# Assignment 4: Transformer Language Models

In this assignment you will implement a transformer-based language model (decoder-only transformer), in PyTorch.

You will implement the different attention layers yourself (you are not allowed to use the pre-made transformer-layers).

In order to get fast development turn-around, we will be working with small-ish models, that can run on a CPU or on a free GPU, and train in a quick time (a few hours or less). This means a character-level model, with few layers and few heads, which also translates to few parameters (about 3M parameters). But the same architecture can work also with significantly more parameters.

## PyTorch etc

We will be working with PyTorch multi-dimensional tensors, and a large part of the work is in creating, combining, splitting and re-arranging their dimensions. Here are some functions that you may find useful:

- torch.ones
- torch.zeros
- torch.triu
- torch.tril
- torch.cat
- torch.split
- torch.range
- torch.arange
- torch.Tensor.view
- torch.Tensor.transpose
- torch.Tensor.permute

## Part 1: Implementing attention

### Overview

The attention implementation is in the file `attention.py`. It is a skeleton implementation, which you will have to fill up.

Start by looking at the `self_attention_layer` function. It takes as input the input vectors `x`, a `kqv_matrix` which are parameters for computing K, Q and V for the given x, and an `attention_mask` which is used to mask part of the attention matrix for making the attention causal. It then uses this to compute `sa`, the self-attention values, which are returned as output. In short, this functions maps from `x` to `sa`.

The input to the layer, `x`, is a variable-length sequence of vectors.
The output of the layer, `sa`, is a sequence of vectors of the same length and same dimensionality.

The function, like all our code, supports batches, so in practice the dimensions of `x` and `sa` are `b x n x d`, where `b` is the batch dimension, `n` is the sequence length, and `d` is the dimension of each vector.

The layer computes `k`, `q` and `v`, then uses `k` and `q` to compute the attention matrix, and finally applies the (masked) attention matrix to `v` in order to compute the self attention `sa`.

We now implement the individual components which this function calls.

## Parameters vs. Implementations

All our values will be PyTorch components (`torch.nn.Module`).
PyTorch separates the creation of network components (which lives inside the module class constructors) and their application (which lives in the `forward` function). We will do the same in our code. So, for each function like `kqv` which takes as input some parameters, we will also create a function for creating these parameters. These will then be called from the layer's constructor (for parameter creation) and `forward` (for the computation).

## Queries, Keys and Values

The first stage is to compute the Queries, Keys and Value. Recall that each input vector $x_i$ is transformed to three vectors $q_i$, $k_i$, and $v_i$, each resulting from a different linear transformation (with different linear parameters) applied to $x_i$.

This is simple to do, but for efficiency we want to:
(a) compute the values for all x_i in a single linear transformation (matrix

multiplication).
(b) instead of computing each of k, q and v in a separate operation, we will do them in a single operation as well.

After you figure out how to do so, you need to implement the functions `create_kqv_parameters(...)` and `kqv(...)`. Their skeletons appear in the file, but you need to provide the implementation.

You are advised to test your implementation of these functions in isolation, before proceeding.

## Attention Scores

Now, compute the attention scores. The attention scores matrix for two sequences of vectors, $a$ of length $n$ and $b$ of length $m$, is a $m \times n$ matrix where the $ij$ index is the score between an item in $a$ and the corresponding item in $b$. The score we will be using follows the "Attention is All you Need" paper, and is the dot-product of the vectors, divided by the square-root of their dimension. As usual, the sequences $a$ and $b$ will be provided as matrices, and we support batching on the first dimension.

### Step 1: Implement a test

Implement the function `test_attention_scores` in `tests.py`, with a simple but convincing set of input and output examples. Note, the skeleton includes space for a single input-output pair, but you are welcome to implement more than a single test case.

### Step 2: Implement the attention scores computation

Now implement the attention score computation so it passes the test. Do so in the `attention_scores()` skeleton function in `attention.py`.

Verify that your implementation passes your tests.

## Self-attention

### Step 1: full self-attention

In a full self-attention, each token attends to all other tokens.
That is, given sequence positions $x_1, \ldots, x_n$ and corresponding values $k_1, \ldots, k_n$, $q_1, \ldots, q_n$, and $v_1, \ldots, v_n$ , the value for position $x_i$ is given by:

$y_i = \sum_{1 \leq j \leq n} a_{ij} v_j$

Here, $a_{ij}$ is score between $q_i$ and $k_j$ (which we have stored in the attention-scores matrix). In other words, each output vector $y_i$ is a weighted sum of all the value vectors $v_j$.

Given the attention scores matrix, all the $y_i$ vectors can be computed in a single matrix multiplication (think how, you will do it soon).

One important detail is that the scores $a_{ij}$ should be normalized such that for each $i$, all the values $a_{ij}$ form a probability distribution (are positive and sum to one). In other words, all the coefficents of the different $v_j$ in a combination are positive and sum to one. If you figured our the matrix-multiplication above, you will realize this can be easily achieved by applying the `softmax` function (`torch.nn.functional.softmax`) to either the rows or the columns (think which one..) of the attention matrix before the multiplication. Do so.

Implement this computation in `self_attention()` in `attention.py`. Ignore the `mask` parameter for now, we will integrate it shortly.

You are highly encouraged to implement also a test function, like you did for the attention scores.

## Step 4: from full self-attention to causal self attention

In our implementation above, each token's attention value is a linear combination of all other token values. However, in a language model setting (and in a transformer-decoder in general), the future tokens are not available, and attention for position i should only be over token $i$ and tokens that appear before it.

In other words, the equation changes to (note we replaced $n$ with $i$ in the sum index):

$y_i = \sum_{1 \leq j \leq i} a_{ij} v_j$

This is called "causal attention". This is easily implemented by setting some values in the attention matrix (those corresponding to future interactions) to zero. This way, they will not participate in the summation. Simple enough.

A small complication is that we want the values to be zero *after* the softmax computation. So, instead of zeroing-out elements, we want to set these elements to $-\infty$ before the softmax computation (because $e^{-\infty} = 0$ ). Given a

tensor $A$ of values and a tensor $M$ (mask) of the same shape in which some values are 0, we can set the corresponding values in $A$ to $-\infty$ using:

```python
A = A.masked_fill(M == 0, float("-inf"))
```

You can go ahead and implement it in your code of `self_attention`.

Another small complication relates to efficiency: we do not want to generate the $M$ tensor from scratch each time, because creating and transferring it to the GPU will be expensive. Instead, we want to generate it before-hand, and re-use whenever needed. However, the tensor $M$ is different in different computations (for example, if the sequence lengths are different). So, what we will do is create a matrix $\tilde{M}$ up-front, from which we can derive $M$ for each individual computation. Since $\tilde{M}$ resides on the GPU and $M$ is only a view of sub-parts of $\tilde{M}$, this will remain efficient.

Implement the creation of $\tilde{M}$ in the function `create_causal_mask`. It should be a 3-dim tensor, where the first dimension is 1. This will be passed to your `self_attention` function as the `mask` parameter. Change your implementation of `self_attention` to take the `mask` matrix, derive $M$ from it, and apply it to the attention scores matrix before computing the softmax.

All of this should be very few lines of code. However, they may be tricky to get right. As before, you are *highly* encouraged to write a test function to verify your implementation.

The implementation of `self_attention_layer` is now complete (you can test it to verify that it works).

## Multi-head Attention

We are almost there, but the transformer's attention is *multi-head*. This means that instead of computing a single attention transformation for a given sequence, we compute $k$ different attention transformations, and then concatenate the $k$ vectors into a single vector. In practice, if the input dimension of each vector is $d$ and the output dimension is also $d$, then we set each the $k$ heads in the multi-head attention to create outputs of dimension $\frac{d}{k}$ (yes, $d$ must be a multiply of $k$), so that when we concatenate the $k$ resulting vectors we get output dimension $d$ again.

We now implement multi-head attention. This is easy: for $k$ heads, we just call `self_attention_layer` $k$ times, each time with smaller `kqv` parameters. At

the end, we concatenate the results (be mindful of the dimensions that you concatenate over).

Implement this in `multi_head_attention_layer`. It has a similar signature to `self_attention_layer` but receives a list of kqv matrices instead of a single one.

There is also a bit trickier (but more efficient) implementation, which you can try out if you want the extra challenge (but no extra points), see comments in the `multi_head_attention_layer` skeleton function if you are interested.

### Final projection:

After concatenating the the output of the attention heads, their outputs are "mixed" using another linear projection (linear layer) from $d$ dimensions to $d$ dimensions. You don't have to implement this part, but it is done in the skeleton code (outside of `multi_head_attention_layer`), see if you can find it.

## The PyTorch layer

We are now done with the causal-self-attention layer! To integrate it into PyTorch, all the code you wrote is called from the `CausalSelfAttention` class in `attention.py`. It is a simple code, see that you understand it.

# Part 2: The Transformer Decoder

We now integrate the causal-attention layer we created as part of a transformer-decoder.

The transformer-decoder is composed of an embedding layer, several transformer blocks (transformer-layers), and finally a vocabulary prediction layer (also called "LM-head", "word-prediction layer" or "unembedding layer").

The overall structure is implemented in the `TransformerLM` class in the `transformer.py` file. The constructor and `forward` functions are complete, see that you understand them. However, this implementation uses the transformer-block and an embedding layer which you have to implement (see below), and also does not do any fancy initialization for the matrix layers, which is left for your to experiment with.

## Transformer-block

We now use the causal-attention layer we created as part of a transformer-layer. A transformer-layer is simply an attention layer (like we now have), followed by position-wise single-layer MLP on each vector in the sequence. There are layer-norm layers before the attention and before the MLP. There are also *residual connections* that bypass the layer-norm+attention and the layer-norm+MLP.

We provided an MLP implementation in `mlp.py`, and, more importantly, an implementation of the transformer-decoder block in the `TransformerDecoderBlock` class in `transformer.py`.

### Your job: the residuals.

The provided decoder block has an implementation without residual connections. **You need to implement the residual connections** in the `forward` function of the `TransformerDecoderBlock` class in `transformer.py`.

## Embeddings

The input to the transformer is vocabulary items, which are then translated to embeddings (as usual). Each vocabulary item will be represented as an integer. There are also `position embeddings`: each embedding is a summation of a vocabulary-embedding and a position embedding vector. The place to implement it is in the `Embed` class in `transformer.py`, fill in the missing details.

## Weight initialization

Weight initialization of the different components can improve training. The place to do it is in the `init_weights()` method of `TransformerLM`. You should play with initializations in the next part.

## Congratulations!

We now have a fully implemented transformer decoder!

# Part 3: Transformer LM

Finally, we can use our transformer as a language model!

Since our compute is very limited, we will be using a small LM setting: character-level language model (that is, predicting the next character in the sequence), trained on small-ish data (up to a million or two characters). These models can have a few millions of parameters, and can train also on a CPU in a reasonable amount of time.

## Skeleton code

We provided skeleton code that runs on a CPU. However, it can be easily adapted to work on GPU (you just need to use the correct "device" when creating tensors, and keeping the tensors on the correct device). This can speed up training considerably, even on relatively weak GPUs such as those in google-colab.

The skeleton code has three parts:

### main.py

This is a very basic training script, that uses the transformer-lm we created as well as the two other parts and pytorch, to train a basic LM. You can run this as `python main.py` to start training. This works, but is very basic. You will need to play with this, see below.

### data.py

This part is in charge of tokenization (translating sequences of characters to sequences of integers and back), data loading, and batching. The code in this file is complete and functioning, but if you want to save and load the tokenizer, you need to provide this code.

You are still highly encouraged to read and understand this code.

### lm.py

These are two functions specific to LM training, that you need to implement (labels generation and loss calculation).

## Labels and Loss

The training loop is in `main.py`, see that you understand its overall structure. It feeds the transformer-LM with labeled training examples, computes the loss, and updates the parameters.

Translating the input data (which is a sequence of vocab items) to labeled data (which are pairs of input-expected-output) is done in `batch_to_labeld_samples` in `lm.py`. Implement this function.

Similarly, you need to implement the loss calculation in `compute_loss` (also in `lm.py`).

## Training

You now have fully working training loop. It prints the loss of the last batch every 10 batches, and prints a sample output every 100 batches.

Train your own model on the provided data in the `data/` folder (the default). These are the works of Shakespeare, which is very intellectually appealing, but also quite boring. Nevertheless, it is a reasonable corpus to experiment with. However, the code will work with any corpus, so feel free to experiment with it and play with your own data!

We also provide `heb-data/`, which is ~1.5 million characters of Bialik and Rachel writings (mostly poems), in Hebrew.

With the provided English corpus, you should start seeing English-looking samples after a while, and you should also get losses below 2.0 after a reasonable time (few hundred batches with the provided parameters), which is also when the samples should start looking like English.

The training code is very basic, and in a more robust implementation you would also save a checkpoint of the model every now and then, as well as be able to load a model and a tokenizer to use it later, continue training from a given checkpoint (maybe with different learning rates), etc. This is not provided, and is not required, but is not hard to add, and you are encouraged to add it (for your convenience).

### Better training

The current model trains, however, it can also train faster / better. LM-training is sensitive to hyper-parameters, and I want you to experiment with them. Can you get the training loss to get lower, or get lower faster? Hack your way around `main.py` and possibly the other `.py` files to achieve this.

You can play with the basic hyper-parameters (number of layers, number of heads, dimensions, etc), but also play with the optimizer settings, as well as adding dropout to the network (places to add dropout can be right after the

embedding layer, right after the attention softmax, and right after the self attention), this dropout should be mild, about 0.1 or so. You can also add other forms of regularization like weight decay. And of course, playing with initialization also matters, and there are hooks for initialization in the provided code. You can also change your learning rate as training progresses. Go at it.

You can also use this to explore questions such as, what is more important? number of layers, number of heads, or the dimensions of the MLP? How important are the residual connections? etc.

### Hebrew Training

What loss do you mange to get for the Hebrew data (`heb-data/`) given the best settings you got for English? Can you get even better loss for Hebrew? How does the loss correlate with output (samples) quality?

## Part 4: Generation

### Better LM sampling

During training, we also sample from examples from the model. We use the `sample_continuation` method in `TransformerLM`. It is very basic, and is missing two key features:

1. supporting top-k sampling (sampling from only the top-k top probability tokens)
2. supporting temperature (softmax with a temperature of $t$ is defined as $\frac{e^{x_i/t}}{\sum_i e^{x_i/t}}$)

Implement both of these in `TransformerLM.better_sample_continuation` and integrate them in your training code (with characters, using topK = 5 works nicely).

## Part 5: Analysis / Interpretability

### Looking for interesting attention patterns

This part provides a glimpse into model analysis and interpretability techniques. Specifically, we will be looking for interpretable attention behavior.

For each input sample, each layer+head results in an N x N attention matrix, where N is the input sample length. These attention patterns are based on the

role of the layer+head combination in the overall computation. Your goal in this part is to analyze these attention patterns and find at least one such layer+head combination whose behavior you can interpret.

For example, is there a layer+head that always look for the previous character? Consonants looking for vowels? The previous space? etc.

To identify these, you want to take a reasonably well-trained LM --- one with a low loss, that generates good-looking texts when sampled from. (you may want to add a save/load functionality to the TransformerLM, so you can train once and then load the pre-trained model for the analysis).

Feed the trained LM with input samples, and store the resulting attention matrices for each sample. Then, start looking at these matrices and the sequences that resulted in them, and look for patterns. It will be useful to visualize them as a heat-map. There are various packages that will allow you to write code to do so. It may also be useful to look at matrices for a given head+layer in aggregate (by position? by letter? something else?). Alternatively, instead of storing or printing the matrices, maybe you can store or print some other statistics or information that will help you discover interesting or interpretable behavior (for example, if you suspect some behavior might exist, you can write code that looks for it). This part is more open ended than the other ones, and requires more creativity.

Describe the attention behavior you found, as well as the way to find it, in a brief report.

## What to submit

Submit all the code files that you edited, as well as any additional file you created. Please do not change the signatures of functions or classes provided in the assignment (though you can add default parameters with sane defaults). Also submit a **short report in pdf format** listing:

1. The loss you managed to get to on the Shakespeare data with the default sequence length. State after how many updates you got to it, how many training sequences, and how many network parameters.
2. What hyperparameters you used to achieve this loss (num layers, dimensions, etc)
3. What other modifications, if any, you did to obtain this loss: initialization, optimizer choices, dropout, anything else...
4. A brief discussion on the Hebrew data results and observations.

5. Your answer to part 5 (Analysis / Interpretability).
6. A brief description of your experience with the project.

## What next?

You implemented a transformer-decoder, and there are various things that can be done with your code (in your free time, if you are interested, not part of this assignment). Here are two of them:

Encoder-decoder: We implemented a decoder-only transformer. The transformers paper discusses an encoder-decoder architecture, where the encoder has full self attention, and the decoder conditions on the encoder. How would you modify/extend your code to support this?

Relative positions: There are various proposed alternatives to static position vectors, for example, using vectors that encode relative positions (you can see an overview of various methods in "Position Information in Transformers: an Overview"). How would you modify your code to support this?