# Lab6 Report:

## German-to-English Translation with Attention-Mechanism Transformer Model

In [1]:
```
%matplotlib inline

import numpy as np
import math
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from typing import Iterable, List
from timeit import default_timer as timer

import torch
import torch.nn as nn
from torch.nn import Transformer
from torch import Tensor

from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence

np.random.seed(0)
torch.manual_seed(0)

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

```
cuda
```

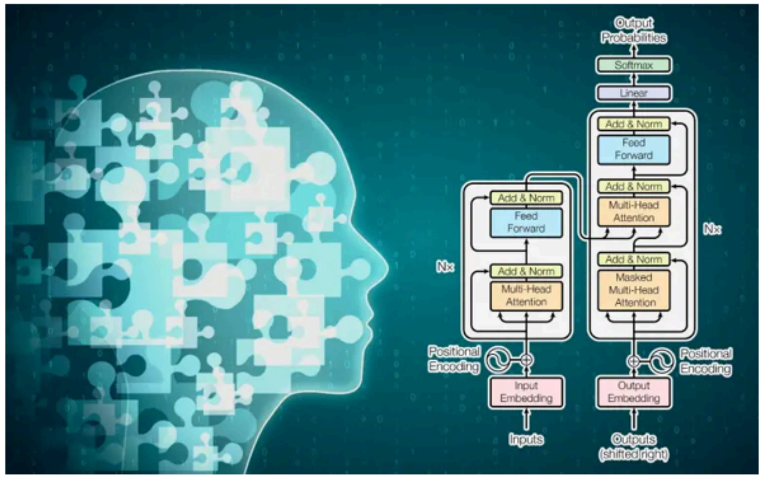
```
C:\Users\Jeff\anaconda3\envs\myenv\lib\site-packages\torchtext\data\__init__.py:4: UserWarning:
/!\ IMPORTANT WARNING ABOUT TORCHTEXT STATUS /!\
Torchtext is deprecated and the last released version will be 0.18 (this one). You can silence this warning by calling the foll
owing at the beginnign of your scripts: `import torchtext; torchtext.disable_torchtext_deprecation_warning()`
  warnings.warn(torchtext._TORCHTEXT_DEPRECATION_MSG)
C:\Users\Jeff\anaconda3\envs\myenv\lib\site-packages\torchtext\vocab\__init__.py:4: UserWarning:
/!\ IMPORTANT WARNING ABOUT TORCHTEXT STATUS /!\
Torchtext is deprecated and the last released version will be 0.18 (this one). You can silence this warning by calling the foll
owing at the beginnign of your scripts: `import torchtext; torchtext.disable_torchtext_deprecation_warning()`
  warnings.warn(torchtext._TORCHTEXT_DEPRECATION_MSG)
C:\Users\Jeff\anaconda3\envs\myenv\lib\site-packages\torchtext\utils.py:4: UserWarning:
/!\ IMPORTANT WARNING ABOUT TORCHTEXT STATUS /!\
Torchtext is deprecated and the last released version will be 0.18 (this one). You can silence this warning by calling the foll
owing at the beginnign of your scripts: `import torchtext; torchtext.disable_torchtext_deprecation_warning()`
  warnings.warn(torchtext._TORCHTEXT_DEPRECATION_MSG)
```

In [2]:
```python
from IPython.display import Image # For displaying images in colab jupyter cell
```

In [3]:
```python
Image('lab6_exercise.png', width = 1000)
```

Out[3]:

Language Translation with Attention-Mechanism Transformer



In this exercise, you will use a Sequence-to-Sequence Transformer model to translate German sentences into English. Your goal is to receive an average validation (Cross-Entropy) loss of less than 2.4.

Before training, it is important to properly tokenize the data. We have worked with language data in the past, but this time the data is more complex and higher-dimensional, and so is the tokenization process. See the example lab for an in-depth look at the tokenization.

During training, we employ the same sliding-window method used in Labs 5 and 6.

After training, demonstrate your model's translation ability by comparing a few of its outputs with the ground-truth labels, along with the inputs.

In [4]:
```
# Seaborn plot styling
sns.set(style = 'white', font_scale = 2)
```

## Download data

In [5]:
```python
# Load the data and store it as a list of tuples: each element in the list should be a tuple of the form (german_sentence, eng
# YOUR CODE HERE
# Load data as a list of tuples
with open("de_to_en.txt", "r", encoding="utf-8") as f:
    lines = f.read().split("\n")

text_pairs = []
for l in lines:
    appendage = l.split("\t")
    text_pairs.append(appendage)
```

## Let's see what the data looks like

In [6]:
```python
# Print the first ten translated lines
# YOUR CODE HERE
# Print the first ten samples
for i in range(10):
    print(text_pairs[i][0], text_pairs[i][1])
```

```
Zwei junge weiße Männer sind im Freien in der Nähe vieler Büsche. Two young, White males are outside near many bushes.
Mehrere Männer mit Schutzhelmen bedienen ein Antriebsradsystem. Several men in hard hats are operating a giant pulley system.
Ein kleines Mädchen klettert in ein Spielhaus aus Holz. A little girl climbing into a wooden playhouse.
Ein Mann in einem blauen Hemd steht auf einer Leiter und putzt ein Fenster. A man in a blue shirt is standing on a ladder clean
ing a window.
Zwei Männer stehen am Herd und bereiten Essen zu. Two men are at the stove preparing food.
Ein Mann in grün hält eine Gitarre, während der andere Mann sein Hemd ansieht. A man in green holds a guitar while the other ma
n observes his shirt.
Ein Mann lächelt einen ausgestopften Löwen an. A man is smiling at a stuffed lion
Ein schickes Mädchen spricht mit dem Handy während sie langsam die Straße entlangschwebt. A trendy girl talking on her cellphon
e while gliding slowly down the street.
Eine Frau mit einer großen Geldbörse geht an einem Tor vorbei. A woman with a large purse is walking by a gate.
Jungen tanzen mitten in der Nacht auf Pfosten. Boys dancing on poles in the middle of the night.
```

## Create source and target language tokenizers

But first, what exactly is a *tokenizer*?

A short but incomplete summary is that a tokenizer converts your text/string into a list of numerical values (a list of *tokens*). We performed tokenization in Lab 5 when we converted each alphanumeric character in our text into a number (an index in a dictionary).

Here, the tokenization is a bit different. Instead of converting each *character* into a number, we want to convert each *word* into a number. As you can imagine, this means the vocabulary of our dataset - the set of unique tokens it contains - will be much larger. There are many more words in English than there are letters! This also means the value of each token will be more unique and meaningful.

Part of tokenizing at the word level is the process of standardizing the text by converting it to lowercase, removing punctuation or special characters, and dealing with contractions or other language-specific features. This is sometimes called *stemming*, reflecting the fact that we want to only extract the *essential meaning* of each word - the "stem" - not necessarily the punctuation, prefixes, suffixes, etc. surrounding it.

Luckily for us, there are some existing Python packages that do this automatically. The below cell downloads two different tokenizers (one each for the source and larget languages), and assigns them to appropriate keys within the "token_transform" dictionary. In Lab 5, you performed tokenization when you converted each character of the text into a specific number.

In [7]:
```
# Define MACRO - a high-level variable that won't change throughout the duration of the code - for your source and target lang
# YOUR CODE HERE
SRC_LANGUAGE = 'de'
TGT_LANGUAGE = 'en'

# Download the German and English tokenizers, and assign them to appropriate keys in your token_transform dictionary
# YOUR CODE HERE

token_transform = {}
token_transform[SRC_LANGUAGE] = get_tokenizer('spacy', language='de_core_news_sm')
token_transform[TGT_LANGUAGE] = get_tokenizer('spacy', language='en_core_web_sm')
```

## Let's see what these specific tokenizers do.

In [8]:
```
# Tokenize the first line of each dataset, and print the tokenized version of it
# YOUR CODE HERE
# Grab the first example
```

```
de_example, en_example = text_pairs[0]

# Tokenize using your SpaCy-based tokenizers
de_tokens = token_transform[SRC_LANGUAGE](de_example)
en_tokens = token_transform[TGT_LANGUAGE](en_example)

# Print results
print("Original German: ", de_example)
print("Tokenized German:", de_tokens)
print()
print("Original English: ", en_example)
print("Tokenized English:", en_tokens)
```

```
Original German:  Zwei junge weiße Männer sind im Freien in der Nähe vieler Büsche.
Tokenized German: ['Zwei', 'junge', 'weiße', 'Männer', 'sind', 'im', 'Freien', 'in', 'der', 'Nähe', 'vieler', 'Büsche', '.']

Original English:  Two young, White males are outside near many bushes.
Tokenized English: ['Two', 'young', ',', 'White', 'males', 'are', 'outside', 'near', 'many', 'bushes', '.']
```

## Create a vocabulary for each language's dataset

In Lab 5, we did this with a simple dictionary that mapped each character to an integer (and vice versa). However, PyTorch has a built-in dictionary object that provides some extra functionality.

We will create this object using torchtext.data.build_vocab_from_iterator(). This function takes an iterator as input and returns a torchtext.vocab.Vocab object. This is a dictionary-like object that maps tokens to indices, but where it differs from a normal dictionary, is that its indices are assigned based on the frequency of the tokens in the dataset. For example, the most frequent token gets the index 0, the second most frequent gets the index 1, and so on. This frequency-index mapping saves a bunch of compute time and resources.

Moreover, this time we will also have the four "special" tokens, that will always be assigned to the first four indices.

In [9]:
```
# Define a helper function that converts a list of strings into a list of lists-of-tokens
# YOUR CODE HERE
def yield_tokens(data_iter, language):
    language_index = {SRC_LANGUAGE: 0, TGT_LANGUAGE: 1}
    for data_sample in data_iter:
        try:
          yield token_transform[language](data_sample[language_index[language]])
        except IndexError:
          print(f"token_transform.keys(): {token_transform.keys()}")
          print(f"language: {language}")
          print(f"data_sample: {data_sample}")
          print(f"language_index: {language_index}")
          raise IndexError

# Define your special tokens and their indeces in your vocabulary
# YOUR CODE HERE
special_tokens = ['<unk>', '<pad>', '<bos>', '<eos>']
UNK_IDX, PAD_IDX, BOS_IDX, EOS_IDX = 0, 1, 2, 3

# Define your vocabulary for each language using the build_vocab_from_iterator function
# YOUR CODE HERE
vocab_transform = {}
for ln in [SRC_LANGUAGE, TGT_LANGUAGE]:
    # Invoke torchtext's Vocab object
    vocab_transform[ln] = build_vocab_from_iterator(yield_tokens(text_pairs, ln),
                                                    min_freq=1,
                                                    specials=special_tokens,
                                                    special_first=True)

# Set ``UNK_IDX`` as the default index.
# YOUR CODE HERE
for ln in [SRC_LANGUAGE, TGT_LANGUAGE]:
  vocab_transform[ln].set_default_index(UNK_IDX)

# Let's see the first 20 words in each vocabulary
 # YOUR CODE HERE
print(vocab_transform[SRC_LANGUAGE].get_itos()[:20])
print(vocab_transform[TGT_LANGUAGE].get_itos()[:20])
```

```
['<unk>', '<pad>', '<bos>', '<eos>', '.', 'Ein', 'einem', 'in', 'und', ',', 'mit', 'auf', 'Mann', 'einer', 'Eine', 'ein', 'de
r', 'Frau', 'eine', 'die']
['<unk>', '<pad>', '<bos>', '<eos>', 'a', '.', 'A', 'in', 'the', 'on', 'is', 'and', 'man', 'of', 'with', ',', 'woman', 'are',
'to', 'Two']
```

## Train-Validate-Test split

In [10]:
```
# Shuffle the text pairs
# YOUR CODE HERE
shuffler = np.random.permutation(len(text_pairs))
text_pairs = [text_pairs[i] for i in shuffler]

# Let's go for a 70-20-10 train-val-test split

# Now I'm using 80-10-10, which performs better
# YOUR CODE HERE
n_train = int(0.8*len(text_pairs))
train_pairs = text_pairs[:n_train]

n_val = int(0.1*len(text_pairs))
val_pairs = text_pairs[n_train:n_train+n_val]

n_test = int(0.1*len(text_pairs))
test_pairs = text_pairs[n_train+n_val:]

# Check the size of each data set
# YOUR CODE HERE

print(f"{len(text_pairs)} total pairs")
print(f"{len(train_pairs)} training pairs")
print(f"{len(val_pairs)} validation pairs")
print(f"{len(test_pairs)} test pairs")
```

```
31019 total pairs
24815 training pairs
3101 validation pairs
3103 test pairs
```

## Mask functions

The mask function plays an essential role in the training of a transformer model, specifically during the pre-training phase when the model learns to understand and generate language. The two main purposes of the mask function are:

1. To facilitate self-attention mechanism: Transformers use self-attention mechanisms to identify relationships between words in a sequence. Masking is used to prevent the model from "cheating" by looking at future tokens when trying to predict the current token. In other words, the mask function ensures that the model only attends to the current token and the previous tokens, not the future tokens, during the training process.

2. To enable masked language modeling (MLM): Masked language modeling is a popular pre-training objective used in transformer-based models like BERT. In MLM, a certain percentage of input tokens are randomly masked (usually around 15%), and the model is tasked with predicting the original tokens at these masked positions. The mask function serves as a way of hiding the original token from the model, forcing it to learn contextual representations that can help it predict the masked tokens accurately.

The use of the mask function in both self-attention and MLM helps the transformer model learn meaningful context-dependent representations, making it more effective at understanding and generating natural language.

In [11]:
```python
# Define your masking function
# YOUR CODE HERE
# Helper
def generate_square_subsequent_mask(sz):
    """
    Create a square attention mask of shape (sz, sz) that masks out
    all positions *after* the current position (i.e., future tokens).
    """
    # Start with an upper-triangular matrix of ones
    mask = torch.triu(torch.ones((sz, sz), device=DEVICE)) == 1
    # Transpose so that mask[i, j] is True when j <= i (allow attending to current and past)
    mask = mask.transpose(0, 1)
    # Convert boolean mask to float: 0.0 where True (allowed), -inf where False (masked)
    mask = mask.float() \
               .masked_fill(mask == 0, float('-inf')) \
               .masked_fill(mask == 1, float(0.0))
    return mask

# Mask creation helper
def create_mask(src, tgt):
    """
    Given source and target sequences, produce:
      1. src_mask            – no masking (all zeros) for src→src attention
      2. tgt_mask            - square subsequent mask for tgt→tgt attention
      3. src_padding_mask    - mask padding tokens in source
      4. tgt_padding_mask    - mask padding tokens in target
    """
    src_seq_len = src.shape[0]
    tgt_seq_len = tgt.shape[0]

    # Mask out future positions in the target sequence
    tgt_mask = generate_square_subsequent_mask(tgt_seq_len)
    # No masking needed for source (allow full self-attention)
    src_mask = torch.zeros((src_seq_len, src_seq_len), device=DEVICE).type(torch.bool)

    # Create padding masks: True at padding positions, shape (batch, seq_len)
    src_padding_mask = (src == PAD_IDX).transpose(0, 1)
    tgt_padding_mask = (tgt == PAD_IDX).transpose(0, 1)

    return src_mask, tgt_mask, src_padding_mask, tgt_padding_mask
```

## Collation

The collation function is what converts our strings into batches of tensors that can be processed by our model, based on the vocabularies and tokenization functions we have built up thus far.

Again, this is something we can do manually, but at some point the data transformations get so complicated that we might as well put them all into a function. Moreover, defining our transformation as a *function* allows us to use some more built-in PyTorch functionality that makes our jobs a whole lot easier. See: torch.utils.data.DataLoader.

In [12]:
```
# Define helper function to club together sequential operations
# YOUR CODE HERE
def sequential_transforms(*transforms):
    def func(txt_input):
        for transform in transforms:
            txt_input = transform(txt_input)
        return txt_input
    return func

# Define function to add BOS/EOS and create a tensor for input sequence indices
```

```python
# YOUR CODE HERE
def tensor_transform(token_ids):
    return torch.cat((torch.tensor([BOS_IDX]),
                      torch.tensor(token_ids),
                      torch.tensor([EOS_IDX])))

# Define your ``src`` and ``tgt`` language text transforms to convert raw strings into tensors indices
# YOUR CODE HERE
text_transform = {}
for ln in [SRC_LANGUAGE, TGT_LANGUAGE]:
    text_transform[ln] = sequential_transforms(token_transform[ln], #Tokenization
                                               vocab_transform[ln], #Numericalization
                                               tensor_transform) # Add BOS/EOS and create tensor

# Define your "collation" function to collate data samples into batch tensors
# YOUR CODE HERE
def collate_fn(batch):
    src_batch, tgt_batch = [], []
    for src_sample, tgt_sample in batch:
        src_batch.append(text_transform[SRC_LANGUAGE](src_sample.rstrip("\n")))
        tgt_batch.append(text_transform[TGT_LANGUAGE](tgt_sample.rstrip("\n")))

    src_batch = pad_sequence(src_batch, padding_value=PAD_IDX)
    tgt_batch = pad_sequence(tgt_batch, padding_value=PAD_IDX)
    return src_batch, tgt_batch
```

## Define training, evaluation functions

Modularization is the name of the game.

Not only does this help us here, but if you ever need to train a similar model in the future, you can simply import the ones defined here!

For example, imagine this was a Python script and not a notebook, and the filename was "german_to_english_transformer.py" Then, in whichever future script or notebook you wish to use these functions, you could simply call: "from german_to_english_transformer import train_epoch, evaluate"

In [13]:
```
# Define a function to train the model for a single epoch
# YOUR CODE HERE
def train_epoch(model, optimizer):
    """
    Run one training epoch over the entire training dataset.
    Returns a list of loss values for each batch.
    """
    # Set the model to training mode (enables dropout, batch-norm updates, etc.)
    model.train()
    loss_list = []

    # Prepare the training iterator and DataLoader
    train_iter = train_pairs
    train_dataloader = DataLoader(
        train_iter,
        batch_size=BATCH_SIZE,
        collate_fn=collate_fn
    )

    # Iterate over batches
    for src, tgt in train_dataloader:
        # Move source and target tensors to the computing device (GPU/CPU)
        src = src.to(DEVICE)
        tgt = tgt.to(DEVICE)

        # For teacher forcing, feed all except the last token as decoder input
        tgt_input = tgt[:-1, :]

        # Generate masks for source and target sequences
        #  - src_mask: no causal mask (allow full src→src attention)
        #  - tgt_mask: prevents attending to future tokens in target
        #  - src_padding_mask / tgt_padding_mask: masks out padding tokens
        src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = create_mask(src, tgt_input)

        # Forward pass: compute raw logits over vocabulary
        # Note: during training, the model will use the masks to ignore padded positions
        # and prevent illegal attention connections in the decoder.
        logits = model(
            src,
            tgt_input,
```

```python
            src_mask,
            tgt_mask,
            src_padding_mask,
            tgt_padding_mask,
            src_padding_mask
        )

        # Zero out any previously computed gradients
        optimizer.zero_grad()

        # The ground-truth labels are the target sequence shifted by one
        tgt_out = tgt[1:, :]

        # Compute cross-entropy loss:
        # reshape logits to (batch*seq_len, vocab_size) and targets to (batch*seq_len)
        loss = loss_fn(
            logits.reshape(-1, logits.shape[-1]),
            tgt_out.reshape(-1)
        )

        # Backpropagate gradients
        loss.backward()

        # Update model parameters
        optimizer.step()

        # Record this batch’s loss value
        loss_list.append(loss.item())

    # Return the list of batch losses for monitoring
    return loss_list

# Define a function to evaluate the model
# YOUR CODE HERE
def evaluate(model):
    """
    Evaluate the model on the validation dataset.
    Returns a list of loss values for each batch (without performing any weight updates).
    """
    # Switch model to evaluation mode (disables dropout, batch-norm updates, etc.)
    model.eval()
    loss_list = []

    # Prepare the validation iterator and DataLoader
    val_iter = val_pairs
    val_dataloader = DataLoader(
        val_iter,
        batch_size=BATCH_SIZE,
        collate_fn=collate_fn
    )

    # Iterate over validation batches
    for src, tgt in val_dataloader:
        # Move source and target tensors to the computing device
        src = src.to(DEVICE)
        tgt = tgt.to(DEVICE)

        # Prepare decoder input by removing the last token
        tgt_input = tgt[:-1, :]

        # Generate masks for source and target sequences
        #  - src_mask: no causal restriction on source
        #  - tgt_mask: prevents the decoder from looking ahead in target
        #  - src_padding_mask / tgt_padding_mask: masks out padding tokens
        src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = create_mask(src, tgt_input)

        # Forward pass (no gradient computation)
        logits = model(
            src,
            tgt_input,
            src_mask,
            tgt_mask,
            src_padding_mask,
            tgt_padding_mask,
            src_padding_mask
        )

        # The ground-truth labels are the target sequence shifted by one
        tgt_out = tgt[1:, :]

        # Compute loss for this batch
        # Reshape logits to (batch*seq_len, vocab_size) and targets to (batch*seq_len)
        loss = loss_fn(
            logits.reshape(-1, logits.shape[-1]),
            tgt_out.reshape(-1)
        )

        # Record the loss value (no backward/optimizer step in evaluation)
        loss_list.append(loss.item())

    # Return the list of batch losses for validation monitoring
    return loss_list
```

## Define model

In [14]:

```
# Define the PositionalEncoding module that quantifies the relative position of words in a sentence
# Notice that this is not actually an MLP or neural network, i.e. it has no learned parameters
# it is just a function that you could represent analytically, if you wanted to
# YOUR CODE HERE
class PositionalEncoding(nn.Module):
    """
    Implements the fixed sinusoidal positional encoding described in
    “Attention Is All You Need”. Adds position information to token embeddings.
    """
    def __init__(self, emb_size: int, dropout: float, maxlen: int = 5000):
        super(PositionalEncoding, self).__init__()
        # Compute the division term (denominator) for the exponent:
        #  exp(-2i * log(10000) / emb_size) for even indices
        den = torch.exp(-torch.arange(0, emb_size, 2) * math.log(10000) / emb_size)
        # Create a (maxlen × 1) tensor of position indices [0, 1, 2, …]
        pos = torch.arange(0, maxlen).reshape(maxlen, 1)
        # Initialize positional embedding matrix (maxlen × emb_size)
        pos_embedding = torch.zeros((maxlen, emb_size))
        # Apply sine to even dimensions
        pos_embedding[:, 0::2] = torch.sin(pos * den)
        # Apply cosine to odd dimensions
        pos_embedding[:, 1::2] = torch.cos(pos * den)
        # Add a singleton dimension for broadcasting over batch
        pos_embedding = pos_embedding.unsqueeze(-2)

        # Dropout layer for regularization
        self.dropout = nn.Dropout(dropout)
        # Register pos_embedding as a buffer (non-parameter tensor, saved with the model)
        self.register_buffer('pos_embedding', pos_embedding)

    def forward(self, token_embedding: Tensor) -> Tensor:
        """
        Add positional encoding to token embeddings and apply dropout.

        Args:
            token_embedding: Tensor of shape (seq_len, batch_size, emb_size)

        Returns:
            Tensor of same shape with position information added.
        """
        # Slice the positional embeddings to match the input sequence length
        seq_len = token_embedding.size(0)
        # Add position embeddings and apply dropout
        return self.dropout(token_embedding + self.pos_embedding[:seq_len, :])

# Define the TokenEmbedding module converts a tensor of vocabulary-indices into a tensor of token-embeddings
# Also not a neural network, but a lookup table
# YOUR CODE HERE
class TokenEmbedding(nn.Module):
    def __init__(self, vocab_size, emb_size):
        super(TokenEmbedding, self).__init__()
        self.embedding = nn.Embedding(vocab_size, emb_size)
        self.emb_size = emb_size

    def forward(self, tokens: Tensor):
        return self.embedding(tokens.long()) * math.sqrt(self.emb_size)

# Define the actual transformer model
# Question: What are we "transforming" between?
# YOUR CODE HERE
class Seq2SeqTransformer(nn.Module):
    """
    A sequence-to-sequence Transformer for machine translation (or similar tasks),
    consisting of an encoder and decoder stack plus a final linear generator.
    """
    def __init__(self,
                 num_encoder_layers: int,
                 num_decoder_layers: int,
                 embedding_size: int,
                 num_heads: int,
                 src_vocab_size: int,
                 tgt_vocab_size: int,
                 dim_feedforward: int = 512,
                 dropout: float = 0.1):
        super(Seq2SeqTransformer, self).__init__()
        # Core Transformer module
        self.transformer = Transformer(
            d_model=embedding_size,
            nhead=num_heads,
            num_encoder_layers=num_encoder_layers,
            num_decoder_layers=num_decoder_layers,
            dim_feedforward=dim_feedforward,
            dropout=dropout
        )
        # Final linear layer to project decoder outputs to target vocab logits
        self.generator = nn.Linear(embedding_size, tgt_vocab_size)
        # Token embedding layers for source and target
        self.src_tok_emb = TokenEmbedding(src_vocab_size, embedding_size)
        self.tgt_tok_emb = TokenEmbedding(tgt_vocab_size, embedding_size)
        # Positional encoding to add order information
        self.positional_encoding = PositionalEncoding(embedding_size, dropout=dropout)

    def forward(self,
                src: Tensor,
                trg: Tensor,
                src_mask: Tensor,
                tgt_mask: Tensor,
                src_padding_mask: Tensor,
                tgt_padding_mask: Tensor,
                memory_key_padding_mask: Tensor) -> Tensor:
        """
        Execute full forward pass: embed inputs, apply Transformer,
        and map to vocabulary distribution.
        """
        # Embed source tokens and add positional encodings
        src_emb = self.positional_encoding(self.src_tok_emb(src))
        # Embed target tokens and add positional encodings
        tgt_emb = self.positional_encoding(self.tgt_tok_emb(trg))
        # Run through encoder-decoder stacks
        transformer_out = self.transformer(
            src_emb,                        # (S, N, E)
            tgt_emb,                        # (T, N, E)
            src_mask,                       # (S, S)
            tgt_mask,                       # (T, T)
            memory_mask=None,               # not used here
            src_key_padding_mask=src_padding_mask,      # (N, S)
            tgt_key_padding_mask=tgt_padding_mask,      # (N, T)
            memory_key_padding_mask=memory_key_padding_mask  # (N, S)
        )
        # Project to target vocabulary
        return self.generator(transformer_out)

    def encode(self, src: Tensor, src_mask: Tensor) -> Tensor:
        """
        Encode source sequence only (returns memory for decoder).
        """
        src_emb = self.positional_encoding(self.src_tok_emb(src))
        return self.transformer.encoder(src_emb, src_mask)

    def decode(self, tgt: Tensor, memory: Tensor, tgt_mask: Tensor) -> Tensor:
        """
        Decode target sequence step-by-step given encoder memory.
        """
        tgt_emb = self.positional_encoding(self.tgt_tok_emb(tgt))
        return self.transformer.decoder(tgt_emb, memory, tgt_mask)
```

## Question #2: What's the significance of the "num_heads" parameter in the **init** function of the Seq2SeqTransformer above?

My answer:

The num_heads parameter specifies how many parallel attention mechanisms the model uses, splitting the embedding dimension into that many subspaces so it can attend to different aspects of the input simultaneously.

## Question #3: In less detail, state the significance of these other two parameters:

1. embedding_size
2. src_vocab_size

My answer:

`embedding_size` determines the dimensionality of each token’s embedding and the model’s internal representations (i.e. the size of the vectors it uses to represent words). `src_vocab_size` tells the model how many unique source-language tokens there are, so it knows how large to make its input embedding lookup table.

## Define hyperparameters

In [15]:
```
# Define your hyperparameters
# YOUR CODE HERE

SRC_VOCAB_SIZE = len(vocab_transform[SRC_LANGUAGE])
TGT_VOCAB_SIZE = len(vocab_transform[TGT_LANGUAGE])
EMB_SIZE = 512
NUM_HEADS = 8 # Why 8? What do you expect to happen if we increase this parameter?
FFN_HID_DIM = 1024
BATCH_SIZE = 64
NUM_ENCODER_LAYERS = 3
NUM_DECODER_LAYERS = 3
NUM_EPOCHS = 10

# Define your model, loss function, and optimizer
# YOUR CODE HERE
transformer = Seq2SeqTransformer(NUM_ENCODER_LAYERS, NUM_DECODER_LAYERS, EMB_SIZE,
                                 NUM_HEADS, SRC_VOCAB_SIZE, TGT_VOCAB_SIZE, FFN_HID_DIM).to(DEVICE)

for p in transformer.parameters():
    if p.dim() > 1:
        nn.init.xavier_uniform_(p)

transformer = transformer.to(DEVICE)
```

```
C:\Users\Jeff\anaconda3\envs\myenv\lib\site-packages\torch\nn\modules\transformer.py:306: UserWarning: enable_nested_tensor is True, but self.use_nested_tensor is False because encoder_layer.self_attn.batch_first was not True(use batch_first for better inference performance)
  warnings.warn(f"enable_nested_tensor is True, but self.use_nested_tensor is False because {why_not_sparsity_fast_path}")
```

## Identify tracked values

In [16]:
```
# YOUR CODE HERE
loss_fn = torch.nn.CrossEntropyLoss(ignore_index=PAD_IDX)

optimizer = torch.optim.Adam(transformer.parameters(), lr=0.0001, betas=(0.9, 0.98), eps=1e-9)
```

## Train the model

In [17]:
```
# Train your model
# YOUR CODE HERE
train_loss_list = []
val_loss_list = []
for epoch in range(1, NUM_EPOCHS+1):
    start_time = timer()
    train_loss = train_epoch(transformer, optimizer)
    train_loss_list.extend(train_loss)
    end_time = timer()
    val_loss = evaluate(transformer)
    val_loss_list.extend(val_loss)
    print((f"Epoch: {epoch}, Epoch time = {(end_time - start_time):.3f}s"))


# Fair warning: you might get an "out of memory" error
# If that happens, try reducing the batch size
```

```
C:\Users\Jeff\anaconda3\envs\myenv\lib\site-packages\torch\nn\functional.py:5504: UserWarning: 1Torch was not compiled with fla
sh attention. (Triggered internally at C:\cb\pytorch_1000000000000\work\aten\src\ATen\native\transformers\cuda\sdp_utils.cpp:45
5.)
  attn_output = scaled_dot_product_attention(q, k, v, attn_mask, dropout_p, is_causal)
C:\Users\Jeff\anaconda3\envs\myenv\lib\site-packages\torch\nn\functional.py:5137: UserWarning: Support for mismatched key_paddi
ng_mask and attn_mask is deprecated. Use same type for both instead.
  warnings.warn(
Epoch: 1, Epoch time = 24.541s
Epoch: 2, Epoch time = 24.332s
Epoch: 3, Epoch time = 24.205s
Epoch: 4, Epoch time = 24.259s
Epoch: 5, Epoch time = 24.154s
Epoch: 6, Epoch time = 24.303s
Epoch: 7, Epoch time = 24.230s
Epoch: 8, Epoch time = 24.848s
Epoch: 9, Epoch time = 24.162s
Epoch: 10, Epoch time = 24.464s
```

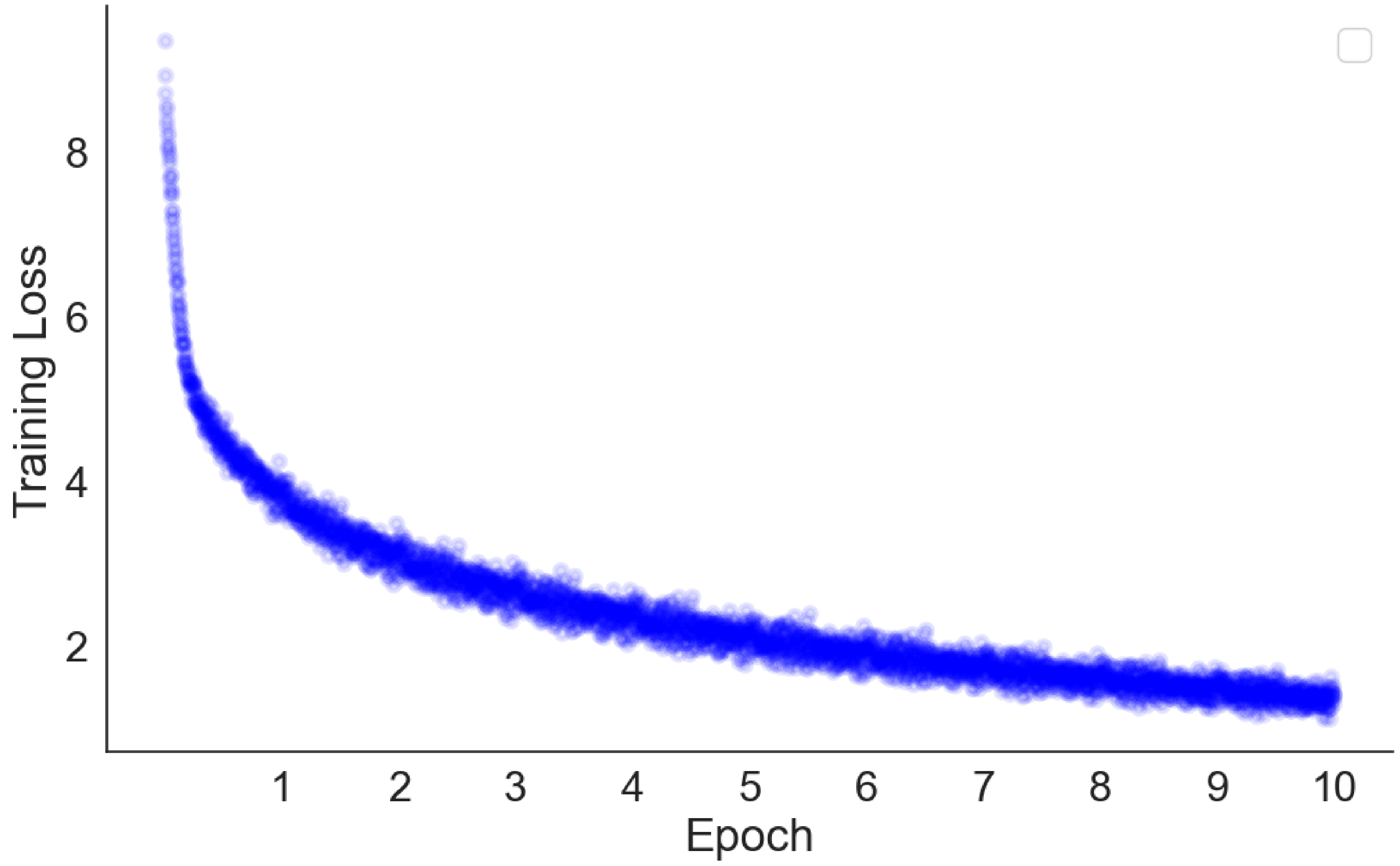
## Visualize and Evaluate the model

In [18]:

```
# Plot the loss
# YOUR CODE HERE
plt.figure(figsize = (12, 7))

plt.scatter(range(len(train_loss_list)), train_loss_list, color = 'blue', linewidth = 3, alpha=0.1)
plt.ylabel("Training Loss")
plt.xlabel("Epoch")
plt.xticks(ticks = [(i+1)*len(train_loss_list)//NUM_EPOCHS for i in range(NUM_EPOCHS)], labels=[f"{i+1}" for i in range(NUM_EP
plt.legend()
sns.despine()
```

```
C:\Users\Jeff\AppData\Local\Temp\ipykernel_10620\2432237141.py:9: UserWarning: No artists with labels found to put in legend.
Note that artists whose label start with an underscore are ignored when legend() is called with no argument.
  plt.legend()
```



In [19]:
```python
# Define a decode function to generate output sequence using greedy algorithm
# This basically saves us some compute time by taking a bunch of shortcuts (e.g. not computing the full softmax)
# YOUR CODE HERE
def greedy_decode(model, src, src_mask, max_len, start_symbol):
    """
    Generate a target sequence token-by-token using greedy decoding.
    Args:
        model       : The Seq2SeqTransformer model
        src         : Source tensor of shape (S, 1)
        src_mask    : Source square mask of shape (S, S)
        max_len     : Maximum length of the generated target sequence
        start_symbol: Token index marking the beginning of sequence (BOS)

    Returns:
        ys: Tensor of generated token indices of shape (T, 1)
    """
    # Move source and mask to the appropriate device
    src = src.to(DEVICE)
    src_mask = src_mask.to(DEVICE)

    # Encode the source sequence to obtain memory for the decoder
    memory = model.encode(src, src_mask)

    # Initialize the output sequence with the start symbol
    ys = torch.ones(1, 1, dtype=torch.long, device=DEVICE).fill_(start_symbol)

    # Iteratively decode one token at a time
    for i in range(max_len - 1):
        # Ensure memory is on the correct device
        memory = memory.to(DEVICE)

        # Create a causal mask for the current target length (prevents looking ahead)
        tgt_mask = generate_square_subsequent_mask(ys.size(0)).to(DEVICE).bool()

        # Decode using the current partial target sequence and encoder memory
        out = model.decode(ys, memory, tgt_mask)        # shape: (T, 1, E)
        out = out.transpose(0, 1)                        # shape: (1, T, E)

        # Project the last decoder output to vocabulary logits
        prob = model.generator(out[:, -1, :])            # shape: (1, V)

        # Pick the token with highest logit (greedy)
        _, next_word = torch.max(prob, dim=1)
        next_word = next_word.item()

        # Append the predicted token to the growing target sequence
        ys = torch.cat([
            ys,
            torch.ones(1, 1, dtype=src.dtype, device=DEVICE).fill_(next_word)
        ], dim=0)

        # Stop if end-of-sequence token is produced
        if next_word == EOS_IDX:
            break

    return ys


def translate(model, src_sentence):
    """
    Translate a raw source sentence string into the target language text.

    Args:
        model       : The Seq2SeqTransformer model
        src_sentence: A string in the source language

    Returns:
        A single string of the translated target sentence.
    """
    # Put model in evaluation mode (disable dropout)
    model.eval()

    # Tokenize & numericalize the source sentence, shape (S, 1)
    src = text_transform[SRC_LANGUAGE](src_sentence).view(-1, 1)
    num_tokens = src.size(0)

    # No causal mask for the encoder (all positions visible)
    src_mask = torch.zeros(num_tokens, num_tokens, dtype=torch.bool, device=DEVICE)

    # Perform greedy decoding to obtain a sequence of token indices
    tgt_tokens = greedy_decode(
        model,
        src,
        src_mask,
        max_len=num_tokens + 5,     # allow a few extra tokens beyond source length
        start_symbol=BOS_IDX
    ).flatten()
    # Convert token indices back to strings, remove special tokens
    tokens = vocab_transform[TGT_LANGUAGE].lookup_tokens(tgt_tokens.cpu().numpy())
    translation = " ".join(tokens).replace("<bos>", "").replace("<eos>", "")

    return translation
```

## Let's try the model out on a few of our test sequences. Print the first 10 target/translated sequences from our test set

In [20]:
```
# YOUR CODE HERE
for i in range(10):
    test_pair = test_pairs[-i]
    test_str_de = test_pair[0]
    test_str_en = test_pair[1]
    print("Target_de: ", "\n", test_str_de)
    print()
    print("Target: ","\n", test_str_en)
    print("Model output: ")
    print(translate(transformer, test_str_de))
    print()
```

```
Target_de:
 Eine Gruppe von Einkäufern trotzt der Kälte, während einer auf sein Lieblingsgeschäft zeigt.

Target:
 A group of faithful shoppers bear the cold while one points to his favorite store.
Model output:
 A group of military workers are interacting with the same time as he points to his shoulder .

Target_de:
 Ich glaube, hier werden Bauarbeiten ausgeführt.

Target:
 I think the construction work is going on here.
Model output:
 I of the clown is being played construction .

Target_de:
 Ein Motorradfahrer rast mit einem grünen Kawasaki Sportmotorrad über eine Straße.

Target:
 A motorcyclist speeding along a road on a green Kawasaki sport motorcycle.
Model output:
 A motorcycle rider is racing through a street covered in green blouse .

Target_de:
 Skaterboy macht ein Kunststück und wird in der Luft fotografiert.

Target:
 Skater boy does a trick and gets his photo taken in midair.
Model output:
 A female does a trick while being photographed in the air .

Target_de:
 Kletterer und Wanderer in der Nähe eines Sees.

Target:
 People rock climbing and hiking near a lake.
Model output:
 A rock climber and hikers near a lake .

Target_de:
 Ein Mann und Frau sind in der Küche und ein Paket Challenge-Butter steht im Vordergrund.

Target:
 A man and woman are in the kitchen and a package of Challenge butter is in the foreground.
Model output:
 A man and woman in the kitchen with a hula hoops stands in the foreground .

Target_de:
 Ein hellhäutiger Mann schwimmt in einem Schwimmbecken auf vielen Schwimmhilfen.

Target:
 A white man is seen in a swimming pool floating above many floating devises.
Model output:
 A white man swimming in a pool with many toys .

Target_de:
 Eine beleuchtete Brücke mit einem Radfahrer und ein paar Autos.

Target:
 A streetlight lit bridge with a cyclist and some cars.
Model output:
 A group of people are gathered around a bridge with a few cars .

Target_de:
 Menschen laufen Rennen und der Mann in Grau führt.

Target:
 People running a race and the guy in gray is in the lead.
Model output:
 People are running and the man in gray performing .

Target_de:
 Ein Snowboarder, der eine orange Jacke und eine himmelblaue Tasche trägt, folgt einer bereits vorhandenen Spur.

Target:
 Snowboarder wearing an orange jacket and carrying a sky-blue bag snowboards toward a previously traveled trail.
Model output:
 A snowboarder wearing an orange jacket and a lot of flowers is carrying a good time .
```

```
In [ ]:
```