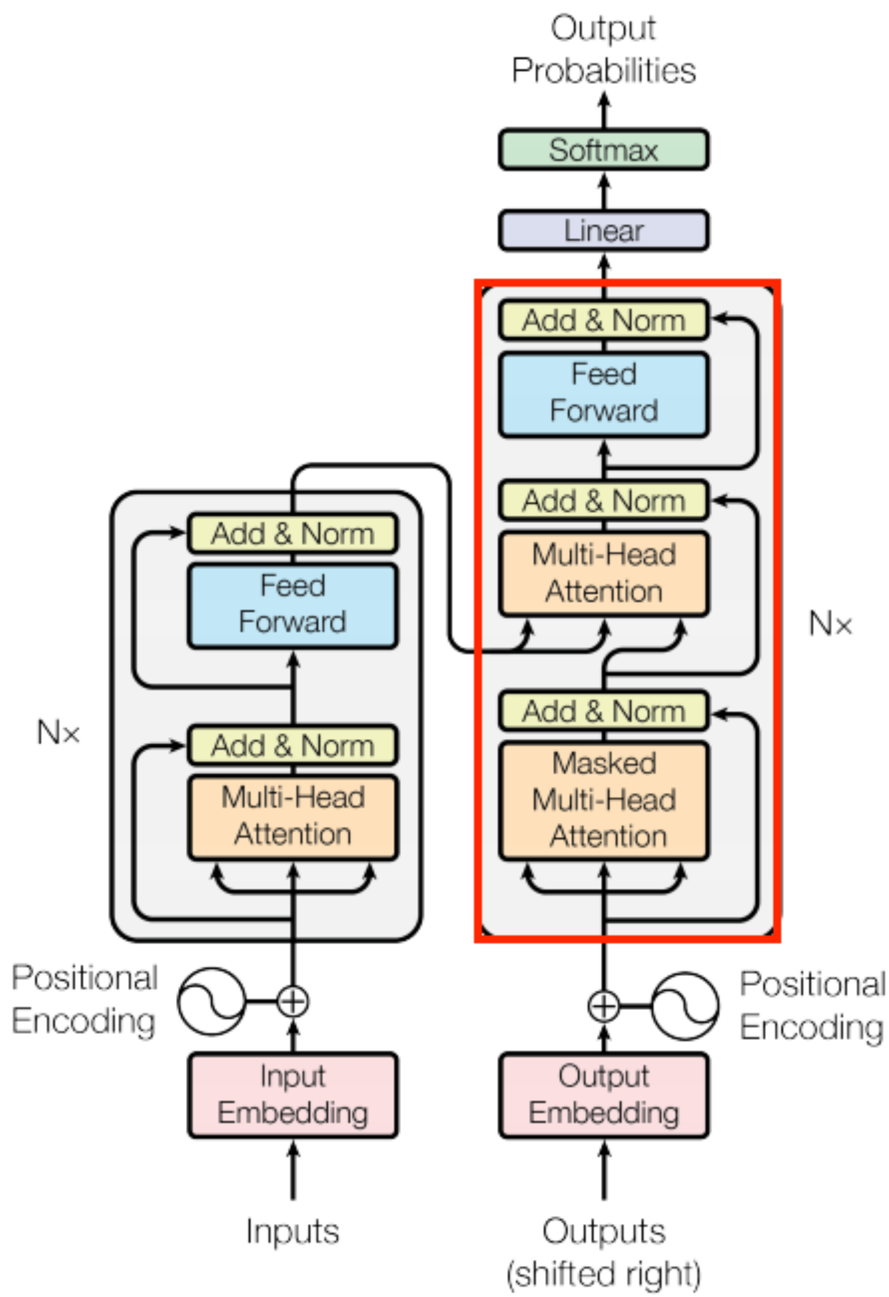


In [25]:
```
class EncoderLayer(nn.Module):
    def __init__(self,d_model, num_heads, dff, dropout_rate=0.1):
        super().__init__()
        self.self_attention = GlobalSelfAttention(
            d_model=d_model,
            num_heads=num_heads,
            dropout=dropout_rate)
        self.ffn = FeedForward(d_model, dff)
```

```python
    def forward(self, x):
        x = self.self_attention(x)
        x = self.ffn(x)
        return x


class Encoder(nn.Module):
    def __init__(self, num_layers, d_model, num_heads,
                 dff, vocab_size, dropout_rate=0.1):
        super().__init__()

        self.d_model = d_model
        self.num_layers = num_layers

        self.pos_embedding = WordPosEmbedding(
            vocab_size=vocab_size, d_model=d_model)

        self.enc_layers = nn.ModuleList([
            EncoderLayer(d_model=d_model,
                         num_heads=num_heads,
                         dff=dff,
                         dropout_rate=dropout_rate)
            for _ in range(num_layers)])
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, x):
        # `x` is token-IDs shape: (batch, seq_len)
        x = self.pos_embedding(x)  # Shape `(batch_size, seq_len, d_model)`.

        # Add dropout.
        x = self.dropout(x)

        for i in range(self.num_layers):
            x = self.enc_layers[i](x)

        return x  # Shape `(batch_size, seq_len, d_model)`.


# Example usage
encoder = Encoder(num_layers=3, d_model=512, num_heads=8, dff=4, vocab_size=
print('Batch of English Sentences:', fr_seq.shape)
print('Output of Causal Self-Attention:', encoder(fr_seq.to(device)).shape)
```

```
Batch of English Sentences: torch.Size([1, 2])
Output of Causal Self-Attention: torch.Size([1, 2, 512])
```

The decoder's stack starts with `WordPosEmbedding`, followed by a series of `DecoderLayer`, containing a `CausalSelfAttention`, a `CrossAttention`, and a `FeedForward` layer.



In [26]:
```python
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dff, dropout_rate=0.1):
        super(DecoderLayer, self).__init__()

        self.causal_self_attention = CausalSelfAttention(
            d_model=d_model,
            num_heads=num_heads,
            dropout=dropout_rate)

        self.cross_attention = CrossAttention(
            d_model=d_model,
            num_heads=num_heads,
            dropout=dropout_rate)
```

```python
        self.ffn = FeedForward(d_model, dff)

    def forward(self, x, context):
        x = self.causal_self_attention(x=x)
        x = self.cross_attention(x=x, context=context)
        x = self.ffn(x)  # Shape `(batch_size, seq_len, d_model)`.

        # Cache the last attention scores for plotting later
        self.last_attn_scores = self.cross_attention.last_attn_scores

        return x


class Decoder(nn.Module):
    def __init__(self, num_layers, d_model, num_heads, dff, vocab_size,
                 dropout_rate=0.1):
        super(Decoder, self).__init__()

        self.d_model = d_model
        self.num_layers = num_layers

        self.pos_embedding = WordPosEmbedding(vocab_size=vocab_size,
                                              d_model=d_model)
        self.dropout = nn.Dropout(dropout_rate)
        self.dec_layers = nn.ModuleList([
            DecoderLayer(d_model=d_model, num_heads=num_heads,
                         dff=dff, dropout_rate=dropout_rate)
            for _ in range(num_layers)])

        self.last_attn_scores = None

    def forward(self, x, context):
        # `x` is token-IDs shape (batch, target_seq_len)
        x = self.pos_embedding(x)  # (batch_size, target_seq_len, d_model)
        x = self.dropout(x)

        for i in range(self.num_layers):
            x  = self.dec_layers[i](x, context)

        self.last_attn_scores = self.dec_layers[-1].last_attn_scores

        # The shape of x is (batch_size, target_seq_len, d_model).
        return x


# Example usage
decoder = Decoder(num_layers=3, d_model=512, num_heads=8, dff=4, vocab_size=
print('Batch of French Sentences:', fr_seq.shape)
print('Batch of English Sentences:', en_seq.shape)
fr_feats = encoder(fr_seq.to(device))
tgt_feats = decoder(en_seq.to(device), fr_feats)
print('Output of Causal Self-Attention:', tgt_feats.shape)
```

```
Batch of French Sentences: torch.Size([1, 2])
Batch of English Sentences: torch.Size([1, 3])
Output of Causal Self-Attention: torch.Size([1, 3, 512])
```

In [27]:

```
class Transformer(nn.Module):
    def __init__(self, num_layers, d_model, num_heads, dff,
                 input_vocab_size, target_vocab_size, dropout_rate=0.1):
        super().__init__()
        self.encoder = Encoder(num_layers=num_layers, d_model=d_model,
                               num_heads=num_heads, dff=dff,
                               vocab_size=input_vocab_size,
                               dropout_rate=dropout_rate)

        self.decoder = Decoder(num_layers=num_layers, d_model=d_model,
                               num_heads=num_heads, dff=dff,
                               vocab_size=target_vocab_size,
                               dropout_rate=dropout_rate)

        self.final_layer = nn.Linear(d_model, target_vocab_size)

    def forward(self, x, context):
        # Extracts global representations from the context sequence
        context = self.encoder(context)  # (batch_size, context_len, d_model

        # Processes the predictions using a causal decoder.
        x = self.decoder(x, context)  # (batch_size, target_len, d_model)

        # Predicts the next token using a final linear layer classifier.
        logits = self.final_layer(x)  # (batch_size, target_len, target_voca

        return logits

# Example usage
transformer = Transformer(num_layers=3, d_model=512, num_heads=8, dff=4,
                          input_vocab_size=fr_vocab.n_words,
                          target_vocab_size=en_vocab.n_words).to(device)
print('Batch of French Sentences:', fr_seq.shape)
print('Batch of English Sentences:', en_seq.shape)
print('Output of Causal Self-Attention:', transformer(en_seq.to(device), fr_
```

```
Batch of French Sentences: torch.Size([1, 2])
Batch of English Sentences: torch.Size([1, 3])
Output of Causal Self-Attention: torch.Size([1, 3, 3111])
```

Evaluation

Evaluation is mostly the same as training, but there are no targets so we simply feed the decoder's predictions back to itself for each step. Every time it predicts a word we add it to the output string, and if it predicts the EOS token we stop there.

We can evaluate random sentences from the training set and print out the input, target, and output to make some subjective quality judgements.

In [28]:
```
@torch.no_grad()
def evaluate(transformer, fr_sentence):
    transformer.eval()
    with torch.no_grad():
        # The French sentence is tokenized and converted to a batch of B=1
        french_tkns = torch.tensor(fr_vocab.tokenize(fr_sentence)).long().ur

        # First, the sentence to be translated is encoded using the transfor
        french_feats = transformer.encoder(french_tkns)

        # The translation sentence is initialized with SOS token
        decoded_tkns = torch.tensor([[en_vocab.word2index['SOS']]]).long().t

        # We'll keep track of the predicted logits in order to compute the p
        pred_logits = []

        # Then, we evaluate the decoder, to generate the next words in the t
        for i in range(MAX_LENGTH-1):
            next_pred_feat = transformer.decoder(decoded_tkns, french_feats)
            next_pred_logit = transformer.final_layer(next_pred_feat)
            next_pred = next_pred_logit.argmax(dim=1, keepdims=True)
            pred_logits.append(next_pred_logit)
            if next_pred.item() == en_vocab.word2index['EOS']:
                break
            decoded_tkns = torch.cat((decoded_tkns, next_pred), dim=1)

        decoded_tkns = decoded_tkns.squeeze(0) # squeeze batch dimension
        translation_words = en_vocab.list2sentence(decoded_tkns[1:].tolist()
        pred_logits = torch.cat(pred_logits, 0)
    return translation_words, decoded_tkns, pred_logits

@torch.no_grad()
def evaluate_one_epoch(transformer, n=100):
    transformer.eval()
    criterion = nn.CrossEntropyLoss()
    loss, perplexity = 0., 0.
    for i in tqdm.tqdm(range(n), desc='[EVAL]'):
        fr_tkns, en_tkns = random.choice(test_data)
        fr = fr_vocab.list2sentence(fr_tkns[fr_tkns > 2].tolist())
        _, _, pred_logits = evaluate(transformer, fr)
        l = criterion(pred_logits, en_tkns[1:1+len(pred_logits)]).item()
        loss += l
        perplexity += np.exp(l)
    return loss / n, perplexity / n

@torch.no_grad()
def translate_randomly(transformer, n=3):
    for i in range(n):
        fr_tkn, en_tkn = random.choice(test_data)
        en = en_vocab.list2sentence(en_tkn[en_tkn > 2].tolist())
        fr = fr_vocab.list2sentence(fr_tkn[fr_tkn > 2].tolist())
        print('>', fr)
```

```
        print('=', en)
        output_sentence, _, _ = evaluate(transformer, fr)
        print('<', output_sentence)
        print('')

translate_randomly(transformer)
loss, perplexity = evaluate_one_epoch(transformer)
print('Loss = ', loss)
print('Perplexity = ', perplexity)
```

```
> je suis tres serieuse
= i m quite serious
< weird weird weird weird weird weird weird weird progress

> nous sommes ses fils
= we are his sons
< weird weird weird weird weird weird weird progress progress

> il craint de commettre des erreurs
= he is afraid of making mistakes
< weird weird weird weird weird weird weird weird weird
```

```
[EVAL]: 100%|██████████| 100/100 [00:14<00:00,  7.14it/s]
Loss =  8.130883345603943
Perplexity =  3436.777892873599
```

Training

To train, we use our typical training loop to optimize all weights of the model by gradient descent. The transformer model receives as input batches of French and English sentence pairs. The encoder processes the French sentence globally, and the decoder processes the English sentence causally (only looking at previous tokens) while simultaneously attending to the encoder output. The transformer finally outputs a prediction if the next word, at each point in the translation. The model is trained to optimize the CrossEntropy loss (over the English dictionary) using the Adam optimizer.

In [31]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import numpy as np
import tqdm


def showPlot(points):
    plt.figure()
    fig, ax = plt.subplots()
    # this locator puts ticks at regular intervals
    loc = ticker.MultipleLocator(base=0.2)
    ax.yaxis.set_major_locator(loc)
```

```python
    plt.plot(points)


def train_epoch(dataloader, transformer, optimizer, criterion):
    transformer.train()

    total_loss = 0
    for fr_tensor, en_tensor in dataloader:
        fr_past = fr_tensor[:, :-1]
        fr_target = fr_tensor[:, 1:]
        # print(en_past.shape, en_target.shape, fr_tensor.shape)
        en_past = en_tensor[:, :-1]
        en_target = en_tensor[:, 1:]

        preds = transformer(en_past, fr_tensor)

        loss = criterion(
            preds.flatten(0, 1),
            en_target.flatten(0, 1)
        )

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(dataloader)

def train(train_dataloader, transformer, optimizer, n_epochs,
          print_every=5, plot_every=1):
    plot_losses = []

    criterion = nn.CrossEntropyLoss()

    for epoch in tqdm.tqdm(range(n_epochs), desc='[TRAIN]'):
        loss = train_epoch(train_dataloader, transformer, optimizer, criteri

        if epoch % print_every == 0:
            te_loss, te_perplexity = evaluate_one_epoch(transformer)
            print(f'[Epoch={epoch}/{n_epochs}] Training Loss={loss:.4f}. Tes
            translate_randomly(transformer, n=3)

        if epoch % plot_every == 0:
            plot_losses.append(loss)

    showPlot(plot_losses)
```

In [32]:

```python
epochs = 30
batch_size = 128
num_layers = 2
learning_rate = 0.001
weight_decay = 0.0005
```

```
train_dataloader, test_dataloader = create_dataloaders(batch_size)

transformer = Transformer(num_layers=num_layers, d_model=256, num_heads=8, d
                          input_vocab_size=fr_vocab.n_words,
                          target_vocab_size=en_vocab.n_words).to(device)

optimizer = optim.Adam(transformer.parameters(), lr=learning_rate, weight_de

train(train_dataloader, transformer, optimizer, epochs)
```

```
[EVAL]: 100%|██████████| 100/100 [00:01<00:00, 51.40it/s]
[TRAIN]:   3%|▎         | 1/30 [00:26<12:50, 26.55s/it]
[Epoch=0/30] Training Loss=2.5170. Test Loss = 5.3079. Test Perplexity = 30
9.37
> je ne suis plus en colere apres toi
= i m no longer angry at you
< i m not a good

> il a la cinquantaine
= he s in his fifties
< i m not a good

> je vais au lit
= i m going to bed
< i m not a good

[TRAIN]:   3%|▎         | 1/30 [00:39<19:13, 39.77s/it]
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[32], line 15
      9 transformer = Transformer(num_layers=num_layers, d_model=256, num_he
ads=8, dff=4,
     10                           input_vocab_size=fr_vocab.n_words,
     11                           target_vocab_size=en_vocab.n_words).to(dev
ice)
     13 optimizer = optim.Adam(transformer.parameters(), lr=learning_rate, w
eight_decay=weight_decay)
---> 15 train(train_dataloader, transformer, optimizer, epochs)

Cell In[31], line 50, in train(train_dataloader, transformer, optimizer, n_e
pochs, print_every, plot_every)
     47 criterion = nn.CrossEntropyLoss()
     49 for epoch in tqdm.tqdm(range(n_epochs), desc='[TRAIN]'):
---> 50     loss = train_epoch(train_dataloader, transformer, optimizer, cri
terion)
     52     if epoch % print_every == 0:
     53         te_loss, te_perplexity = evaluate_one_epoch(transformer)

Cell In[31], line 28, in train_epoch(dataloader, transformer, optimizer, cri
terion)
     25 en_past = en_tensor[:, :-1]
     26 en_target = en_tensor[:, 1:]
```

```
---> 28 preds = transformer(en_past, fr_tensor)
     30 loss = criterion(
     31     preds.flatten(0, 1),
     32     en_target.flatten(0, 1)
     33 )
     35 optimizer.zero_grad()

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/torch/nn/modules/module.py:1532, in Module._wrapped_call_impl(self,
*args, **kwargs)
   1530     return self._compiled_call_impl(*args, **kwargs)  # type: ignor
e[misc]
   1531 else:
-> 1532     return self._call_impl(*args, **kwargs)

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/torch/nn/modules/module.py:1541, in Module._call_impl(self, *args,
**kwargs)
   1536 # If we don't have any hooks, we want to skip the rest of the logic
in
   1537 # this function, and just call forward.
   1538 if not (self._backward_hooks or self._backward_pre_hooks or self._fo
rward_hooks or self._forward_pre_hooks
   1539         or _global_backward_pre_hooks or _global_backward_hooks
   1540         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1541     return forward_call(*args, **kwargs)
   1543 try:
   1544     result = None

Cell In[27], line 22, in Transformer.forward(self, x, context)
     19 context = self.encoder(context)  # (batch_size, context_len, d_mode
l)
     21 # Processes the predictions using a causal decoder.
---> 22 x = self.decoder(x, context)  # (batch_size, target_len, d_model)
     24 # Predicts the next token using a final linear layer classifier.
     25 logits = self.final_layer(x)  # (batch_size, target_len, target_voca
b_size)

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/torch/nn/modules/module.py:1532, in Module._wrapped_call_impl(self,
*args, **kwargs)
   1530     return self._compiled_call_impl(*args, **kwargs)  # type: ignor
e[misc]
   1531 else:
-> 1532     return self._call_impl(*args, **kwargs)

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/torch/nn/modules/module.py:1541, in Module._call_impl(self, *args,
**kwargs)
   1536 # If we don't have any hooks, we want to skip the rest of the logic
in
   1537 # this function, and just call forward.
   1538 if not (self._backward_hooks or self._backward_pre_hooks or self._fo
```

```
rward_hooks or self._forward_pre_hooks
   1539         or _global_backward_pre_hooks or _global_backward_hooks
   1540         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1541     return forward_call(*args, **kwargs)
   1543 try:
   1544     result = None

Cell In[26], line 52, in Decoder.forward(self, x, context)
     49 x = self.dropout(x)
     51 for i in range(self.num_layers):
---> 52     x  = self.dec_layers[i](x, context)
     54 self.last_attn_scores = self.dec_layers[-1].last_attn_scores
     56 # The shape of x is (batch_size, target_seq_len, d_model).

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/torch/nn/modules/module.py:1532, in Module._wrapped_call_impl(self,
*args, **kwargs)
   1530     return self._compiled_call_impl(*args, **kwargs)  # type: ignor
e[misc]
   1531 else:
-> 1532     return self._call_impl(*args, **kwargs)

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/torch/nn/modules/module.py:1541, in Module._call_impl(self, *args,
**kwargs)
   1536 # If we don't have any hooks, we want to skip the rest of the logic
in
   1537 # this function, and just call forward.
   1538 if not (self._backward_hooks or self._backward_pre_hooks or self._fo
rward_hooks or self._forward_pre_hooks
   1539         or _global_backward_pre_hooks or _global_backward_hooks
   1540         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1541     return forward_call(*args, **kwargs)
   1543 try:
   1544     result = None

Cell In[26], line 19, in DecoderLayer.forward(self, x, context)
     17 def forward(self, x, context):
     18     x = self.causal_self_attention(x=x)
---> 19     x = self.cross_attention(x=x, context=context)
     20     x = self.ffn(x)  # Shape `(batch_size, seq_len, d_model)`.
     22     # Cache the last attention scores for plotting later

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/torch/nn/modules/module.py:1532, in Module._wrapped_call_impl(self,
*args, **kwargs)
   1530     return self._compiled_call_impl(*args, **kwargs)  # type: ignor
e[misc]
   1531 else:
-> 1532     return self._call_impl(*args, **kwargs)

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/torch/nn/modules/module.py:1541, in Module._call_impl(self, *args,
```

```
**kwargs)
   1536 # If we don't have any hooks, we want to skip the rest of the logic
in
   1537 # this function, and just call forward.
   1538 if not (self._backward_hooks or self._backward_pre_hooks or self._fo
rward_hooks or self._forward_pre_hooks
   1539         or _global_backward_pre_hooks or _global_backward_hooks
   1540         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1541     return forward_call(*args, **kwargs)
   1543 try:
   1544     result = None

Cell In[10], line 8, in CrossAttention.forward(self, x, context)
      7 def forward(self, x, context):
----> 8     attn_output, attn_scores = self.mha.forward(
      9         query=x, key=context, value=context)
     10     x = self.add_norm(x, attn_output)
     12     # Cache the attention scores for plotting later.

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/torch/nn/modules/activation.py:1266, in MultiheadAttention.forward(
self, query, key, value, key_padding_mask, need_weights, attn_mask, average_
attn_weights, is_causal)
   1252     attn_output, attn_output_weights = F.multi_head_attention_forwar
d(
   1253         query, key, value, self.embed_dim, self.num_heads,
   1254         self.in_proj_weight, self.in_proj_bias,
   (...)
   1263         average_attn_weights=average_attn_weights,
   1264         is_causal=is_causal)
   1265 else:
-> 1266     attn_output, attn_output_weights = F.multi_head_attention_forwar
d(
   1267         query, key, value, self.embed_dim, self.num_heads,
   1268         self.in_proj_weight, self.in_proj_bias,
   1269         self.bias_k, self.bias_v, self.add_zero_attn,
   1270         self.dropout, self.out_proj.weight, self.out_proj.bias,
   1271         training=self.training,
   1272         key_padding_mask=key_padding_mask,
   1273         need_weights=need_weights,
   1274         attn_mask=attn_mask,
   1275         average_attn_weights=average_attn_weights,
   1276         is_causal=is_causal)
   1277 if self.batch_first and is_batched:
   1278     return attn_output.transpose(1, 0), attn_output_weights

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/torch/nn/functional.py:5469, in multi_head_attention_forward(query,
key, value, embed_dim_to_check, num_heads, in_proj_weight, in_proj_bias, bia
s_k, bias_v, add_zero_attn, dropout_p, out_proj_weight, out_proj_bias, train
ing, key_padding_mask, need_weights, attn_mask, use_separate_proj_weight, q_
proj_weight, k_proj_weight, v_proj_weight, static_k, static_v, average_attn_
weights, is_causal)
```

```
   5467         attn_output_weights = torch.baddbmm(attn_mask, q_scaled, k.trans
pose(-2, -1))
   5468 else:
-> 5469         attn_output_weights = torch.bmm(q_scaled, k.transpose(-2, -1))
   5470 attn_output_weights = softmax(attn_output_weights, dim=-1)
   5471 if dropout_p > 0.0:

KeyboardInterrupt:
```

In [ ]:

Visualizing Attention

A useful property of the attention mechanism is its highly interpretable outputs. Because it is used to weight specific encoder outputs of the input sequence, we can imagine looking where the network is focused most at each time step.

You could simply run `plt.matshow(attentions)` to see attention output displayed as a matrix. For a better viewing experience we will do the extra work of adding axes and labels:

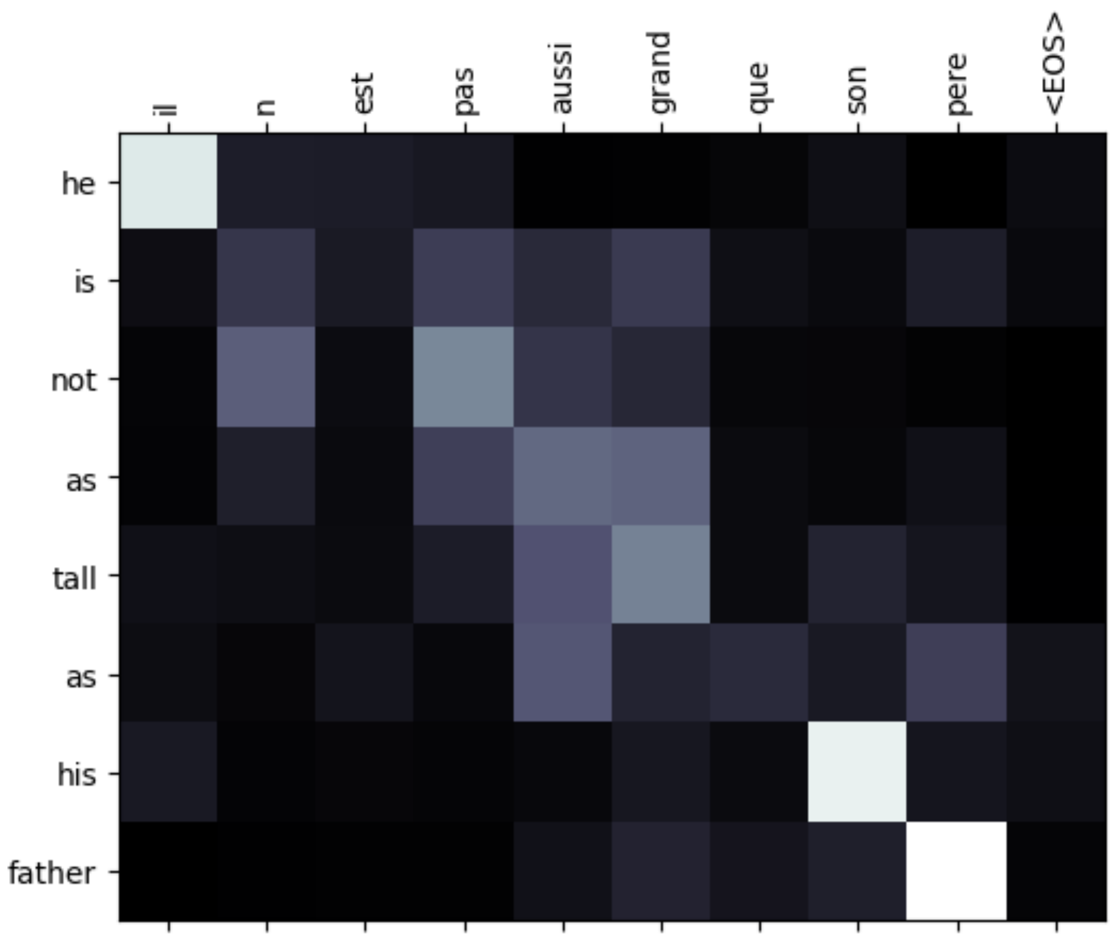
In [ ]:

```
%matplotlib inline
def showAttention(input_sentence, output_words, attentions):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.matshow(attentions.cpu().numpy(), cmap='bone')

    # Set up axes
    ax.set_xticklabels([''] + input_sentence.split(' ') +
                       ['<EOS>'], rotation=90)
    ax.set_yticklabels([''] + output_words)

    # # Show label at every tick
    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    plt.show()


def evaluateAndShowAttention(input_sentence):
    output_sentence, _, _ = evaluate(transformer, input_sentence)
    attention_scores = transformer.decoder.last_attn_scores
    print("="*30)
    print('input =', input_sentence)
    print('output =', output_sentence)
    showAttention(input_sentence, output_sentence.split(), attention_scores[


evaluateAndShowAttention('il n est pas aussi grand que son pere')

evaluateAndShowAttention('je suis trop fatigue pour conduire')
```

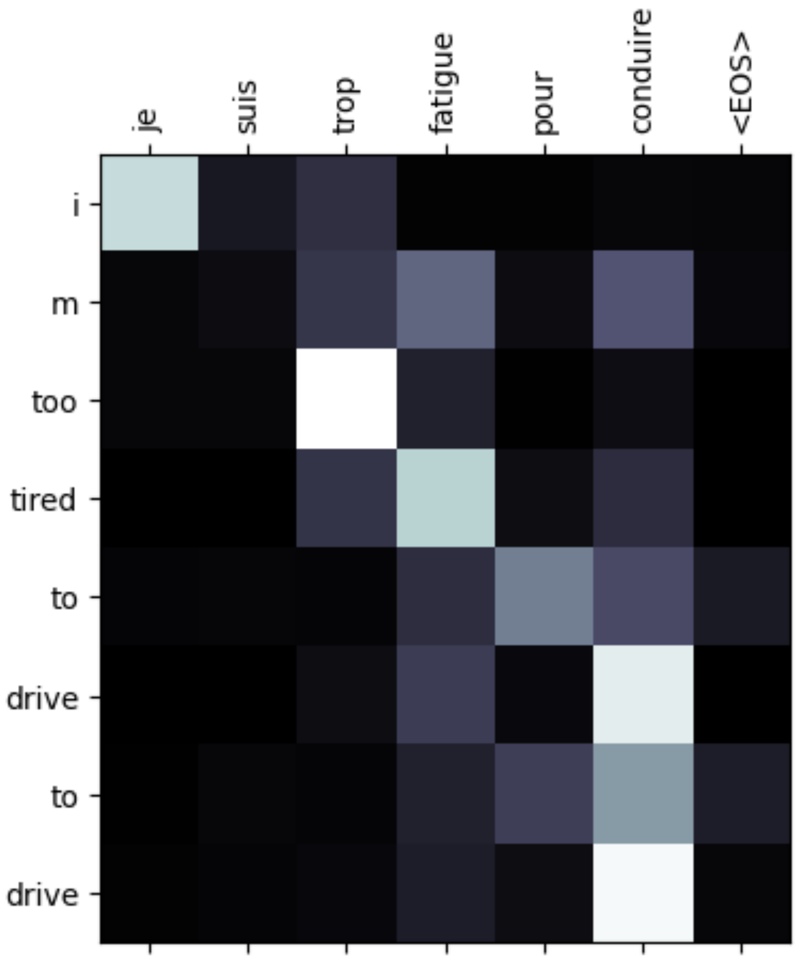
```
evaluateAndShowAttention('je suis desole si c est une question idiote')

evaluateAndShowAttention('je suis reellement fiere de vous')
```

```
==============================
input = il n est pas aussi grand que son pere
output = he is not as tall as his father
```

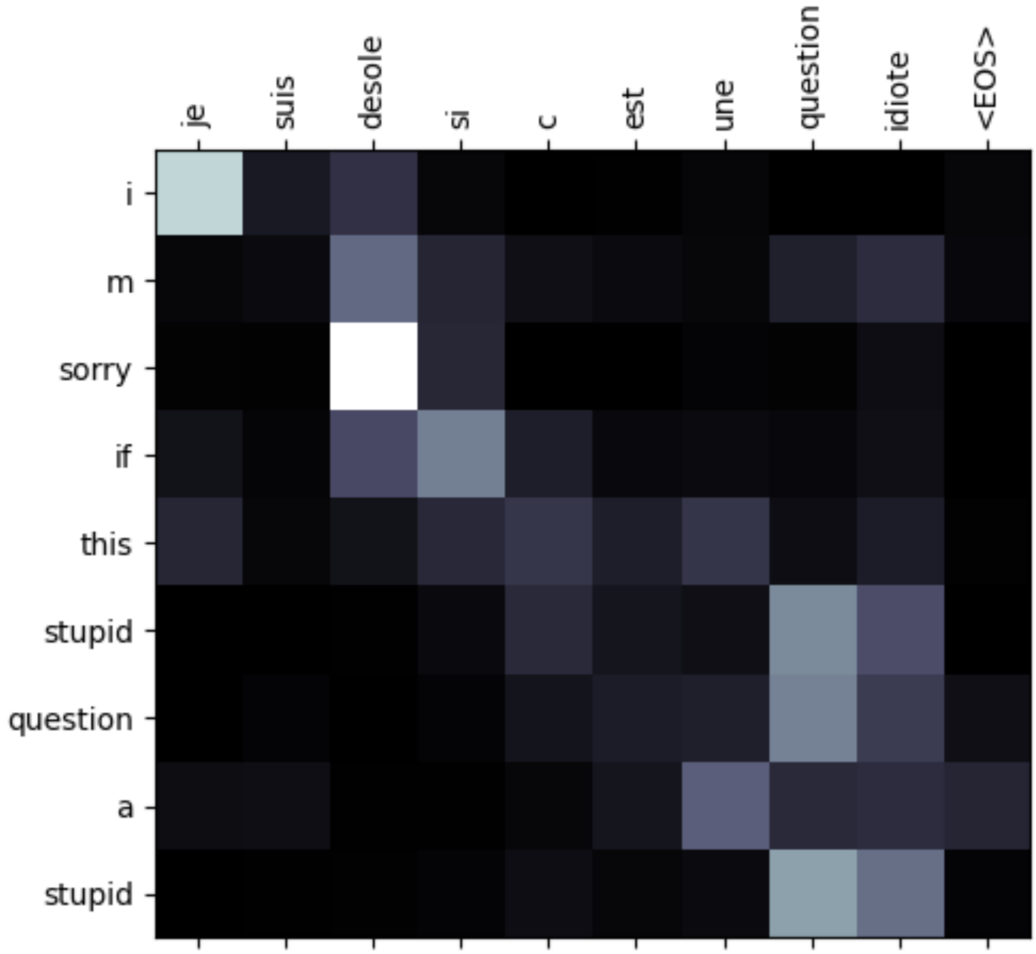
```
<ipython-input-39-117c288cf561>:8: UserWarning: FixedFormatter should only b
e used together with FixedLocator
  ax.set_xticklabels([''] + input_sentence.split(' ') +
<ipython-input-39-117c288cf561>:10: UserWarning: FixedFormatter should only
be used together with FixedLocator
  ax.set_yticklabels([''] + output_words)
```



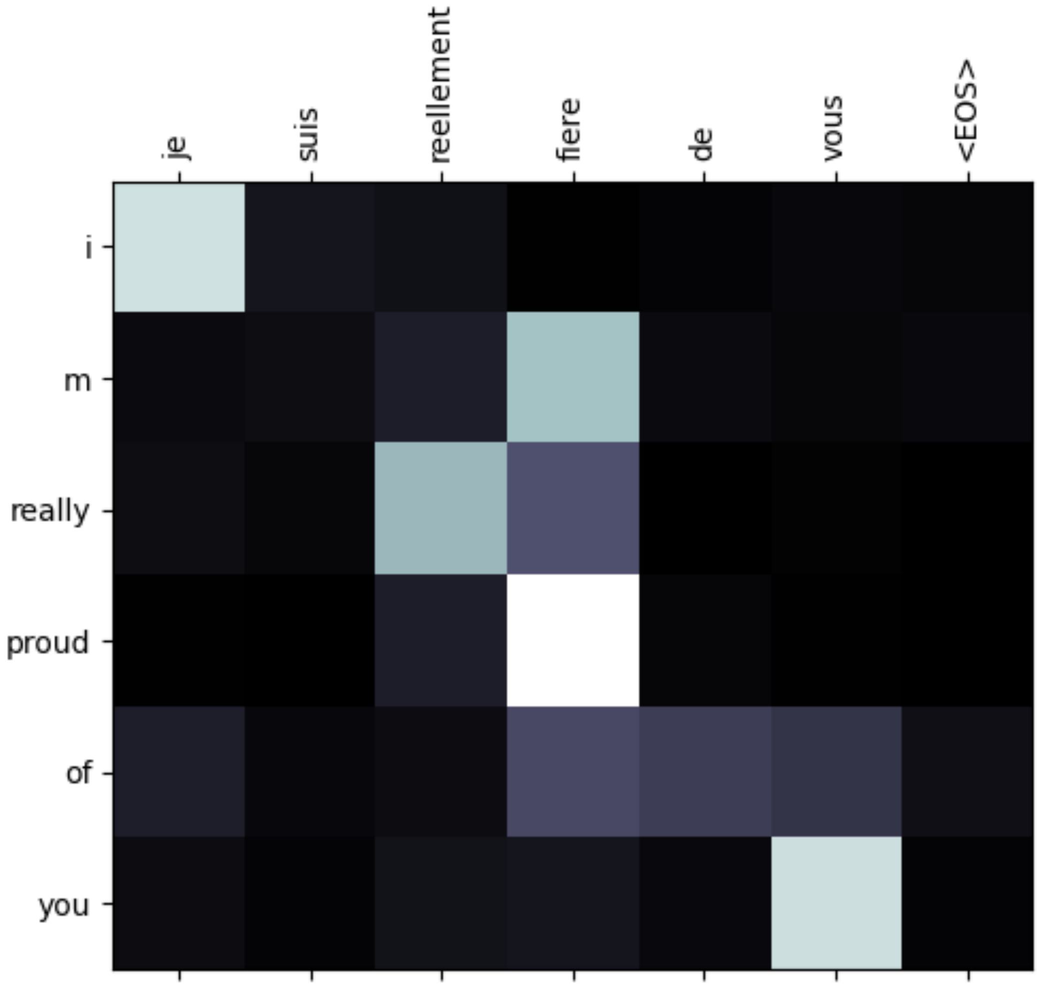

```
==============================
input = je suis trop fatigue pour conduire
output = i m too tired to drive to drive
```




```
==============================
input = je suis desole si c est une question idiote
output = i m sorry if this stupid question a stupid
```




```
==============================
input = je suis reellement fiere de vous
output = i m really proud of you
```



In [ ]:

```
en_vocab.list2sentence(en_vocab.tokenize("i am going to do well on the next
```

Out[ ]:

```
'SOS i am going to do well on the next test EOS'
```

In [ ]:

```

```