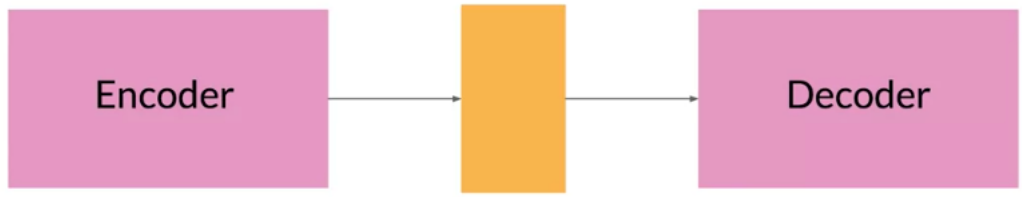
Natural Language Processing with Attention Models

Week 1 - Neural Machine Translation

Seq2Seq model

- Introduced by Google in 2014
- Maps variable-length sequences to fixed-length memory
- LSTMs and GRUs are typically used to overcome the vanishing gradient problem

Seq2Seq model



How are you today?

Wie geht es Ihnen heute?

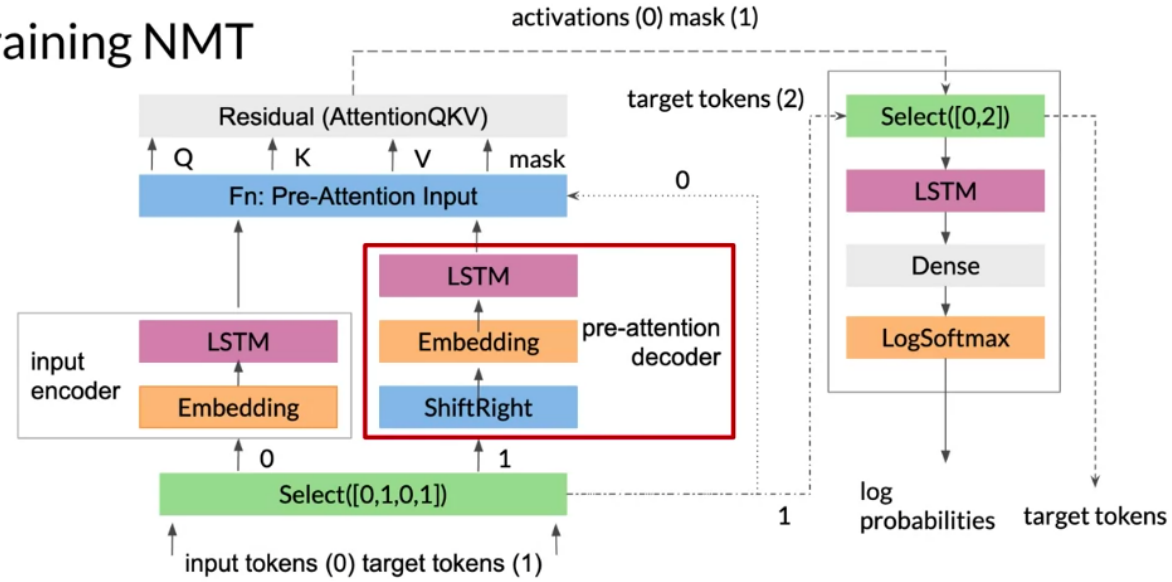
Seq2Seq shortcomings

- Variable-length sentences + fixed-length memory =

- As sequence size increases, model performance decreases

1.25

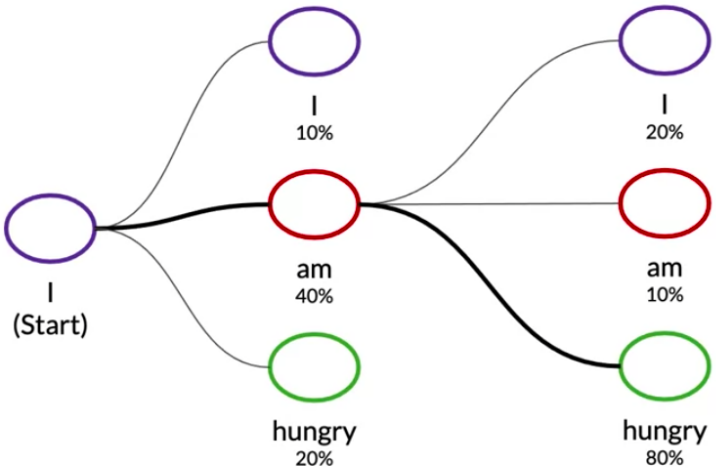
Training NMT



Summary

- BLEU score compares “candidate” against “references” using an n-gram average
- BLEU doesn’t consider meaning or structure
- ROUGE measures machine-generated text against an “ideal” reference

Beam search example



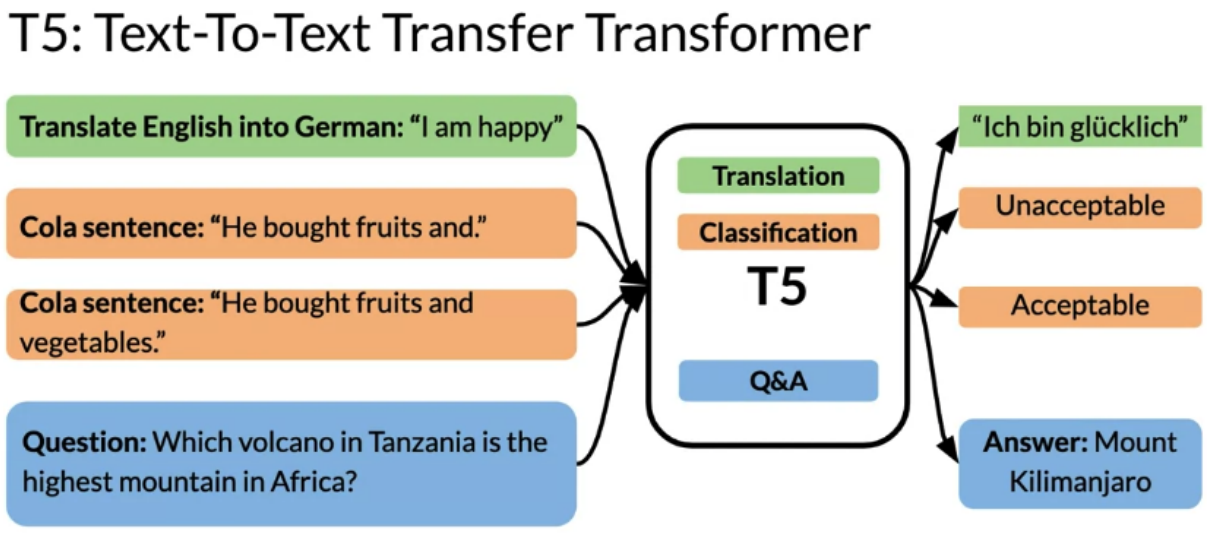
B = 3

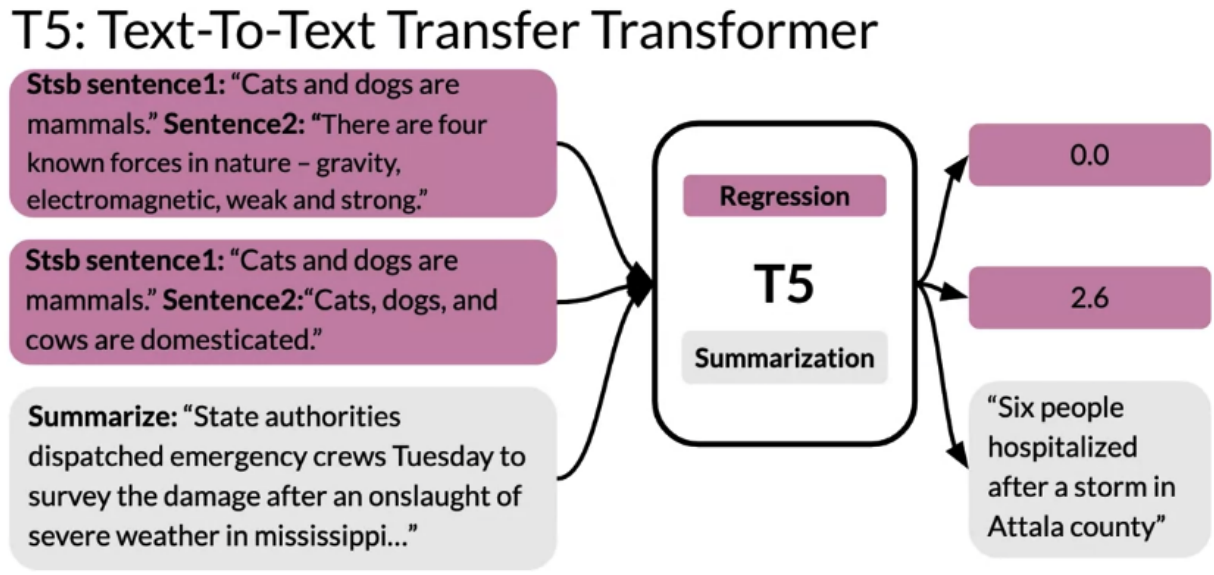
Example: MBR Sampling

To generate the scores for 4 samples:

1. Calculate similarity score between sample 1 and sample 2
2. Calculate similarity score between sample 1 and sample 3
3. Calculate similarity score between sample 1 and sample 4
4. Average the score of the first 3 steps (Usually a weighted average)
5. Repeat until all samples have overall scores

Week 2 - Text Summarization





Summary

- Transformers are suitable for a wide range of NLP applications
- GPT-2, BERT and T5 are the cutting-edge Transformers
- T5 is a powerful multi-task transformer

Summary

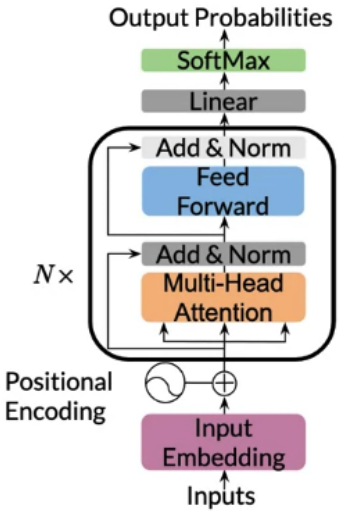
- Dot-product Attention is essential for Transformer
- The input to Attention are queries, keys, and values
- A softmax function makes attention more focused on best keys
- GPUs and TPUs is advisable for matrix multiplications

Summary

- Different heads can learn different relationship between words
- Scaled dot-product is adequate for Multi-Head Attention
- Multi-Headed models attend to information from different representations at different positions

1.50

Transformer decoder



Overview

- input: sentence or paragraph
  - we predict the next word
- sentence gets embedded, add positional encoding
  - (vectors representing $\{0, 1, 2, \ldots, K\}$)
- multi-head attention looks at previous words
- feed-forward layer with ReLU
  - that's where most parameters are!
- residual connection with layer normalization
- repeat N times
- dense layer and softmax for output

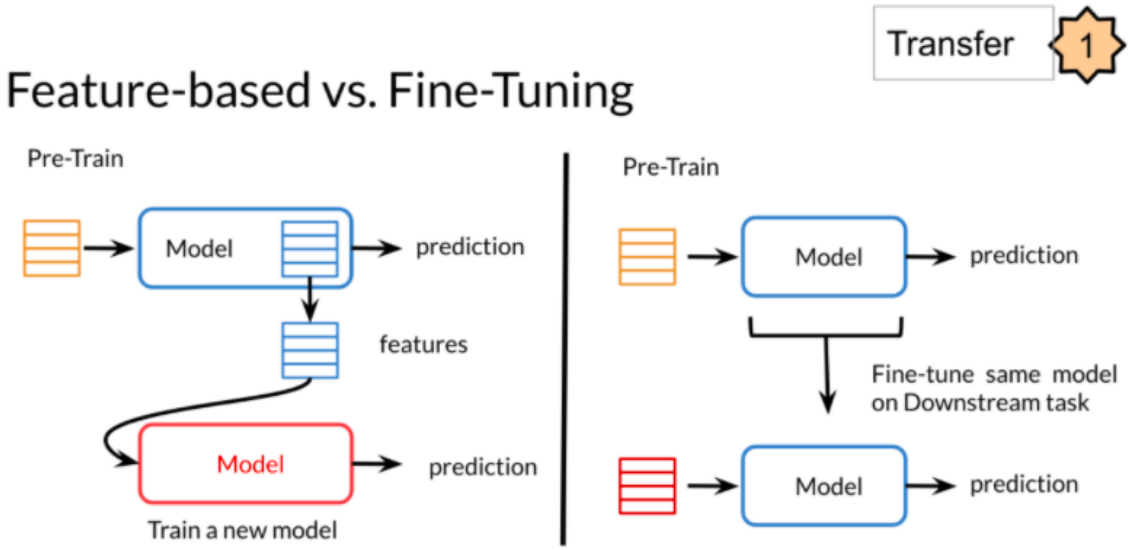
Summary

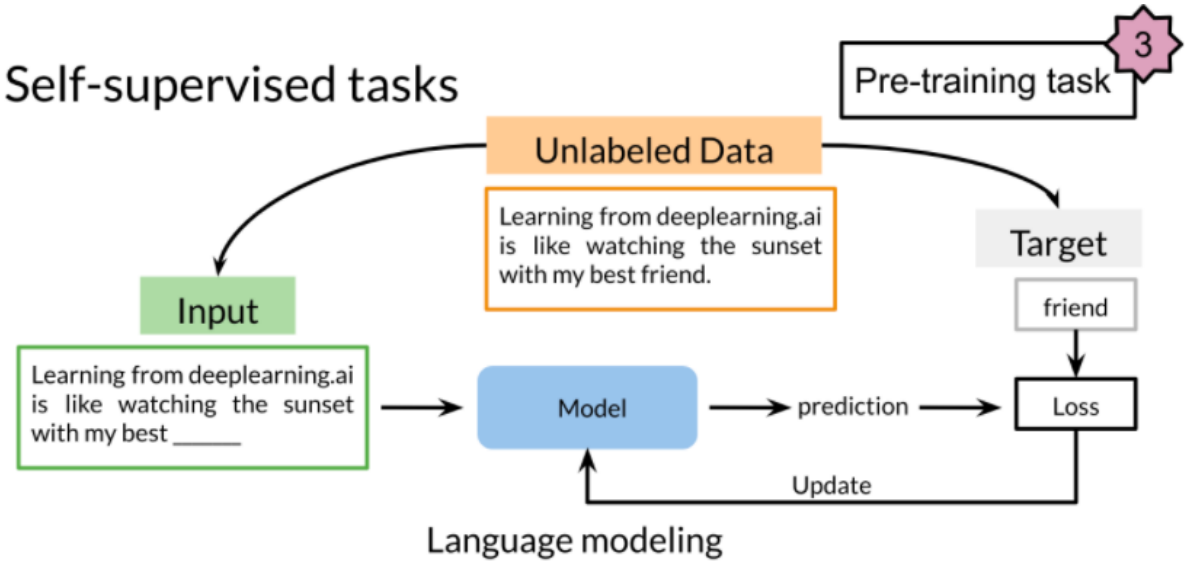
- Transformer decoder mainly consists of three layers
- Decoder and feed-forward blocks are the core of this model code
- It also includes a module to calculate the cross-entropy loss

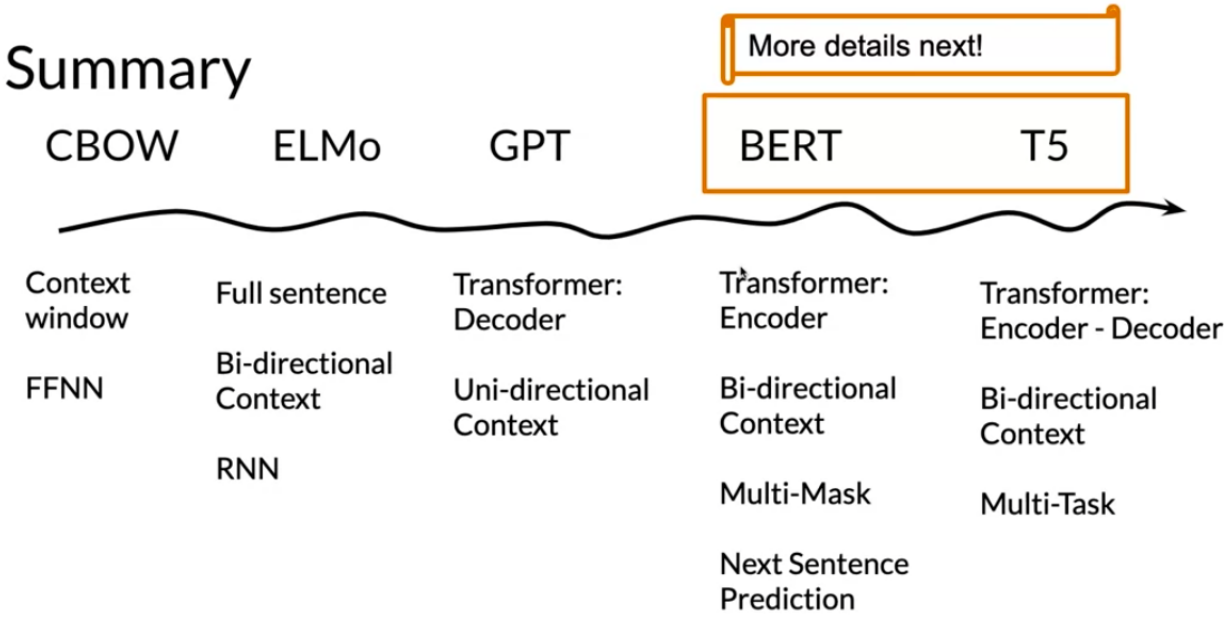
Week 3 - Text Summarization

There are three main advantages to transfer learning:

- Reduce training time
- Improve predictions
- Allows you to use smaller datasets



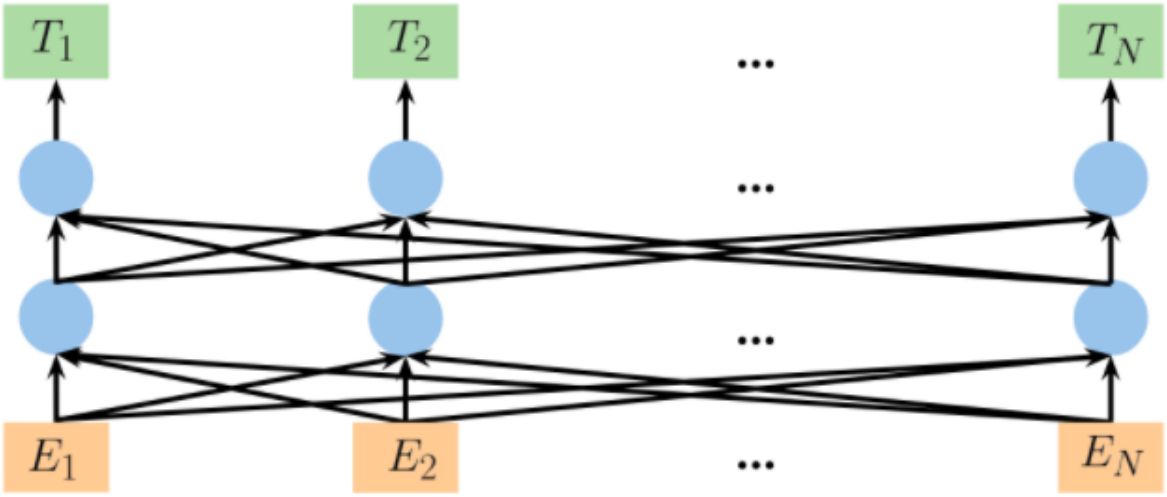




BERT

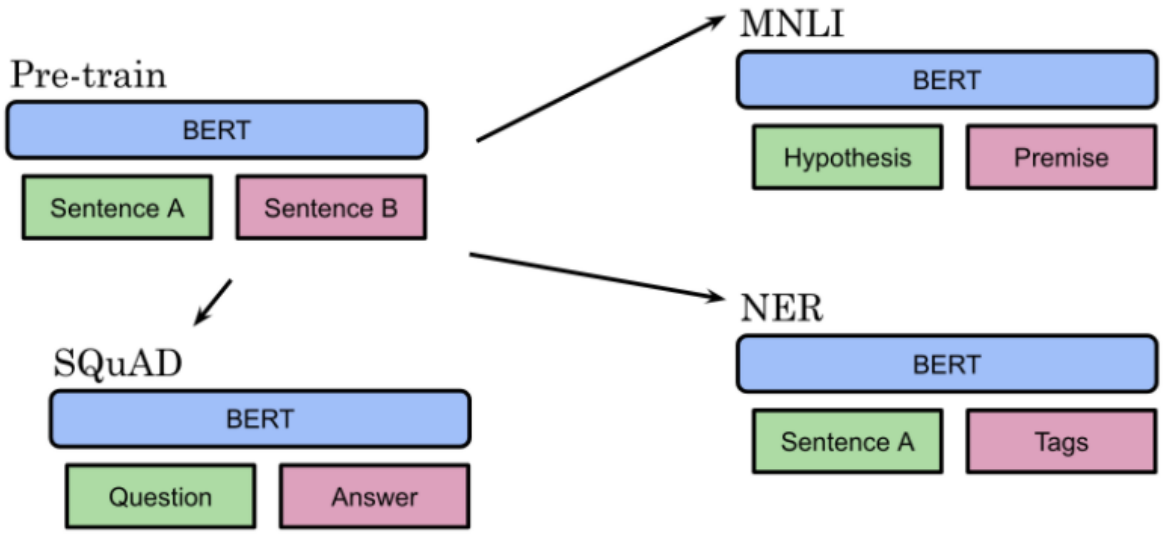
- Choose 15% of the tokens at random: mask them 80% of the time, replace them with a random token 10% of the time, or keep as is 10% of the time.
- There could be multiple masked spans in a sentence
- Next sentence prediction is also used when pre-training.

- Makes use of transfer learning/pre-training:

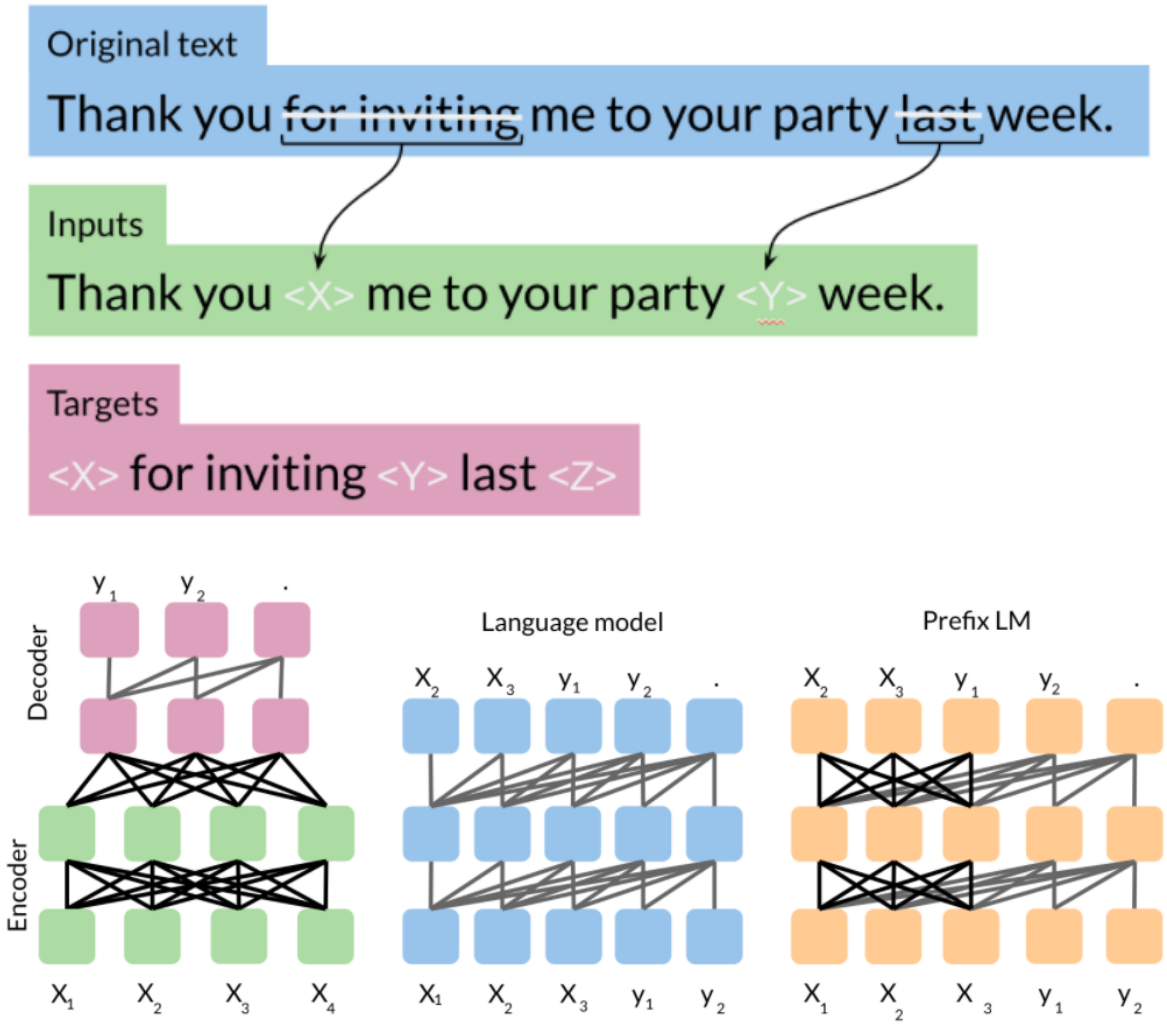


Fine tuning BERT

Once you have a pre-trained model, you can fine tune it on different tasks.



Transformer T5



Week 4 - Chatbot

Transformer Complexity

One of the biggest issues with the transformers is that it takes time and a lot of memory when training. Concretely here are the numbers. If you have a sequence of length L, then:

L=100	L^2 = 10K	(0.001s at 10M ops/s)
L=1000	L^2 = 1M	(0.1s at 10M ops/s)
L=10000	L^2 = 100M	(10s at 10M ops/s)
L=100000	L^2 = 10B	(1000s at 10M ops/s)

So if you have N layers, that means your model will take N times more time to complete. As L gets larger, the time quickly increases.

- Attention: $\text{softmax}(QK^T)V$
- Q, K, V are all [L, d_model]
- QK^T is [L, L]
- Save compute by using area of interest for large L

When you are handling long sequences, you usually don't need to consider all L positions. You can just focus on an area of interest instead. For example, when translating a long text from one language to another, you don't need to consider every word at once. You can instead focus on a single word being translated, and those immediately around it, by using attention.

To overcome the memory requirements you can recompute the activations. As long as you do it efficiently, you will be able to save a good amount of time and memory. You will learn this week how to do it. Instead of storing N layers, you will be able to recompute them when doing the back-propagation. That combined with local attention, will give you a much faster model that works at the same level as the transformer you learned about last week.

LSH Attention

In Course 1, I explained how locality sensitive hashing (LSH) works. You learned about:

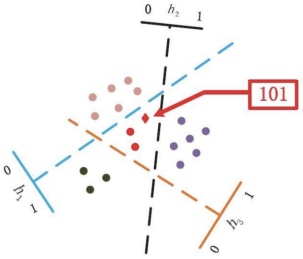
- KNN
- Hash Tables and Hash Functions
- Locality Sensitive Hashing

- Multiple Planes

Here are the steps you follow to compute LSH given some vectors. The vectors could correspond to the transformed word embedding that your transformer outputs. Attention is used to try which query (q) and key (k) are the most similar.

Compute the nearest neighbor to q among vectors $\{k_1, ..., k_n\}$

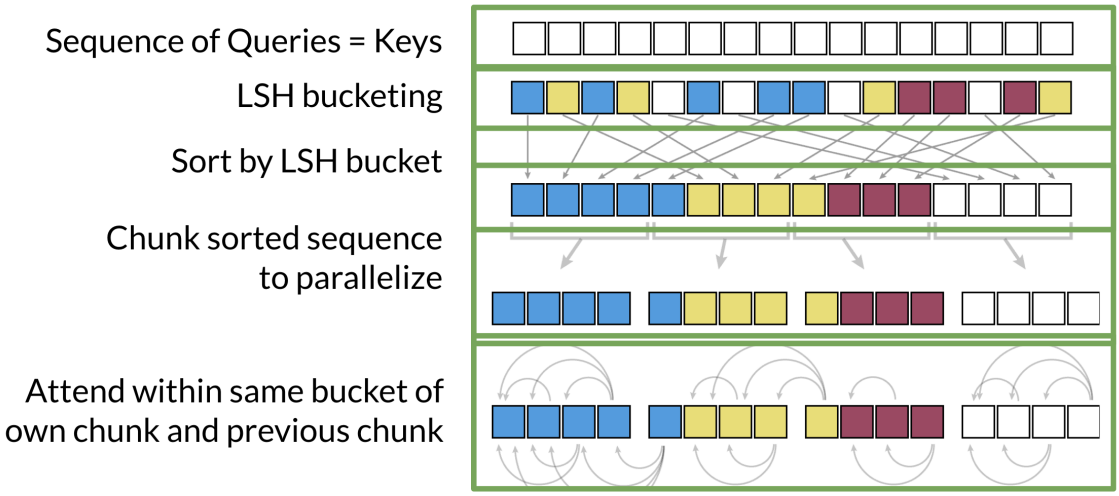
- Attention computes $d(q, k_i)$ for i from 1 to n which can be slow
- Faster *approximate* uses locality sensitive hashing (LSH)
- Locality sensitive: if q is close to k_i:
  hash(q) == hash(k_i)
- Achieve by randomly cutting space
  hash(x) = sign(xR) R: [d, n_hash_bins]



To do so, you hash q and the keys. This will put similar vectors in the same bucket that you can use. The drawing above shows the lines that separate the buckets. Those could be seen as the planes.

$$A(Q, K, V) = softmax(QK^T)V$$

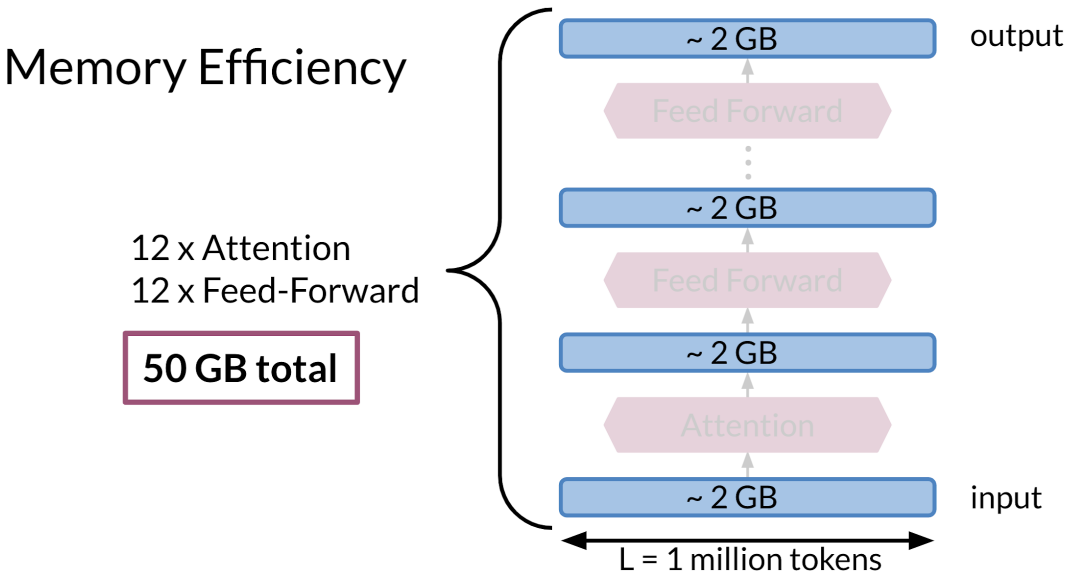
Once you hash Q and K you will then compute standard attention on the bins that you have created. You will repeat the same process several times to increase the probability of having the same key in the same bin as the query.



Given the sequence of queries and keys, you hash them into buckets. Check out Course 1 Week 4 for a review of the hashing. You will then sort them by bucket. You split the buckets into chunks (this is a technical detail for parallel computing purposes). You then compute the attention within the same bucket of the chunk you are looking at and the previous chunk. Why do you need to look at the previous chunk?

Motivation for Reversible Layers: Memory!

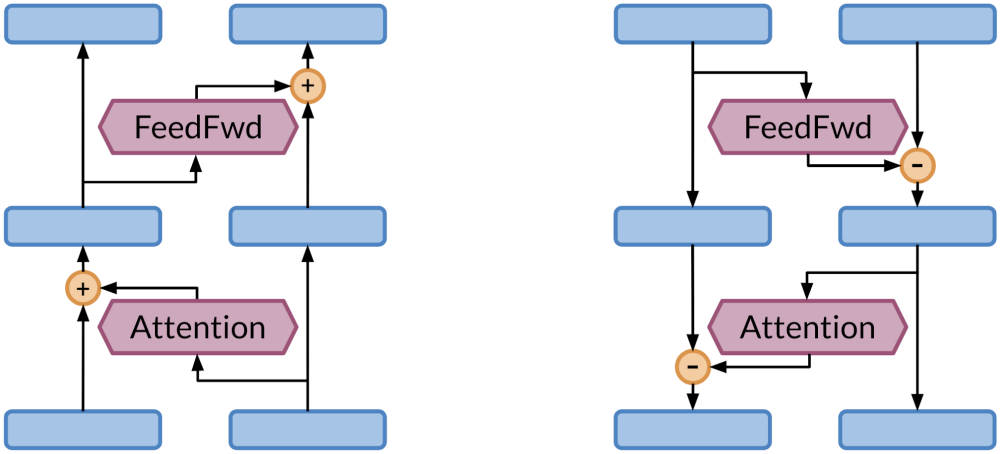
Every time you run a forward propagation, you need to compute the back propagation to update the weights. The biggest issue with doing this is that you have to store the weights to be able to compute the back-prop. With these very large models, that could be a lot of memory.



For example in the model above it requires 2GB to compute the Attention and 2GB for the feed forward. You have 12 layers for attention and 12 layers for the feedforward. That is equal to 12 * 2 + 12*2 + 2 (for the input) = 50 GB. That is a lot of memory. In the next video you will learn how to solve such problems.

Reversible Residual Layers

Reversible residual layers allow you to reconstruct the forward layer from the end of the network. Usually you have two similar branches in the network that you use to compute the network.



In the left picture, you have the forward propagation. One side of the network is used as input and the other is used for the attention. On the left side, the same thing is happening but in the opposite direction.

Standard Transformer:

$$y_a = x + \text{Attention}(x) \qquad y_b = y_a + \text{FeedFwd}(y_a)$$

Reversible:

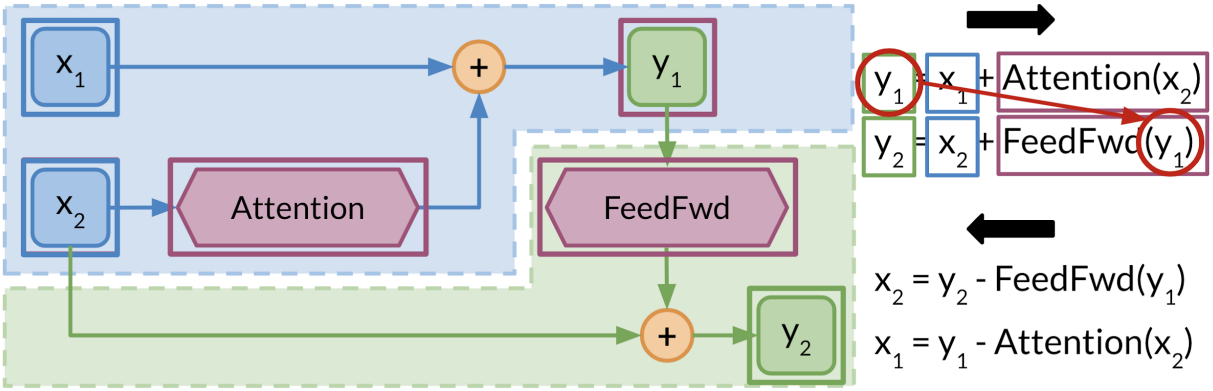
$$y_1 = x_1 + \text{Attention}(x_2) \qquad y_2 = x_2 + \text{FeedFwd}(y_1)$$

Recompute x_1, x_2 from y_1, y_2:

$$x_1 = y_1 - \text{Attention}(x_2) \qquad x_2 = y_2 - \text{FeedFwd}(y_1)$$

Take a few minutes and try to understand the equations above. You basically make use of the two branches of the network. When coming back for the back propagation, you only need the y's to compute x_2 and then you can use x_2 along with y_1 to compute x_1. Pretty neat! Now you

don't have to store the weights, because you can just compute them from scratch. This image shows you a visualization of what is happening.



Reformer

The reformer allows you to fit up to 1 million tokens on a single 16 gigabyte GPU. It is designed to handle context windows of up to 1 million words. It combines two techniques to solve the problems of attention and memory allocation which are bottlenecks for the transformer networks.

Reformer uses locality sensitive hashing, which you saw earlier in this specialization, to reduce the complexity of attending over long sequences. It also uses reversible residual layers to more efficiently use the memory available. In the picture below you can see how the reformer performs when compared to a normal full-attention model.