Homework 11

IANNwTF

November 6th, 2023

This homework's deadline is *17.02., 23:59*.
Submit your homework via https://forms.gle/ApAZ5ubY8ewgNmJA9
Remember to review the work done by two of your fellow groups!

Contents

1 Recap on Reviews

Welcome back to your next assignment in IANNwTF. Here's your weekly refresher on how to do reviews:

In addition to doing your homework, you will have to review two groups' homework submissions from last week. Record which two groups you wrote reviews for on the homework submission form. This requires you to find two other groups with whom you can have a short (10 min) meeting in which you go over and discuss each other's homework submissions. Write a summary of your criticisms and post it on each other's forum pages. We recommend using the Q&A timeslots to find other groups to work with, but you are free to meet however and whenever you like. **If there still remain questions regarding your or another group's code at the end of your discussions, please feel invited to bring them up with us in the QnA sessions!** Just to be clear: We will not penalize any submission (by you or any of the groups you reviewed) based on such questions. The point is only to help you to better understand concepts and to improve your code.

Here's an example of what your meetings could look like: One group starts out by presenting their code to the other two groups. As they are being walked through the code, the other two groups give feedback, e.g. "I like how you designed your data pipeline, it looks very efficient.", "For this function you implemented there actually exists a method in (e.g.) NumPy/TensorFlow which you could've used instead." or "Here you could've used a different network architecture/set a higher learning rate to achieve better results.". They post their main points of critique on the forum and then another group starts to present their code...

Important! Not every member of every group has to be present for this. You could for example implement a rotation where every group member of yours only has to participate in a review every third week.

2 Assignment: Transformers

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

Since you're probably all stressed because of the upcoming exam period, this homework differs from the previous ones. We will now tackle the outstanding task from last week, but this time with a different network architecture. Additionally, we provide a detailed step-by-step guide on how to implement everything - you'll just have to follow along with the text.

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.

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. Here we decide to use the decoder in order to use its autoregressive property to train it on prediction errors of all tokens in the input sequence. You can either explore decoder-block based generative language models on your own or (recommended) just follow the steps described in detail in this document. Read them slowly step by step and you should have an easy time completing this homework.

2.1 The dataset, preprocessing & tokenization

You will need to import tensorflow and tensorflow_text along with io, datetime and tqdm. Now you want to load the text file that you will try to fit your model to.

With the text loaded as a string, you now want to prepare the training data. First, you might want to clean up the data like last week and save it again as a text file.

Next, split the text into tokens (sub-words) that are indices in a vocabulary. For this, we suggest you train a SentencePiece tokenizer on your text file and load the trained model with tensorflow-text (take a look at the last section of this notebook for how to utilize the SentencePiece tokenizer). You will have to decide on the vocabulary size. You can experiment with values between 2000 and 7000. With the loaded SentencePiece tokenizer model, you can and should now tokenize your text data. As an alternative to training your own tokenizer, you can also use a pre-trained tik-token tokenizer from OpenAI (the gpt-2 variant, as shown in the "how to build a language model" section) but keep in mind that this will have a much larger vocabulary size.

Before moving on, let's be clear on what we try to do. We want to have input sequences of length m tokens (m should be between 32 and 256). We want to use the same sequence but shifted one to the right as the targets. 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. Alternatively, use the window method in the data pipeline, as you've done in a previous homework. Then, out of the sequence with length m+1 you declare the first m tokens as the inputs and the last m tokens as the targets.

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` (unless you've done so before in order to use the window method.

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. Then batch the dataset.

To recap: In your data pipeline you should

- Tokenize the text with your SentencePiece tokenizer (or with a pre-trained tik-token tokenizer)
- Create the input sequences of length m that also serves as the targets
- Create a tensorflow dataset from the text and window it (unless you've done so already)
- shuffle, batch, prefetch

That's it - the data side of the homework is complete.

2.2 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. Causal masking in the multi-head-attention layer allows us to use the entire input sequence as the targets, since embeddings at time point t are only allowed to attent to previous tokens, not subsequent tokens.

2.2.1 The Embedding

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 embed not only the token indices but also their position in the input. Transformer based models do not inherently operate on sequences but rather on sets of tokens. This means the model will also need to learn a 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 way.

- In the **init method** define the two embeddings for the token indices and their position using `tf.keras.layers.Embedding`. Assign their input and output dim as described above.
- In the **call method** first construct a tensor with `tf.range` from *zero* to m (the input sequence length, which can be obtained from the input's shape). This will act as the indices to look up the positional code for each

sub-word. Then, feed the token index embedding layer with the input sequence and the positional embedding layer with the newly constructed range tensor. FInally add the two embeddings (like it was done in a ResNet) and return the result.

2.2.2 The TransformerBlock layer

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 like to understand what is happening inside the MultiHeadAttention, refer to the Courseware to see how it can be implemented.

- In the **init method** you have to instantiate the MHA layer, with the `num_heads` and `key_dim` arguments assigned as described above. Additionally, you also need to instantiate two Dense layers. The first has a ReLU activation 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. Remember that dropout layers require a training argument in the call method. THis needs to be passed down all the way through higher order layers, down to the call method of this layer. Finally, add two layer-normalization layers, with an epsilon of 1e-6.

- In the **call method** 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. Importantly the `use_causal_mask` argument has to be set to true during the call, so the model does not attend to future tokens. Then you use dropout on the output of the MHA and add the result of that back to the layer input, like in residual connections in a ResNet. To the sum, you apply layer normalization. Sometimes, adding the layer norm is done before the sum for stability reasons. The result - let's call it ln_out - will be used for another residual connection: Apply the two dense layers to it, followed by another dropout layer, and then add ln_out back to the result of those dense and dropout layers (the residual connection). Remember the training argument. Finally, apply the second layer normalization and return the final output.

Wow. That was a lot! But the good news is, you're almost done building a GPT-like text generator!

2.2.3 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 SentencePiece tokenizer as part of it (such that it can output text and not just token IDs). Also, add the optimizer (try Adam with a learning rate of 0.001) 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. Add the tf.keras.metrics objects (as presented in this notebook from the Courseware). Instantiate your custom token-positional-embedding layer, the transformer block(s) and a dense layer (no activation, with as many units as the vocabulary size).
- In the **call method** apply the initialized layers to the input in the order in which they were instantiated.
- The **reset_metrics** and `train_step` methods can both be taken from the same notebook, with minor changes (we only have inputs in our data set and use them as the targets).
- The **generate_text(self, prompt, output_length, top_k)** method should receive a text prompt and the desired output length and return a continuation of the prompt of a specified length.

Before you implement the generate text method, we need to be clear on what exactly 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, to finally obtain the probabilities for the next token that we can use to sample it. We want to sample not only one token but a pre-defined number of tokens (you choose how much text you want to generate). To do so we generate (sample) one token, add it to the prompt and then repeat.

You need to tokenize your prompt and add an extra batch dimension. GPT does not require padding.

Then you can obtain the logits from the model by calling it on the padded prompt (think of logits as unnormalized scores for how likely each token in the vocabulary is to be used next), and now you want to sample the next token. This can be done with `tf.random._categorical` on the last time-step in the sequence of logits that your model outputs (since you want to predict the next word, not a word that is part of the inputs). It is common practice however to not allow super unlikely words to be sampled, which is why you want your method to take an additional argument "top_k", specifying how many of the most likely (sub-)words will be sampled from.

To implement this 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 returns two tensors, one with the top_k highest logits and another with the corresponding token indices. Sampling can then be done using these top_k logits and then using the sampled index to index the tensor that contains the corresponding token indices.

After having sampled a token, you want to concatenate the last model input with the freshly sampled token, if necessary truncate the length of the input (e.g. by indexing with [-self.max_len:]), and repeat until the desired length is reached.

Finally, you want the method to use the tokenizer to detokenize the result and then return it, such that we can add it to the Tensorboard for model performance inspection during training.

So to recap: your generate_text method should tokenize the prompt it is given and then create predictions that are concatenated to the prompt until it is of the desired length. This sequence is then detokenized and returned as the output of the method. Until the sequence is of the desired length the following process should be repeated:

- tokenize your text input.
- Use `tf.expand_dims` to add a batch dimension to the sequence
- Pad the input with the padding by using `tf.pad`
- Feed the input to the model
- From the outputs last logits (should be a single vocab-size dimensional vector) obtain the top_k highest logit scores and corresponding indices
- Sample one token from the top_k distribution
- Index the tensor with the corresponding token index
- Concatenate the token to the sequence
- repeat if the desired output length is not yet achieved

2.3 Training

For the training loop, take a look at the Tensorboard notebook again. Try to incorporate adding the generated text from each epoch to the Tensorboard log, so you can see the prediction of your model during training. You will have to feed your generation method a short starting prompt that it can generate off of, e.g. "What is". For a validation set, you can use a different part of the text that you did not use for the training data. Keep in mind that transformers require ridiculous amounts of training data and your model will overfit the training text specifics.

Before running the training loop, you should load the Tensorboard, instantiate a log writer and you should also look at the empty Tensorboard log *before* starting the training in order to keep track of the live training progress and the generated text (unless you start the Tensorboard from a terminal). Depending on the text used, training can take a long time, 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. To achieve the best results training could take between 100 and 600 epochs depending on the text used - but you don't have to train until the model converges as it can take quite a very long time.