Homework 11

IANNWTF 21/22

Submission until 06 Feb 23:59 via https://forms.gle/n6ERdhYx3uBPzuGn9

Welcome back to the 11th homework for IANNWTF. Just like last week, we will apply our Deep Learning skills on Natural Language Processing, except in this week we will be making use of the Transformer architecture that literally has transformed the field of natural language processing.

1 Bonus Points for this Homework

Since you're probably all stressed because of the upcoming exam period, this homework differs from the previous ones. This week we provide a detailed step-by-step guide on how to implement everything - you'll just have to follow along with the text.

On top of that it is not mandatory to hand in this homework if you have passed all other homeworks, but instead you will get two bonus points if you do it (which should be very easy). If you've missed a homework before, you can use this homework to make up for one missed homework.

2 Generative text modeling

This week's task will be the same task that we left as an additional bonus task last week: text generation. We ask you to implement a Transformer architecture model (instead of an RNN-model) that predicts a categorical distribution over possible next tokens such that sampling from this distribution leads to plausible next tokens.

The model will take a fixed number of input tokens from a text and predict the distribution over the vocabulary for the next token. With a trained model of this kind, we can iteratively sample the next token, append it to the input, predict the next token distribution and sample again, until we have generated a piece of text.

3 Using an appropriate architecture

Since you are asked to only predict the next token, you do not need a sequence to sequence architecture comprising both encoder and decoder as it would for

instance be used for language translation. Different generative text models such as BERT or GPT use either stacks of the encoder or the decoder. You can either explore these options on your own or (recommended) just follow the steps described in detail in this document. Read them slowly step by step and you will have a very easy time completing this homework.

4 Steps to build a transformer based text generator

4.1 The dataset, preprocessing and tokenization

You will need to import tensorflow and tensorflow_text along with io, datetime and tqdm. Now you want to load a text file (the last section of this document discusses possible texts) that you will use to fit a model to. You could also fit a model to a folder containing multiple text files, but for simplicity we assume all the text comes from one file.

With the text loaded as a string, you now want to prepare the training data. That means, split the text into tokens (sub-words) that are indices in a vocabulary. For this you now train a sentencepiece tokenizer on your text file and load the trained model with tensorflow-text (take a look at the tokenization notebook for how to do that). You will have to decide on the vocabulary size. You can experiment with values between 2000 and 7000 for good results. With the loaded sentencepiece tokenizer model, you can and should now tokenize your text data (which is a single string). Before moving on, let's be clear on what we try to do. We want to have input sequences of m tokens (m should be between 32 and 256) and we want to have a target of one token, which is the next token that comes after the m tokens. So we want to use a sliding window over our tokenized text to split it into all possible windows of $m+1$ tokens. To achieve this, you can use tf_text.sliding_window and pass the tokenized text and the width $m+1$ as arguments. Now that you have a dataset of all possible windows, you can turn the tensor into a tensorflow dataset as usual with from-tensor-slices. Before batching, as always, you want to shuffle the data to decorrelate the training examples (and the kind of tokens that are in them) that are shown sequentially to the model during training. Do not forget to batch the dataset next. The only thing that's left to do then is to split the windows into input data and targets. For that, you can use the map method and a lambda function that maps a sequence of $m+1$ to a tuple of two tensors. The first tensor has the first m tokens and the second tensor has the last token. That's it - the data side of the homework is complete.

4.2 The model components

With the dataset ready, it's time to write the code for the language model itself. We will go for the GPT version of transformer-based models which uses the decoder block from the original transformer model to build a next-token predictor.

The first thing you want to do is write a subclassed layer class that embeds the individual token indices in the input (each index should be mapped to a vector that is looked-up from a table). For this you can use tf.keras.layers.Embedding, in which the input dimension should be the vocabulary size that you chose and the output dimension is the dimensionality of the embeddings (try something between 64 and 256). What this subclassed layer should do is to embed not only the token indices but also their position in the input. This means the model will also learn the positional embedding with a second embedding layer that has an input dimension of the sequence length and the same output embedding dimension. Without positional encoding, the order of the input sequence would not affect the model in any meaningful way. In the call method of the subclassed layer that you are working on, you want to construct a tensor with tf.range from zero up to m (input sequence length). This will act as the indices to look up the positional code for each sub-word. One of the embedding layers will be called on the input sequence and the other on the generated tf.range indices. You will add the two embeddings (like it was done in a ResNet) and that was the first step of building the model.

The next part is the TransformerBlock. You again should create a subclassed layer for this. In it, create a MultiHeadAttention layer with 2-4 attention heads and a key dimension of the dimensionality used for the embeddings in the previous step. If you'd rather understand what is happening inside MultiHeadAttention, refer to the Courseware to see how it can be implemented. Besides the MHA layer, you also need to instantiate two Dense layers. The first has a ReLU and between 32 and 256 units and the second has no activation and again as many units as the dimensionality of the embeddings. You will also want to instantiate two dropout layers with a dropout rate of 0.1. Another important part of the transformer block are two layer-normalization layers, which you get with tf.keras.layers.LayerNormalization. Set their epsilons to 1e-6. Now in the call method, you give the input to the multi head attention layer as both value and query arguments (internally it will then also be used as the keys), meaning the embedded inputs are used as both the query, value and key arguments. Then you use dropout on the output and add the output to the layer inputs, like in residual connections in a ResNet. To the result, you apply the layer normalization. The result - let's call it ln_out - will be used for another residual-connection: Apply the two dense layers to it, followed by dropout, and then add ln_out to the result (the residual-connection), before applying the second layer normalization. Wow. That was a lot! But the good news is, you're almost done building a GPT-like text generator!

4.3 The subclassed model

Now you need to create a subclassed tf.keras.Model. The model will have an init method, a call method, a method to reset metrics, a train step method and

a method to generate text given a user defined prompt. We'll go through the methods step by step.

In the init method, you will set up everything that will be used in the other methods of the model. The model will have the trained tokenizer as part of it (such that it can output text and not just token IDs). Also add the optimizer (try Adam) and the loss function. As a loss function you want to use SparseCategoricalCrossentropy because the targets aren't one-hot encoded but indices. Set the from_logits argument to True because our model will not have a softmax activation function (as this can lead to numeric instability). Add the tf.keras.metrics objects (as presented in the tensorboard notebook). Instantiate your custom token-positional-embedding layer, the transformer block, a 1D global pooling layer and a dense layer (no activation) with as many units as the vocabulary size. In the call method, apply these layers in that order to the input.

Both the reset_metrics and train_step methods can be taken from the tensorboard notebook.

Before you implement the generate_text method, we need to be clear on what it is supposed to do. We want a function that takes a string input (a prompt), tokenizes the string and uses it as the input for the model. Since the prompt will probably not be of the right length, we need to pad the token tensor, as well as add an extra batch dimension. With the logits obtained from the model (think of them as scores for how likely each token in the vocabulary is to be used next), we want to sample the next token. This can be done with tf.random._categorical. It is common practice however to not allow super unlikely words to be sampled, which is why we want our function to take an additional argument "top_k", specifying how many most likely tokens can be sampled from. With that taken care of, we want to concatenate the last model input with the freshly sampled token (which is a token index), truncate the length of the input, and repeat until the maximal input length is reached (still containing the original prompt). Finally we want the function to use the tokenizer to detokenize the result and return it, such that we can also add it to the tensorboard for model inspection during training.

To implement the top_k sampling, you will need to find a way to sample only from the top_k highest logit scores. You can achieve this by using tf.math.top_k with sorted=True, which gives you two tensors, one with the logits and another with the corresponding indices. Sampling can then be done using these logits, using the sampled index as an index for the tensor that contains the real indices.

The iterative sampling can be achieved with a for loop and for the padding you need to add "pad" tokens to the tokenized prompt. The log that shows when you trained the sentencepiece tokenizer contains information on special tokens and their indices.

4.4 The training loop

For the training loop, take a look at the tensorboard notebook (from the GANs week). Try to incorporate adding the generated text from each epoch to the tensorboard log. Also there is no validation set in this task, since the notion of overfitting here isn't as straightforward as in other tasks.

Before running the training loop, you should load the tensorboard and instantiate a log writer and you should also show the tensorboard before starting the training in order to keep track of the training progress and the generated text. Depending on the text used, training can take a while, so it might make sense to regularly save models to your google drive (if you're using colab) and then resume training later by loading the weights. Training should take between 100 and 600 epochs depending on the text used - but you don't have to train until the model converges.

5 The text data

You can use any text data that you find interesting, however the data should be accessible through a link (such that we can reproduce your results) and the corpus should be sufficiently large. You can use the bible text from last week, but you can also use "Beyond Good and Evil" by Nietzsche which can be obtained here: https://s3.amazonaws.com/text-datasets/nietzsche.txt

Other examples (feel free to choose whatever you find interesting) could be text files containing a collection of recipes, books, news articles, essays etc. The only constraint is that it must be downloadable with a URL. You can (and should) then download the file to a path with e.g.

```
path = tf.keras.utils.get_file("nietzsche.txt",
origin="https://s3.amazonaws.com/text-datasets/nietzsche.txt")
```