Assignment 3B: Implementing a transformer in PyTorch

Goal: Implement (self) attention in Pytorch and work it into a transformer architecture.

Submission: As before, provide a PDF that directly answers each question, and use the provided report template. This is a *group exercise*. As a result, the amount of work is a little higher, so make sure to collaborate efficiently.

Changelog:
26 Nov. Fix layer numbers in Q2.

In this assignment, we will look at Transformers. As before, answer all explicitly labeled questions in your report.

Note that this assignment is new this year, so you should expect more corrections and clarifications to appear than with the other assignments. Check the changelog regularly, and ask your TA about anything that is unclear.

There are a few additional questions in the text of the exercise but these are meant for reflection only. Discuss them in the text of your report if it makes sense, but otherwise, just make sure you can answer them for yourself.

This exercise involves a lot of hyperparameter tuning. Most of the points are awarded for correct implementation, not for tuning, so we recommend focussing on the implementation in your first pass, and then tuning as much as time allows.

New datasets. Install the the `wget` module in your python environment with the command

```
pip install wget
```

and then add the following dataset script to your code:
https://gist.github.com/pbloem/1e7750850d3e1095e3ccfb3e98e3db77

Like the script of assignment 1, this contains basic functions for loading various datasets.

Part 1: Classification: data loading

The IMDb dataset is the sequence-learning equivalent of MNIST: a large, challenging classification dataset with plenty of examples, that is still light enough to train models for on a laptop. It contains 50 000 reviews of movies, taken from the Internet Movie Database which are either highly positive or highly negative. The task is to predict which for a given review. This is known as *sentiment analysis*.

If this seems like a simple task, note that even large pretrained models like ELMo don't get far over 95% accuracy, so this is by no means a solved or a toy problem.

To simplify things, we've preprocessed and tokenized the data for you. Our tokenization is a little crude: we lowercase everything, remove all non-letters and split on whitespace. It'll do for our current purposes, but in practice, you'd look to more refined tokenization strategies to extract more information from the raw text. All words have been converted to integer indices in a fixed vocabulary.

IMDb-synth and XOR It turns out that the word-level IMDb dataset doesn't show the difference between our baseline and a full transformer very well. To give you something that more clearly distinguishes the two models, we've created a synthetic version with a simple structure. Every instance follows one of the following four patterns with equal probability:

```
this old movie was very great     positive
this old movie was not great      negative
this old movie was very terrible  negative
this old movie was not terrible   positive
```

For the words `old`, `movie` and `was`, alternative words are used randomly, independently of the class label. This means that only the presence of the words `very` and `great` contain any information about the label. The key is that each by themself is as likely to appear in a positive as a negative instance. You need to see both to predict the label correctly.

The structure is similar to the XOR problem with which you may be familiar. Interestingly, in a sequence setting, the XOR problem has a slightly different structure if you represent it like this, with true and false as tokens in two-token sequences.

```
false false    negative
false true     positive
true false     positive
true true      negative
```

The key difference with the above problem, is that we have to assign the same embedding to true and false in both positions (unless we add position embeddings). This makes a difference in which stripped-down transformers can solve IMDb-synth and which can solve XOR. We've included dataset generators for both.

To load the data, call the `load_imdb`, `load_imdb_synth` or `load_xor` function as follows:

```
(x_train, y_train), (x_val, y_val), (i2w, w2i), numcls = load_imdb(final=False)
```

The return values are as follows:

- `x_train` A python list of lists of integers. Each integer represents a word. Sorted from short to long.

- `y_train` The corresponding class labels: 0 for positive, 1 for negative.
- `x_val` Test/validation data. Laid out the same as `x_train`.
- `y_val` Test/validation labels.
- `i2w` A list of strings mapping the integers in the sequences to their original words. `i2w[141]` returns the string containing word 141.
- `w2i` A dictionary mapping the words to their indices. For example `w2i['film']` returns the index for the word "film".

If `final` is true, the function returns the canonical test/train split with 25 000 reviews in each. If `final` is false, a validation split is returned with 20 000 training instances and 5 000 validation instances. For the synthetic datasets, this switch has no effect.

To have a look at your data (always a good idea), you can convert a sequence from indices to words as follows

```
print([i2w[w] for w in x_train[141]])
```

To train, you'll need to loop over `x_train` and `y_train` and slice out batches. Each batch will need to be padded to a fixed length and then converted to a torch tensor. Make sure to review the relevant part of the Sequences lecture.

question 1: Implement this padding and conversion. Show the function in your report.

Tips and comments:

- We've included a special padding token in the vocabulary, represented by the string ".pad". Consult the `w2i` dictionary to see what the index of this token is.
- There are some more special tokens which you can ignore.
- If you feed a list of lists to the function `torch.tensor()`, it'll return a torch tensor.
    - The inner lists must all have the same size
    - Pytorch is pretty good at guessing which datatype (int, float, byte) is expected, but it does sometimes get it wrong. To be sure, add the datatype with
      `batch = torch.tensor(lists, dtype=torch.long)`.
- As we will see later, large transformers are often pre-trained without padding, using a much cruder method of preparing data. However, this only applies to *pre-training*. When you are finetuning for a specific task, understanding padding can still be essential.
- The slides suggest sorting the data by length and using a dynamic batch size. You can do this (it's a useful trick to understand), but it's also fine to use a fixed batch size and to keep the data as is.

Part 2: Classification, baseline model

We'll start with the simplest sequence-to-sequence model discussed in the lectures: a linear layer applied to each token in the sequence separately. This will be our baseline. On the

IMDb data, this is hard to beat, but on the synthetic data, you should be able to work out what its accuracy will be. .

question 2: Build a model with the following structure. Include the model class in your report.

	input	**layer**
1	tensor of `dtype=torch.long` with size (batch, time)	**`nn.Embedding(...)`** Convert the integer indices to embedding vectors
2	tensor of `dtype=torch.long` with size (batch, time, emb)	Global pool along the time dimension See below. This is not a class, you can just implement each pooling operation yourself with a single line in your forward().
3	tensor of `dtype=torch.float` with size (batch, emb)	**`nn.Linear(emb, numcls)`** Project down to the number of classes
output	tensor of `dtype=torch.float` with size (batch, num_clases)	

Notes:

- Make sure to read the documentation for the layers used carefully.
- Use embedding size 300.
- The embedding layer needs to know how many tokens there are (since that is how many embedding vectors it needs to create). How can you find this from the return values of the load_imdb function?
- The model is not memory-intensive, so you can easily go to large batch sizes to speed up training. Note, however, that the amount of padding also increases with the batch size. Can you explain why?
- It's most common not to make softmax part of the model, but to apply it as part of the loss function, since this can be done in a more numerically stable way. The function `torch.nn.functional.cross_entropy` can be used to calculate the loss here. Make sure to read the documentation carefully so that you know what it expects.
    - Note that when computing accuracy, we only need the class with the highest score, so we can just compute the argmax over the linear outputs.
    - A common mistake is to mix up cross entropy, which applies the softmax, and NLLLoss which expects the model to apply a (log) softmax. You end up either not applying a softmax or applying it twice.

Question 3: Implement the model. Implement three different global pools:

- take the **mean** along the time dimension.
- take the **max** along the time dimension.
- **select** only the first element along the time dimension.

Which pool you use is a hyperparameter.

Train for at least one epoch on all three datasets and and compute the validation accuracy. Get an 0.8 accuracy or higher on IMDb within 5 epochs, and report your hyperparameters.

Explain what you expect the performance for IMDb-synth and XOR to be. Does this agree with your experiments?

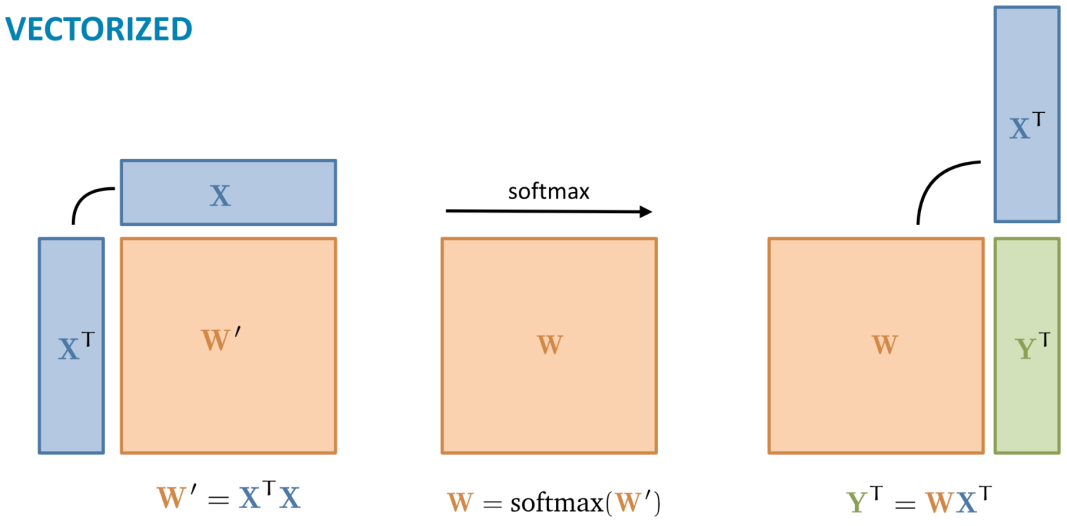
- No points will be deducted if your expectations were incorrect: feel free to be honest.
- Sometimes the XOR model lands on a suboptimal solution. Repeating the run a few times may help to see what the optimal performance is.

Part 3: Self-attention

To solve `IMDb-synth` and `XOR` we'll need to somehow allow the model to "propagate information over the time dimension", as the slides say. That means, the model needs to somehow see whether it has *great* in a sentence together with *not* or with *very*.

We'll build up from simple self-attention and see what we need to solve `IMDb-synth`.

The vectorized self-attention is described in [the following slide]:



Question 4: Copy the baseline model and add one layer of simple-self attention. Change the global mean pool to a global max pool (a max over the time dimension). Put your implementation in the report.

- Note that the image doesn't show the batch dimension.
- You can put the self attention in a separate module, or just write it out in the `forward()`. It can be done in one line per step.
- Don't take the order of the indices too literally. What are rows in the picture may not be the first indices in your tensors. If your embedding vectors are columns, then the operations are the ones shown. You may need to translate this to your order of dimensions.

- If you're familiar with einsum, that's a simple way to implement the first and third steps. You can also use batched matrix multiplication. Make sure to carefully read the documentation on the different matrix multiplications in pytorch to see how batch dimensions are handled.
- Self-attention gets a little slow and memory hungry for long sequences. We recommend setting a maximum sequence length (`256` is a good value) and slicing down any sequences longer than that.

Question 5*: Run the experiments with this model on the three datasets. Tune the hyperparameters for Synth and XOR, but *use only the select pooling*.
Tips:

- Synth is solvable in this setting and XOR isn't. However, it may be difficult to get synth solved with gradient descent
- IMDb won't be different from the baseline, so don't spend too much time tuning that.
- The synth dataset is a bit sensitive to learning rate, so don't give up too soon. As many as a 100 epochs may be necessary.
- When training big models, we usually say that the bigger the batch size the better. For small and very specific problems like these, small batch sizes may work better (probably because they inject a little noise to help us avoid local minima), so you'll need to treat the batch size like a hyperparameter and tune it.
- .75 accuracy should be possible with XOR.
- If you'd like to experiment with automated hyperparameter tuning, optuna is a relatively lightweight option. For the synthetic datasets, this can do quite a few trials in an hour or so, even on your laptop.

Next, we'll add the bells and whistles to our self-attention. In order of complexity, these are:

1. Scaling the activations by $\sqrt{d}$ before the softmax.
2. Passing the input embeddings through three different linear layers to create separate embedding vectors for the key, query and value.
3. Performing the self attention in parallel for k different heads.

For the third part, the key is to realize that you can simply make the heads a separate dimension, and then implement the self attention in such a way that it only looks at the two rightmost dimensions, and *treats the rest as batch dimensions*. In pseudo-code, it looks as follows.

```
given input x with dimensions (batch, time, emb)

k, q, v <- tokeys(x), toqueries(x), tovalues(x)

hemb <- emb // k # the per-head embedding size
reshape k, q and v to (batch, time, hemb, k)
reorder x's dims to (batch, k, time, semb)

y <- apply self-attention to dimensions 3 and 4 (or -2 and -1) of x
assert y.size() == (batch, k, time, time)
```

```
z <- y ⨉ v.T # matrix multiply on the last two dims.
assert z.size() == (batch, k, time, hemb)

reorder z to (batch, time, k, hemb) and reshape to (batch, time, emb)

apply a final linear operation
```

question 6: Translate this to pytorch. Include the implementation in your report.

- Be very careful with the dimension reordering and reshaping. The pseudocode is purposely vague at points, or different from how it looks in pytorch, so that you have to reason through the operations step by step.
- There are different ways to do batched matrix multiplication in pytorch. Read the documentation carefully to see which one you need, and how to apply it so that the right dimensions are matched.
- If you get stuck on this question, try to work out a very simple example with some small matrices on pen and paper (either for just the sizes of the whole matrix). It's a little bit of monkey work, but it can help you see exactly where your implementation is going wrong.

question 7: Re-run the experiments with full self-attention, comparing to simple self attention. You should see a difference between full and simple self-attention on the XOR data with the select pool. Report this, and any other differences you find.

question 8: Add position embeddings to model. Show how you've implemented this in your report.

Tune the simple self-attention model with select pooling on XOR. Show the difference with and without position embeddings.

- Position embeddings can be implemented as a second embedding layer. Instead of passing them the integer indices of the words, you can pass the integers from 0 to L where L is the current sequence length. Then you just add them to the embeddings you already have.
- Be aware of the batch dimension. If the position embeddings don't have a batch dimension, then broadcasting will kick in when you add them to the token embeddings.

question 9*: Turn your self-attention into a transformer block. This means adding layer normalization, a feedforward layer, residual connections and dropout. Put the implementation of the transformer block in your report.

Build a model consisting of 3 blocks, followed by a select pool and train it on IMDb. Tune the hyperparameters to roughly the same performance as the baseline model.

- The feedforward should have two `nn.Linear` modules, with a relu nonlinearity in the middle. The hidden size should be 4 times the size as the input size.
- Note that the self-attention is the only part of the block that propagates information over the time dimension. All the other operations are applied to each input vector independently.
- You can experiment to see if you can get any (significant) performance above the baseline on the IMDb data by making the model bigger, but so far it seems that this is impossible without pre-training. The best we get out of the baseline after much tuning is ... validation accuracy.

Part 4: Autoregressive models

In this final part, we will move from a sequence classification model (sequence-to-label) to an autoregressive model (sequence-to-sequence) in the style of GPT.

The task of this model is to predict the next token in a sequence given all the tokens that precede it.

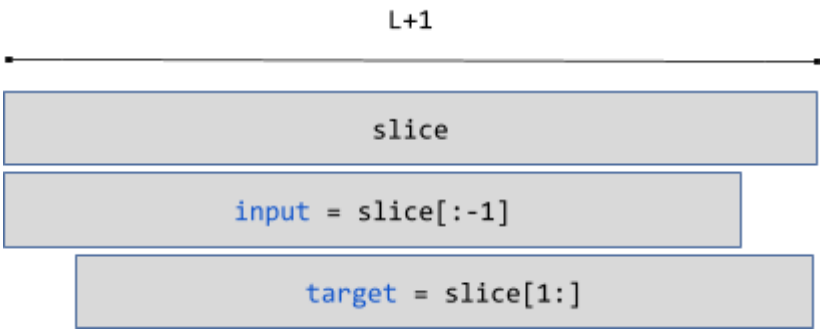
Data The end goal is to train a model that can sample convincing English language. Since this takes a while, we'll treat that as an optional goal and start with English-looking language, generated from a simple toy grammar. You can load this with

```
(train, test), (i2c, c2i) = load_toy(n=150_000)
```

The data is loaded as a single sequence of *characters* (they're actually just two strings), with all sentences simply concatenated. Instead of carefully batching instances of variable length we will just generate instances of length `L` by slicing them from random points in the data, entirely ignoring sentence starts and ends. This means we don't need to pad, and we can always train on batches of size `(batch, L)`.

question 10: Build a function that takes a dataset in the form of an integer tensor, slices out `b` instances of length `l`, and returns a batch of size `(b, L)`.

- The simplest way to do this is with a for loop. This works, but is a bit slow. A faster way is to generate random starting indices between 0 and `N-l` (with `N` the size of the data and `L` the length of the instances), and to slice out the batch in one operation.
- When calling this function later, make sure that this doesn't end up as a node in the computation graph.
- In our training loop, the batch needs to be turned into an input and target output. The output is just the input shifted one character to the left. The simplest way to do this is to sample a batch `x` of length `L+1`, and to slice out `x[:, :-1]` for the input and `x[:, 1:]` for the target.



Next, we'll build our model. This is much the same as the previous model with just two changes. First, we need to remove the global pool and return a vector of probabilities (or logits) at every point in time.

Second, we need to mask our self-attention so that it becomes causal. For this, we must set the attention weights above the diagonal to -∞. One way to do this is with the function triu_indices. Try it out in a notebook or ipython shell first to get a sense of how it works.

If you have the indices for a matrix in two lists `is` and `js`, then you can set those indices to a particular value v across a batch of matrices as follows:

```
matrices[..., is, js] = v
```

Everything but the last two dimensions will be treated as batch dimensions. As ever, reason carefully through all the dimensions. It's possible, depending on your implementation choices, you'll need to mask in a slightly different way from what is shown in the slides.

Note that this is an in-place operation: we're modifying matrices, not getting a new tensor as the result of the operation. These are usually best avoided, since we're building a computation graph (so we need to store the value of `matrices` *before and after the operation). In this case, however, Pytorch seems to handle it fine.*

question 11: Implement this model. Show how you've implemented the main structure of the model (the `forward()` of the main `nn.Module`) and show how you've implemented the masking.

For our loss function, we want to use a cross entropy loss at every point in time. This is possible with the pytorch cross entropy loss function, but if your output tensor has size (batch, time, vocab) and your target (integer) tensor has size (batch, time), you'll need to do some shuffling of dimensions. Read the documentation carefully to figure out the details.

Warning: Some loss functions apply the softmax function for you, and some expect you to do it manually. Read the documentation carefully. Applying softmax twice is a very common bug. We recommend not applying the softmax in the model (that is, outputting *logits*) and using the loss that applies the softmax for you.

With that, you have everything you need for a training loop. Implement it, and check that the loss curve looks good (to make sure that you don't have any obvious bugs).

Next, we want to see how the model is doing. We'll do this in two ways: validation loss and sampling.

Validation loss For the evaluation, we want to see how well the model predicts the next character. The model makes predictions at every point in time, but the best predictions happen at the last token, where the model has a full context to work with. That means that if we want to measure the prediction performance over a validation dataset, we should always do a forward pass for each token to be predicted, with that token at the last position.

This can be a bit slow, but we can approximate it easily: we will just sample a batch of random slices of length `L`---just as we do for the training---and we will compute `- log p(t)` of the target character t only at the last position (where `p(t)` is the model prediction).

question 12 Implement this, and average over 1000 batches to improve the accuracy. Report the average log-probably in bits (that is, convert to $\log_2$). Show your implementation.

This method essentially ignores the starting `L` tokens of the validation set, where full contexts aren't available. We'll allow this slight inaccuracy to keep things simple.

Next, we'll need to sample from the model. To do this, we'll first need to slice random *seed* of `S` characters from the validation data. We'll set `S` to 16 for all experiments. Then, we simply pass the seed through the model, sample from the probabilities on the last token, add our sample to the seed, and repeat.

For sampling from the output distribution, you can use the following function:

```
import torch.distributions as dist

def sample(lnprobs, temperature=1.0):
    """
    Sample an element from a categorical distribution

    :param lnprobs: Outcome logits
    :param temperature: Sampling temperature. 1.0 follows the given
        distribution, 0.0 returns the maximum probability element.
    :return: The index of the sampled element.
    """
    if temperature == 0.0:
        return lnprobs.argmax()

    p = F.softmax(lnprobs / temperature, dim=0)
```

```
        cd = dist.Categorical(p)

        return cd.sample()
```

The temperature hyperparameter allows you to balance between sampling from the given probabilities (temp 1.0) and giving more weight to the high probability tokens. You can play around with this or just keep the default value.

Question 13: Implement sampling from the model. Run the model for 50 000 batches, sampling and computing validation loss every 10 000 batches. Show that as the validation loss goes down, the samples begin to look less random.

- This can be a challenging training task, even on toy data. Make sure to plot your loss curves and gradient norm curves. Gradient clipping and learning rate warmup are very likely to help.
- You don't need to show full realistic English yet, but should see pretty quickly, for instance, that letters like a, b, c become more likely in the samples than characters like %, * and ~. Then, you should see that the space creates word-sized chunks in your data, and perhaps that certain letters like t and h cluster together.

Question 14: Do a full training run on the toy data. Try to get your samples as close to the data as you can. We strongly recommend an experiment tracking tool like *tensorboard* or *weights and biases* for plotting loss curves in this exercise.

- Once you have a smoothly decaying loss curve over the first 10 or so epochs, you'll have to let the model run for around 50 or so epochs to see real progress.
- Make sure to compute the loss *per token*, so that the batch size and instance length don't affect the smoothness of the loss curve. Pass `reduction='sum'` to the loss function, and divide by the number of tokens manually.
- This task may require a GPU for really good results. If you don't have one, and you don't have the time to set up training on DAS, then report how far you get with your CPU.

For the final report, you only need to get things working on the toy data. However at this point, you may be curious about using a transformer to generate real English language. On a TitanX GPU, this should be possible with about 12 to 24 hours of training time.

Load the Wikipedia data with

```
(train, val), (i2t, t2i) = load_wp()
```

This dataset contains 100mb of text from wikipedia (taken from the first Hutter prize) encoded as a sequence of *bytes*. That is, we have 256 tokens.

These mostly correspond directly to ASCII characters, but special characters are encoded in multiple bytes.

Use the same model as in the previous part, but increase the hidden size, the embedding dimension and possibly the number of layers.

It should be possible to get the model to produce semi-realistic text by training for ~12 hours on a GPU.

Grading key (10pt total)

- question 1: 0.5pt
- question 2: 0.5pt
- question 3: 1pt
- question 4: 1pt
- question 5: 0.5pt
- question 6: 1pt
- question 7: 0.5pt
- question 8: 0.5pt
- question 9: 1pt
- question 10: 0.5pt
- question 11: 1pt
- question 12: 1pt
- question 13: 0.5pt
- question 14: 0.5pt