CS182/282A Deep Neural Networks **April 3^{th}, Spring 2023**

Lecture 18: Transformers,Fine-tuning, Embeddings

Lecturer: Anant Sahai *Scribes: Kyle Pedersen, Yewei Li*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

18.1 Transformers: continued

18.1.1 Recap and overview: From RNN-style seq2seq models to transformers

We started with the RNN-based encoder-decoder style approach as shown in Figure18.1. The encoder was built internally using stacks of RNNs which were weight shared across time and passed their states to the decoder network. The decoder network was built similarly except that it was always used at test time in an autoregressive fashion. Specifically, the previous output was then fed in as the next input.

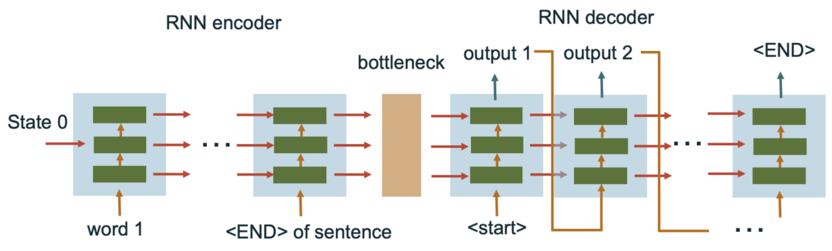


Figure 18.1: RNN-based encoder-decoder sturcture[2]

Problem of this architecture: The state has to carry information about everything and it might not be able to focus on the most important thing as they carry the entire input sentence (through time).

Solution: Use attention to create the counterpart of a U-net in the sequence-to-sequence task. We will later explore how to make full use of the attention mechanism to develop transformers.

Several steps will be taken to move from RNN-style seq2seq models to transformers. The steps include

1. Use attention(cross-attention) to give decoder access to relevant local context in input.
2. Use attention(self-attention) to give decoder causal access to output so far.
3. Remove RNNs from decoder.
4. Introduce attention(self-attention) in the encoder. Allow RNNs to be removed in the encoder but keep it causal like step 2 above.
5. Remove causality from encoder and use non-causal self-attention.

These steps show that entire thought-process of introducing the attention mechanism while removing the unnecessary RNN structure.We will elaborate on these steps in the following sections.

18.1.2 Cross-attention and decoder self-attention

This corresponds to step 1 through 3.

Step 1 introduces cross-attention to solve the problem of bottleneck.

Furthermore, the decoder might also want to have relevant local context about the output it has already generated. Just as the case in passing information from encoder to decoder where the state has to carry all the relevant information about the input, in the decoder itself, the state has to carry everything about what has already been done in the past. Therefore, we use attention (self-attention) to give decoder causal access to output so far.

A natural intellectual question that comes up is what now the role of the state in the decoder is. By step 1 and 2, we have a way of accessing what we've already done back here so why do we need this state and what is the point of retaining the RNN states? The idea of having zero states in spirit and the idea of transformers were (first) explored in the paper 'attention is all you need' in 2017[1]. Therefore, we take the third step, which is to remove RNNs from decoder. This step is natural and reasonable since we already have self-attention in the decoder. However, we still have RNNs in the encoder. In order to remove RNNs in the encoder, attention is introduced at the encoder.

18.1.3 Multi-head attention in transformers

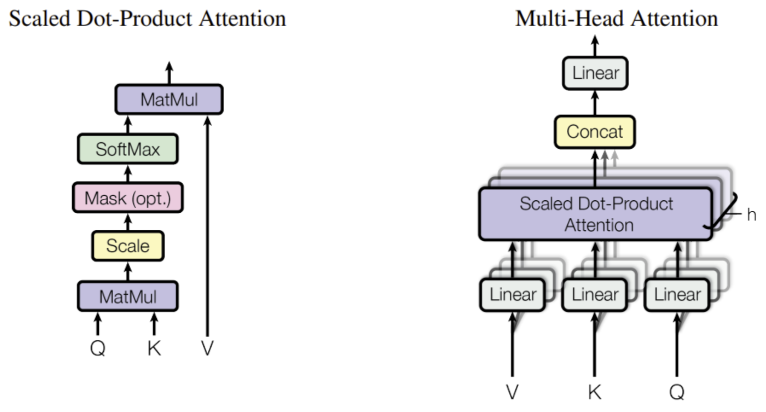


Figure 18.2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel. [1]

The steps of attention are as follows.

1. First, we receive the input from the previous layer.
2. Use the input to create key vector $k_{t,l}$, query vector $q_{t,l}$ and value vector $v_{t,l}$ by applying linear layers with different weights respectively.Here, $l = 1, 2, \ldots, m$

Comment: Linear layers are what the original paper suggested. But for queries keys and values (blocks), they can also have MLPs inside.

3. Store the key vector and value vector into the table. The table contains all key-value pairs generated by the input sequence at this layer and previous layers.
4. For each attention head, pass the query vector as an input to the attention block (which have access to the table) and generate the output of the softmax.

Mathematically, for each attention head, the attention score $A = softmax(\frac{QK^T}{\sqrt{d}})V$, where Q, K, and V are the corresponding query, key and value matrix for that attention head and d is the dimension of each query/key.

A few more comments:

- The attention block has a nonlinearity inside - the softmax that outputs the attention scores.
- Step 2,3,4 are called multi-head attention where we have m parallel attention mechanism with m pairs of queries, keys and values.
- We can have multi-head attention for both self-attention and cross-attention.

18.1.4 Encoder block of transformers

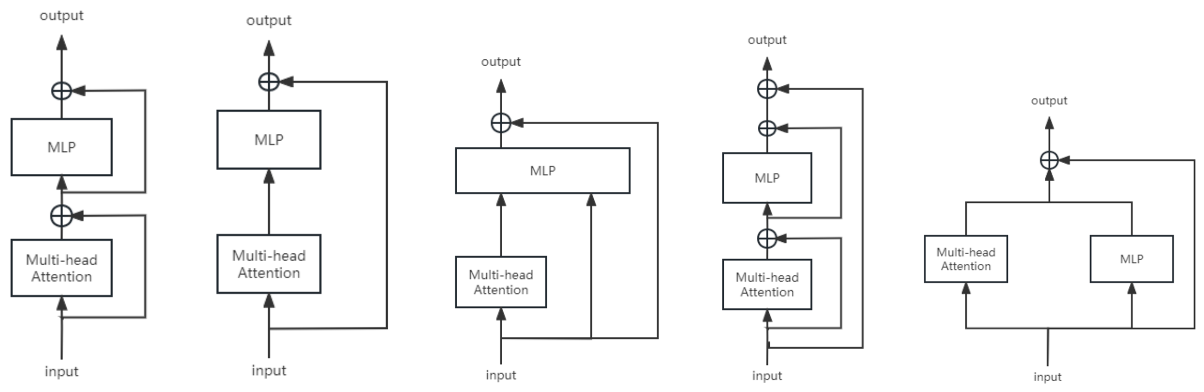
After introducing(a brief recap) of the (multi-head) attention mechanism, we will explore the rest two steps and see how we can build the encoder of transformers in this section.

First, we introduce attention(self-attention) in the encoder, which allows RNNs to be removed in the encoder but keep it causal like step 2 above. This is a conceptual step. Then we will remove the causality in encoder.

When building the transformer, we keep the idea of layers in the encoder in RNN. However, we remove the states from RNNs and the structure inside the transformer block is different.

For each block, there are three ingredients: (1) Attention, (2) MLP, (3) Residual block.(We also have normalization block, it has been discussed in the previous lecture.)

Compared with CNNs: We have MLP component and residual connection in ResNet. Attention plays the role of accessing context and local structure which is what the convolutions were doing in CNNs. We will explore how to combine all of these three ingredients to form a transformer block.



(a) option 1 (b) option 2 (c) option 3 (d) option 4 (e) option 5

Figure 18.3: Possible options of encoder architecture (taken from the lecture)

Some possible combinations are listed above in Figure18.3.

Question: Comment on the disadvantage of option 2 (compared to option 1)

- Attention is trying to route information from earlier part in the sequence that might be relevant to you. With the extra residual connection for the attention block only, MLP has the input of the entire block as its input. Therefore, the multi-head attention can learn something else. Without it, the key-value pairs for the attention block are forced to do double duty, i.e. carrying the information sequence(identity part) as well as processing it to get bottleneck down to the smaller size of $\frac{1}{m}$, where m is the number of attention heads.
- Another reasonable comment is that if the true mapping of attention is (near) zero for option 1, you have to learn an identity mapping or a mapping close to identity for option 2, which is perhaps more tricky.

Side notes: Recall that we hope to concatenate the output of different attention heads instead of summing them up. Furthermore, to enable residual connection, we align the shape of the input and output. Therefore, the shape of the output for each attention heads is $\frac{1}{m}$.

Comment on Option 1 through 5:

- Option 1 is the architecture that was introduced in the original transformer paper.
- Option 2 is not so good as we have discussed before.
- Option 3 is the most intuitive one. This is the first structure we can think of to combine multi-head attention and MLP. The structure also works since it manages to pass the identity mapping to the MLP, which relieves the multi-head attention block of the duty ti learn the identity mapping, as what we try to achieve in option 1.
- We don't use option 4 since the extra residual connection it introduces is not necessarily doing that much except multiplying things by two. The gradients can already pass through the two residual connections introduced in option 1.

- Option 5: Option 1 can be thought of as a serial interconnection of attention and MLP. It is natural to think of whether we can build parallel interconnection. An example of it is option 5. Given that the transformer blocks are stacked one after another, it is not crazy to build this architecture since the output of the attention blocks get processed by the MLP in the next block. Everything get processed just in a more mixed way and there is still enough non-linearity. This was proposed in a recent paper to deal with much bigger Transformer models since it is easier to implement. The below figure is the architecture of an encoder layer in recent paper, which is an example of using option 5.

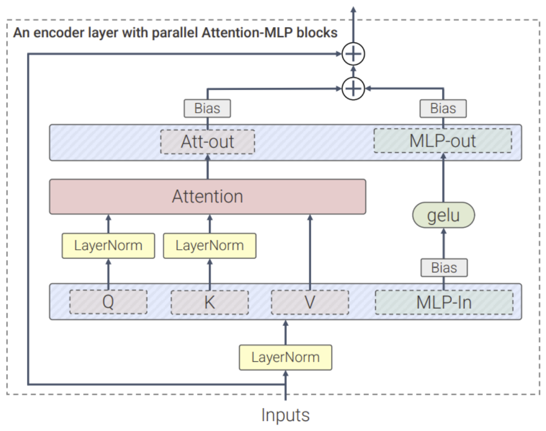


Figure 18.4: Parallel ViT-22B layer, an example of parallel interconnection of attention and MLP blocks [3]

Question: Do we have to be causal for the encoder block?
We keep it causal just to maintain the spirit of what RNN is doing. The decoder has to be causal because the next input is the previous output and we cannot run it till generation without knowing what to feed in next at test time.
Therefore, in step 5, we remove causality from encoder and use non-causal self-attention. Instead of thinking of these key-value pairs as only coming from the past, allow self-attention to see them even from later on in the sequence. There's nothing blocking us from doing that in the code.

The encoder block consists of a stack of N identical layers as shown in figure**??**.(N=6 in the original transformer paper.[1])

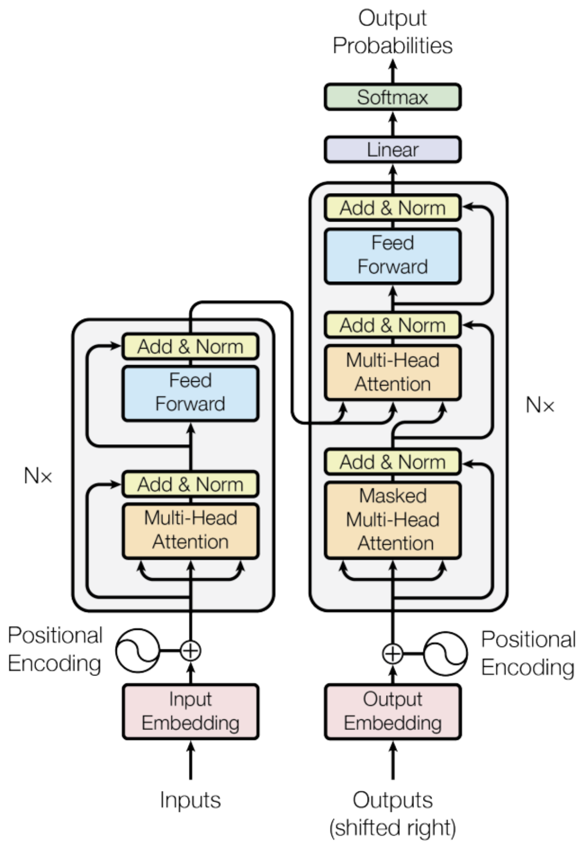


Figure 18.5: The model architecture of Transformer. [1]

Conclusion: We have introduced how to remove any kind of RNN and explicit state. All state is simply held in memory.

18.1.5 Training transformers

Training transformers is very much like training any other sequence-to-sequence models. For the auto-regressive connections in a supervised setting with a target trying to hit, pre-populate the input with what you want to get out during training.

Denote the first input of the decoder as Y_0, the label for the output of the decoder Y_i's, the actual output of the decoder $\hat{Y}_i$'s, where $i = 1, 2, \ldots$

The key question here is: How to force causal access of decoder self-attention during training?

- A naive way: Get one output at a time and train it sequentially.

  Specifically, input Y_0, get $\hat{Y}_1$ and compute the loss and gradients. Then,without zeroing the gradients(doing 'zero-grad' in PyTorch) , stick $\hat{Y}_1$ in and get $\hat{Y}_2$. And so on so forth. However, this method does not fully utilize the processor and it is less efficient.

- A better way: Do causal masking during implementation.

We want to exploit the parallelism across these different parts of the sequence, namely load all of the Y's in at once and have things execute in a way that can take advantage of the parallel processing. The challenge is that the self-attention block should look at different things for different positions in the sequence so that it cannot look into the future even though the information has already been calculated. One way to implement this is to make the logits of 'future' $-\infty$, or equivalently, make the future attention weights zero.

18.2 Transformers In Practice

Transformers are pretrained on many tasks plus self-supervision, and for future problems, it is adjusted via fine-tuning. Fine-tuning allows for better performance on a particular tasks carried on from general pretraining.

18.2.1 Transformer Fine-Tuning

Given a neural network that is pretrained on general data which takes the input X from some domain, the output will typically have a linear layer as its last layer. The linear layer will provide either the maximum label in a classification task or a linear combination of outputs for regression tasks. For example, this can be noticed in ResNet when a global average pool is performed after the initial convolutional and pooling layers.

We can interpret the output before linear layer as a learned feature, which we call an embedding of the input–seeing the output as the most important features of the network.

This process can be summarized as:

1. Remove old head
2. Add new head with random initialization
3. Train new head on task-specific data while freezing the rest of the network

Instead of developing entirely new neural networks for each new task, which can be difficult due to data scarcity, we can utilize pre-existing networks as feature generators. In doing so, we can the new head to be trained for the a new task while discarding the original head, as it was primarily there to help the assist in the network training.

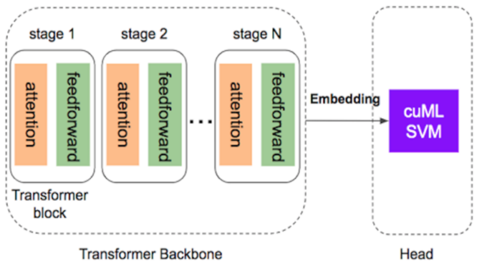


Figure 18.6: With an initial model trained on large amounts of data , we can use its last layer while freezing the other parameters to learn other downstream tasks. In this example, a new head is attached to the pretrained transformer to use SVM for another task via fine-tuning.[5]

One might ask, if an old pretrained Transformer model is, say, trained on images of cats and lemurs, how would this be helpful for diagnosing images for cancer? The reasonable deep learning response to such a question would be this: more data helps a model learn patterns we potentially might not see at the surface, and, since random lemur data may show benefits to extending it skills to identifying cancerous MRI images, it is worth trying.

18.2.2 Learning Embeddings for Language Models

18.2.2.1 Tokenization

Normally the output of a neural network, Y, is an member of $\mathbb{R}^n$. However, when working with language text as an input, this becomes challenging as each piece of text is a collection of characters. To transform the text into vectors, we perform a process called tokenization.

In doing so, we turn a string of characters into string of vectors. Naively, we could map every character to a vector, but, to allow for a variable length of characters, we make a mapping to a fixed length vector of real numbers. After parsing the input string into tokens, we then use a lookup table that turns each token into a vector. This is done by converting the input string into tokens that represent meaningful objects like words, parts of words, or even the original characters. At the output layer, the output is interpreted as a token from its corresponding numerical vector.

In the lookup table, the real-number mapping of these tokens are values that are learned through the training process. However, the tokens themselves are not tuned, as they are parsed separately. Unlike the rest of the network running on the GPU for a transformer, the tokenization process is run on the CPU.

18.2.2.2 GPT Style

One method for learning embeddings is in the style of GPTs (Generative Pre-trained Transformers). Their goal is to self-supervise with the goal of predicting the next token of a sequence. For each token going from the beginning of the network to the linear layer, the network predicts what the next token will be. This can be thought of as similar to state estimation for a dynamical system. Similar to learning parameters of an unknown dynamics model, the tokens are being predicted via classification.

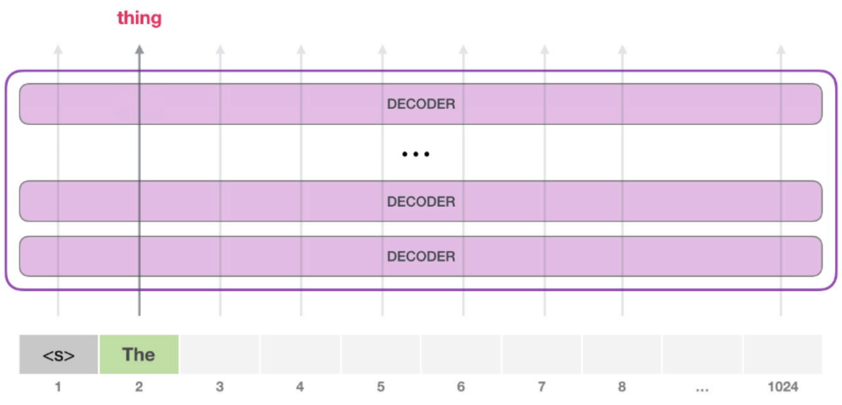


Figure 18.7: In GPT style networks, the token 'The' predicts the token 'thing.' The token 'thing' is then fed through the network again to predict the next words, and so on. [6]

GPT style networks are trained to be good at autocomplete for sequences of tokens, but extending them to other tasks is an important part of applying their functionality. The importance of the training enables the inner architecture of the network to have learned how to express itself in real-world language output. This, as a result, has the output be a learned set of features that other machine learning algorithms (i.e. SVM, another neural network, or a random forest) can be stacked on top of to learn further tasks. In this case, the additional downstream layers added are then trained using an optimizer to further craft a model for a more particular task using the original pre-trained model.

18.2.2.3 BERT Style

Whereas GPT sought to predict the next token in a word sequence, BERT style networks mask random inputs to the encoder (similar to the masked autoencoder). Then, the encoder is told to reconstruct the input, which is the model's attempt at filling in the missing tokens. By masking out random tokens (typically 15%), the model learns to fill in words contextually and not simply returning the same input phrase (the identity function).

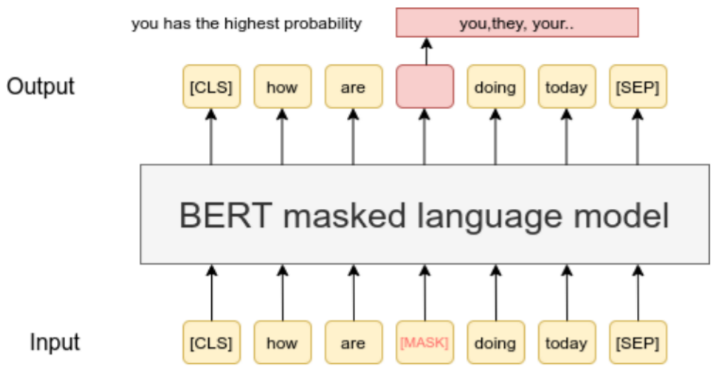


Figure 18.8: In this BERT network's encoder, the input token 'you' is masked and fed through while being reconstructed. The word with the highest probabilty at the output is 'you,' which was what the training data label was. [7]

18.3 What we wish this lecture also had to make things clearer?

For the transformer part, we wish that the lecture could elaborate more on the variant of transformers like vision transformers.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. 2017.

[2] Chen, X., & Choi, Y. (2022). Fall 2022 EECS 182 Lecture 14, Attention/self-supervision [Scribe notes]. University of California, Berkeley. https://www.eecs182.org

[3] Dehghani, Mostafa, et al. "Scaling vision transformers to 22 billion parameters." arXiv preprint arXiv:2302.05442 (2023).

[4] Jianzhi W., Jason Y.(2022). Fall 2022 EECS 182 Lecture 18, Transformers [Scribe Notes]. University of California, Berkeley. https://www.eecs182.org

[5] Jiwei Liu & Chris Deotte (2022). Fast Fine-Tuning of AI Transformers Using RAPIDS Machine Learning. https://developer.nvidia.com/blog/fast-fine-tuning-of-ai-transformers-using-rapids-machine-learning/

[6] Jay Alammar (2019). The Illustrated GPT-2 (Visualizing Transformer Language Models). https://jalammar.github.io/illustrated-gpt2/

[7] Nils Reimers (2019). MLM. https://www.sbert.net/examples/unsupervised_learning/MLM/README.html/