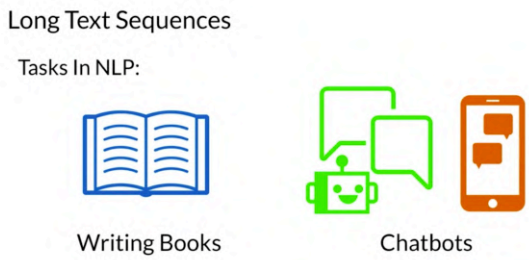


Welcome, you're an expert on transformers by now. In this week I will push the transformer model even further, to make it work on really long sequences. First, let's see examples of tasks where this is needed, let's dive in. >> Tasks with long sequences in NLP include many applications, and are becoming increasingly sophisticated. Well known use cases include writing books and storytelling, or building intelligent agents for conversations like chatbots.

In fact, it's becoming increasingly difficult to tell if the book here reading was written by a human or AI, or if the conversation you're having is rael or with the computer. The models behind these long sequence applications presents considerable challenges, largely due to their size in training. Many are based on the GPT-3 transformer model, which is just a larger version of GPT-2 that you already know from previous weeks. But these models can take industrial scale compute and cost a lot of money to train.

In this section, we'll see some of the bottlenecks in these larger transformer models, and solutions you can use to make them trainable for you. I'll teach you about the re former model known as the reversible transformer, I'll explain why it's important and how it works. Then you will use this new knowledge to build and train a real working chatbot in this week's assignments. You'll be able to ask it anything you like on almost any topic at all.

Processing long text sequences is at the core of building chatbots. A chatbot model needs to use all the previous pieces of the conversation as inputs for the next reply, this can make for some really big context Windows. But what exactly is a chatbot, and how does it relate to say context based question and answer, and closed loop based question and answer that you learned in the previous weeks. To recap the context based Q and A needs both a question and relevant text from where it's going to retrieve an answer.

Closed loop Q and A, however, doesn't need extra text to go along with a question or prompt from a human. All the knowledge is stored in the weights of the model itself during training. And this is how your chat bot will work, and what you learn to do this week. So you'll use all the knowledge of NLP you built over the previous three courses, and the power of transformers over long sequences to build your own chatbots.

Optional AI Storytelling

Dragon model for Dungeon is based on GPT-3. It generates an interactive story based on all previous turns as inputs. That makes for a task that uses very long sequences. Check it out!

https://play.aidungeon.io/main/landing

Transformer Issues

- Attention on sequence of length L takes L^2 time and memory

  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)

- N layers take N times as much memory

  GPT-3 has 96 layers and new models will have more

If you try to run a large transformer on the long sequence, you just run out of memory. In this section, you'll understand why, and identify the two main parts responsible for that.

Transformers can get big, and that introduces a lot of engineering challenges. So let me show you what I mean. Attention on a sequence of length L, takes L squared time and memory. For example, if you're doing attention on two sentences of length L, you need to compare each word in the first sentence to each word in the second sentence, which is L times L comparisons or L squared.

If you have N layers of attention, then it takes N times as much memory. GPT-3, for example, already has 96 layers, and new models will have more, even modern GPUs can struggle with this kind of dimensionality

Transformer Issues

- Attention on sequence of length L takes L^2 time and memory

  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)

- N layers take N times as much memory

  GPT-3 has 96 layers and new models will have more

For example, if L equals 100, then L squared equals 10,000.

Transformer Issues

- Attention on sequence of length L takes L^2 time and memory

  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)

- N layers take N times as much memory

  GPT-3 has 96 layers and new models will have more

If L equals 10,000, then L squared equals a 100,000,000

Transformer Issues

- Attention on sequence of length L takes L^2 time and memory

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)

which at 10,000,000 operations per second is already taking 10 seconds to compute.

- N layers take N times as much memory

  GPT-3 has 96 layers and new models will have more

Attention Complexity

- 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

Recall attention is softmax of Q times K transpose times V, where Q is the query, K the key, and V, the value. Q, K and V are all of dimension L by d_model, where L is the length of the sequence, and d_model is the depth of attention. So Q times K transpose will be L by L or L squared. However, 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 English to German, 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.

Memory with N Layers

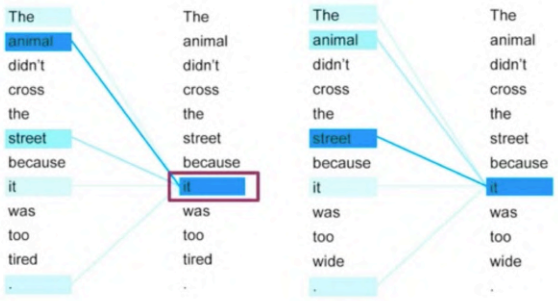
- Activations need to be stored for backprop
- Big models are getting bigger
- Compute vs memory tradeoff

The more layers a model has, the more memory it needs. This is because you need to store the forward pass activations for backprop. You can overcome this memory requirements by recomputing activations, but it needs to be done efficiently to minimize taking too much extra time. GPT-3 for example, which already has 96 layers, would take a very long time to recompute activations, and these models will continue to get bigger. So what you need is a way to speed up this re-computation, so that it's efficient to use less memory and I'll show you how to do that next.

The first part that contributes to the complexity of transformer on long sequences is the dot product attention. To improve it, I will show you how to use locality sensitive hashing, which you've learned before.

What does Attention do?

Select Nearest Neighbors (K,Q) and return corresponding V

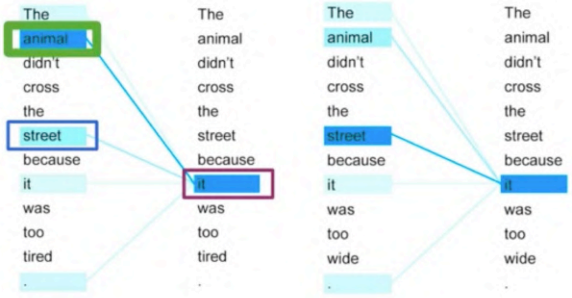


In this picture you can see what attention is doing.

image ©
(Transformer: A Novel Neural Network Architecture for Language Understanding.)

What does Attention do?

Select Nearest Neighbors (K,Q) and return corresponding V



Take the word it attention is focused on certain words to determine if it refers to the streets or to the animal. For example, in the sentence the animal didn't cross the street because it was too tired, it refers to the animal.

image ©
(Transformer: A Novel Neural Network Architecture for Language Understanding.)

What does Attention do?

Select Nearest Neighbors (K,Q) and return corresponding V

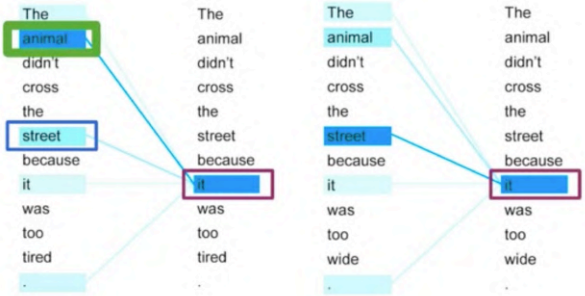
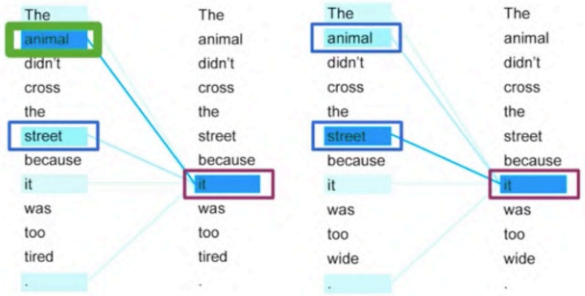


image ©
(Transformer: A Novel Neural Network Architecture for Language Understanding.)

What does Attention do?

Select Nearest Neighbors (K,Q) and return corresponding V

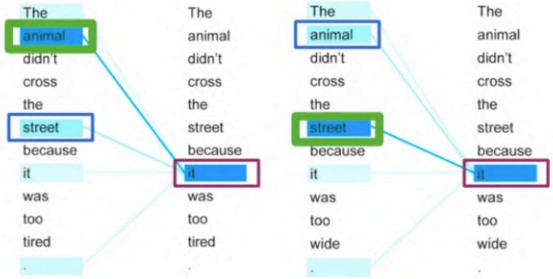


Or in the second example, the animal didn't cross the street because it was too wide, the only difference being the last word.

image ©
(Transformer: A Novel Neural Network Architecture for Language Understanding.)

What does Attention do?

Select Nearest Neighbors (K,Q) and return corresponding V



It again refers to either the streets or the animal, but in this case it refers to the streets. In both attentions you can see that it is either the animal or the streets. You only need to look at the nouns because it can only refer to the nouns, not all of the words. A pronoun is a word that substitutes for a noun. Its for animal or its for streets for example, as I just showed you. So when working with pronouns, you know you only need to look at the other nouns, and you can ignore the other words like it and was and tired. This is an example of why you only want to work with the nearest neighbors to speed up the attention parts.

image ©
(Transformer: A Novel Neural Network Architecture for Language Understanding.)

Nearest Neighbors

Course:

Natural Language Processing with Classification and Vector Spaces

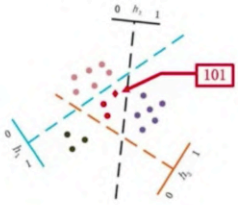
Lessons:

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

Nearest Neighbors

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)



Using locality sensitive hashing, you can hash both the query q and (q, k).

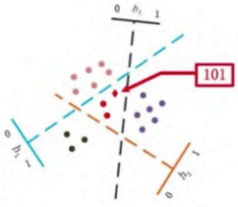
This helps you group similar query and key vectors together, just like the nouns with pronouns. Examples you saw before, then you only run attention on keys that are in the same hash buckets as the query.

When choosing the hash you want to make the buckets roughly the same size. You know that hash of x is the sign of x times R where R is random with size of d for dimension times the number of hash bins. And design tells you which side of the plane the hash will be on. The process is then repeated depending on the number of hashes that you have.

Nearest Neighbors

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]



LSH Attention

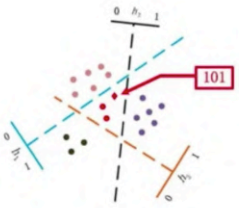
You also already know how standard attention works. Well, let me show you how to speed this up using LSH attention. First you hash Q and K, then perform standard attention, but within the same-hash bins. This reduces the search space for each K to the same LSH bucket as Q. You can repeat this process multiple times to increase the probability of finding Q and K in the same bin. And this can be done efficiently to take advantage of parallel computing.

Standard Attention:

$$\mathrm{A}(Q, K, V) = \mathrm{softmax}(QK^T)V$$

LSH Attention:

- Hash Q and K
- Standard attention within same-hash bins
- Repeat a few times to increase probability of key in the same bin

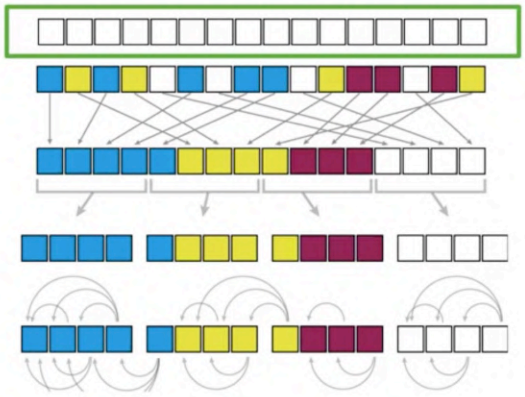


LSH Attention

image © (Reformer: The Efficient Transformer)

Sequence of Queries = Keys

Now I'll show you how to integrate LSH into attention layers. To start, you modify the model so that it outputs a single vector at each position, which serves both as a query and a key. This is called QK attention and performs just as well as regular attention.



LSH Attention

image © (Reformer: The Efficient Transformer)

Sequence of Queries = Keys

LSH bucketing

Next you might beach vector to a bucket with LSH.

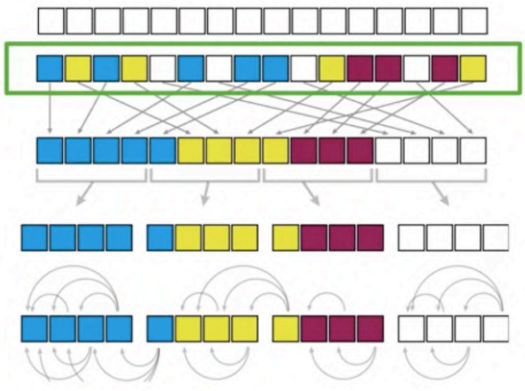


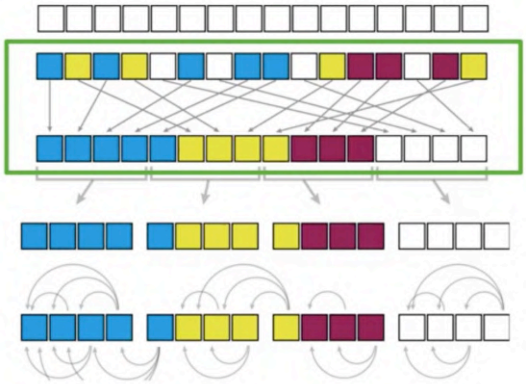
image © (Reformer: The Efficient Transformer)

LSH Attention

Sequence of Queries = Keys

LSH bucketing

Sort by LSH bucket



Then you sort the vectors by LSH bucket, and finally you do attention only in each bucket. You could do this one bucket at a time, but that doesn't take advantage of hardware parallelism. Instead, I'll show you how to do a batch computation.

image © (Reformer: The Efficient Transformer)

LSH Attention

Sequence of Queries = Keys

LSH bucketing

Sort by LSH bucket

Chunk sorted sequence to parallelize

The first step for batching is to split the sorted sequence into fixed size chunks. This allows for some parallel computation

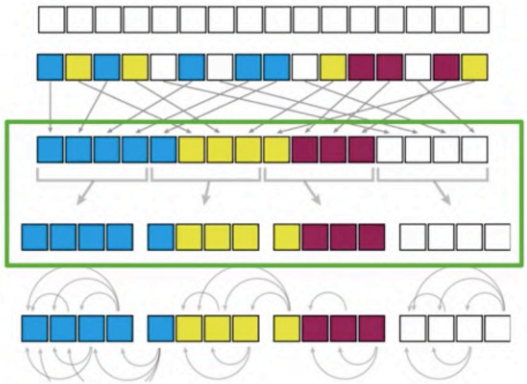


image © (Reformer: The Efficient Transformer)

LSH Attention

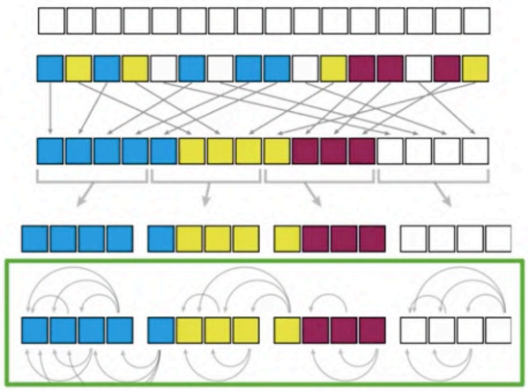
Sequence of Queries = Keys

LSH bucketing

Sort by LSH bucket

Chunk sorted sequence to parallelize

Attend within same bucket of own chunk and previous chunk



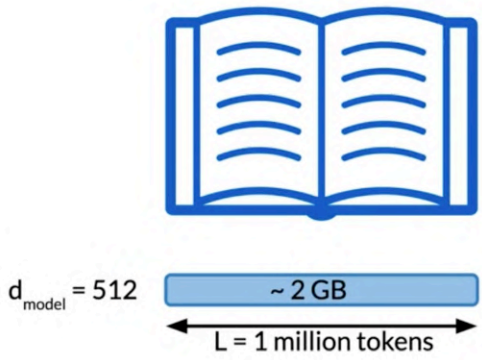
Then you let each chunk attend within itself. And the adjacent chunks discovers the case with a hash bucket that is split over more than one chunk, like you see for the blue, yellow, and magenta buckets here. And that's the core of LSH attention.

One final point to consider is that LSH is a probabilistic, not deterministic model. This is because of the inherent randomness within the LSH algorithm. Meaning that the hash can change along with the buckets a vector finds itself map tune.

Memory Efficiency

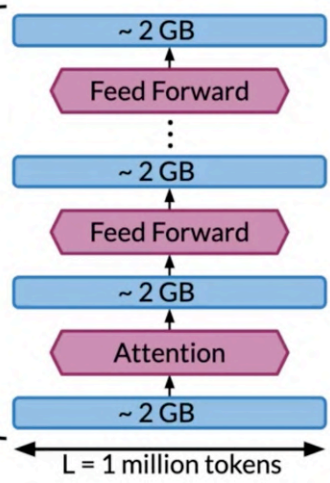
If you want to run a transformer over the entire text of a book, you might have around 1 million tokens to process. And that requires paying attention, unattended to memory management.

For example, if you want to run a transformer over this book, you might have inputs of 1 million tokens to process. And each of these tokens will have an associated feature vector of some size, for example, 512. This means that just the input for the model is already 2GB in size. On a 16 gigabyte GPU, this is one-eight of your total memory budget, and you haven't even touched the layers yet.



Memory Efficiency

12 x Attention
12 x Feed-Forward

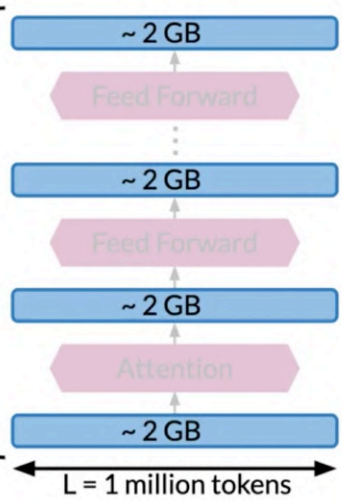


Speaking of layers, a transformer has two types of layers, attention layers an feedforward layers. A model might have 12 of each type of layer. This is the case for example, with bed space. The size of the inputs and outputs for each layer will also be 2GB. To do backpropagation through this model, the forward path will need to store some intermediate quantities in memory. Suppose all you save is the activations from the boundaries between each layer and nothing from within the individual layers themselves.

Memory Efficiency

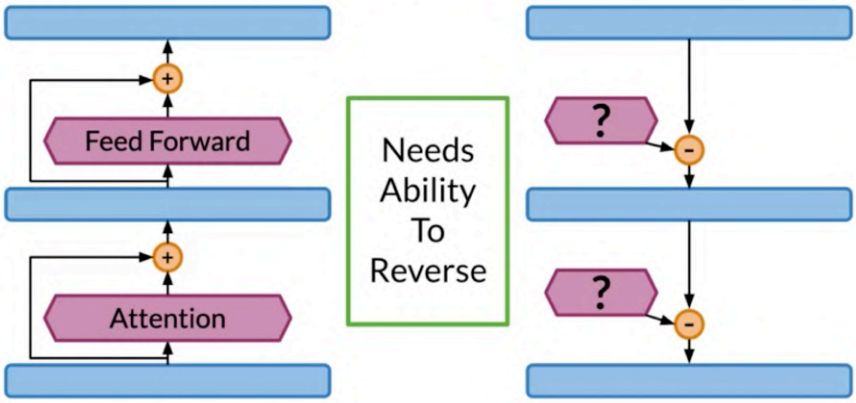
12 x Attention
12 x Feed-Forward

50 GB total



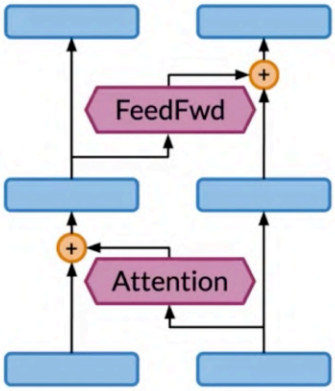
Already you can see an issue, saving these activations would use around 50 GB of memory, which wont fit on a single device. Keep in mind that the latest transformers are much deeper than the 12 layer design I'm using here. This is the first fundamental efficiency challenge, transformers phase. Memory usage also grows linearly with the number of layers, so there's no way to scale to the million token regime. Fortunately, there is a solution where you don't need to save anything for the backward path, and I'll show you that next.

Residual Blocks in Transformer



The transformer network precedes by repeatedly adding residuals to the hidden states. To run it in reverse, you can subtract the residuals in the opposite order, starting with the outputs of the model. But in order to save memory otherwise used to store the residuals, you need to be able to recompute them quickly instead

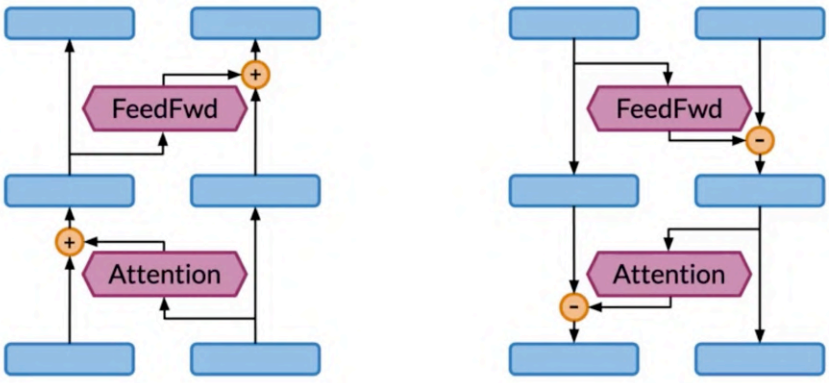
Reversible Residual Blocks



This is where reversible residual connections come in. The key idea is that you start with two copies of the model inputs, then at each layer you only update one of them. The activations that you don't update will be the ones used to compute the residuals.

where this configuration you can now run the network in reverse. Layer 1 is attention and layer 2 is feedforward. The activations in the model are now twice as big, but you don't have to worry about caching for the backwards pass.

Reversible layers



Reversible layers equations

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:

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

The standard transformer equations give Y_a is equal to x plus attention of X, and Y_b is equal to Y_a plus feedforward of Y_a. This is the normal residual connection.

In the reversible case, you will have Y_1 equals x_1 plus attention of x_2, and Y_2 equals x_2 plus feedforward of Y_1.

Then to save the memory, you can reconstruct the inputs x_1 and x_2 as follows, x_1 equals Y_1 minus attention of x_2 and x_2 equals Y_2 minus feedforward of Y_1. Feel free to take a moment here for a moment to make sure you understand what's happening.

Reversible layers equations

Standard Transformer:

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

Reversible:

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

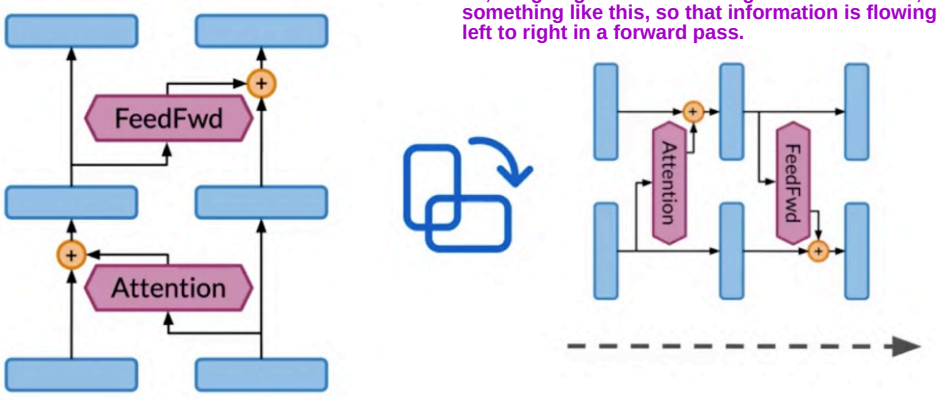
Recompute x_1, x_2 from y_1, y_2:

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

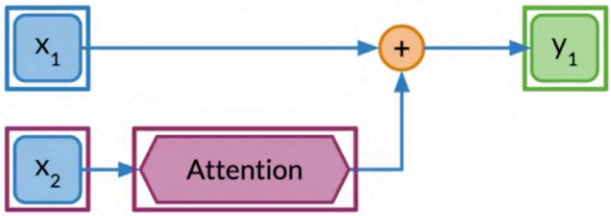
ide
er
ne
of
ute
ay,
w
x_1

Reversible layers equations

Let me show you how these new formulas fits into this reversible layers illustration I've already shown you. To do so, I'm going to rotate the diagram onto its side, something like this, so that information is flowing from left to right in a forward pass.



Reversible layers equations

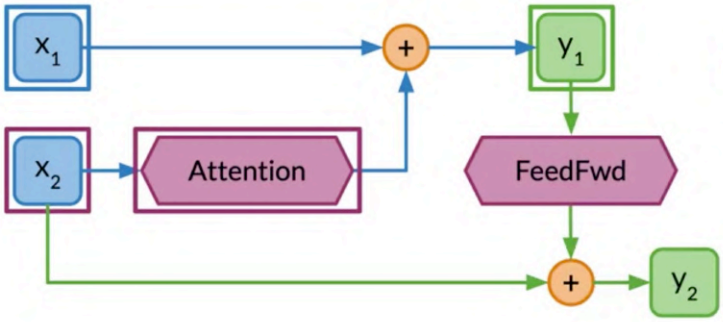


Restating the equations I showed you earlier, the first step is to calculate Y_1 equals x_1 plus the attention of x_2.

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

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

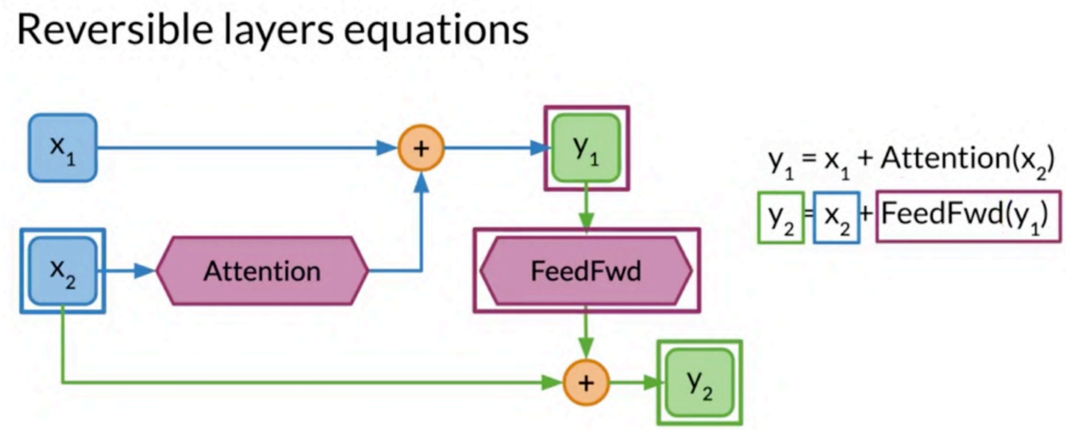
Reversible layers equations

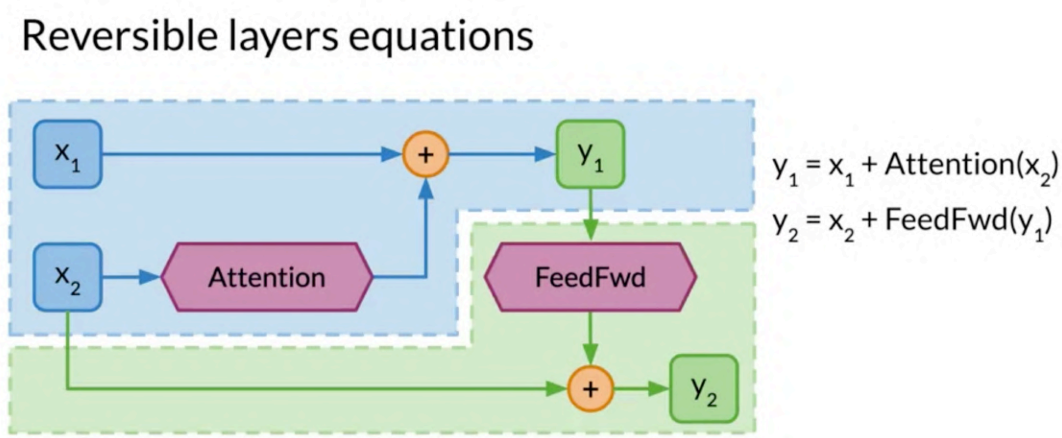


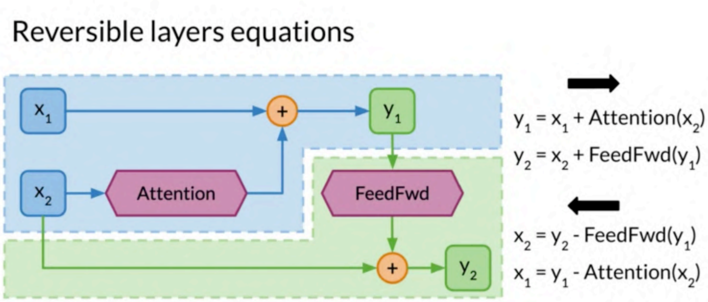
Then after you've done this, the second step is to calculate Y_2 with the second formula, which has a dependency on Y_1, and requires a few extra parts in the illustration that weren't used in the first step.

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

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







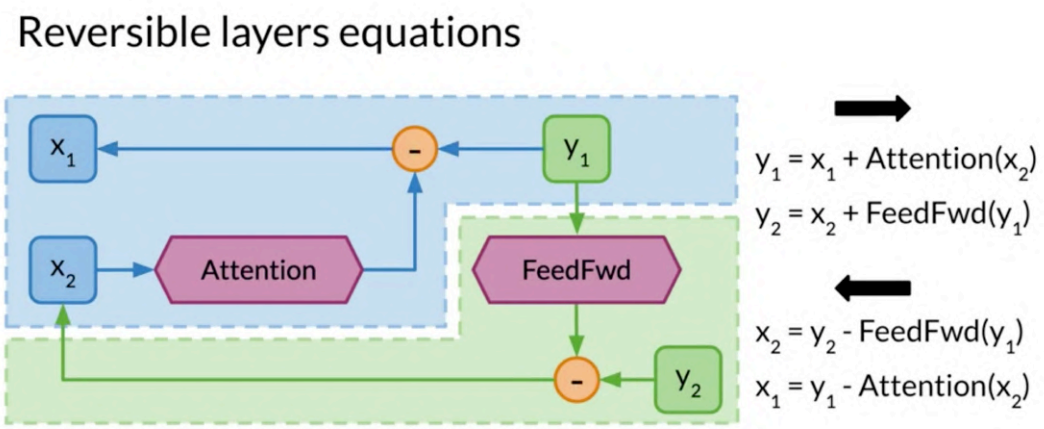
That's a forward pass for irreversible residual block. It's combined standard attention and feedforward residual layers from a regular transformer into a single reversible residual block, and there is nothing to be saved in memory except the Y_1 and Y_2 of the output layer instead of activations for every individual layer. Memory saved.

Now, I'll show you how to recompute x_1 and x_2 from Y_1 and Y_2 for a backward pass.

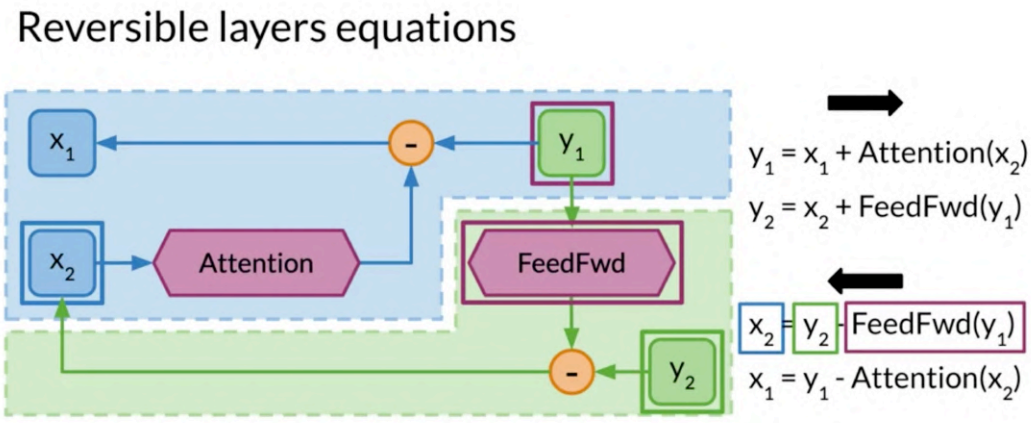
First thing to notice is that I'm going to calculate x_2 before x_1, the reason for which I'll explain soon.

First, I'll reverse some of the arrows in the illustration.

The reverse direction is to indicate the information is flowing backwards. Notice I also changed the plus signs in the orange circles to be minuses to indicate subtraction will now be taking place.

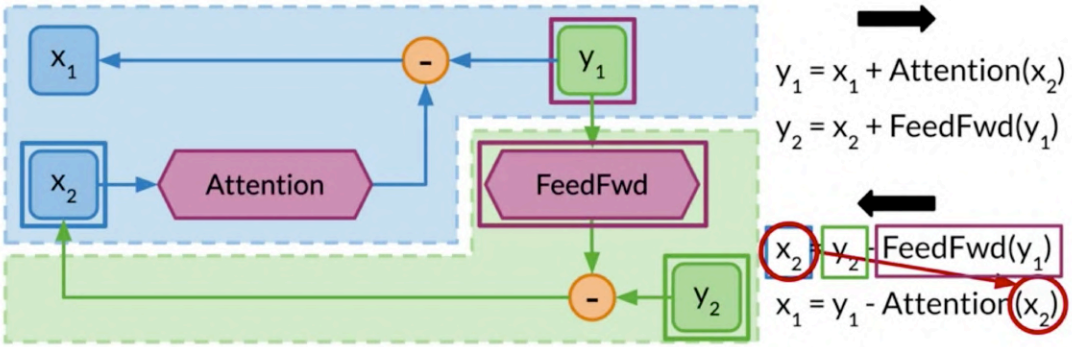


The first step is to calculate x_2 equals Y_2 minus feedforward of Y_1, and great work, you just calculated an input x_2 from the two outputs of the forward pass.



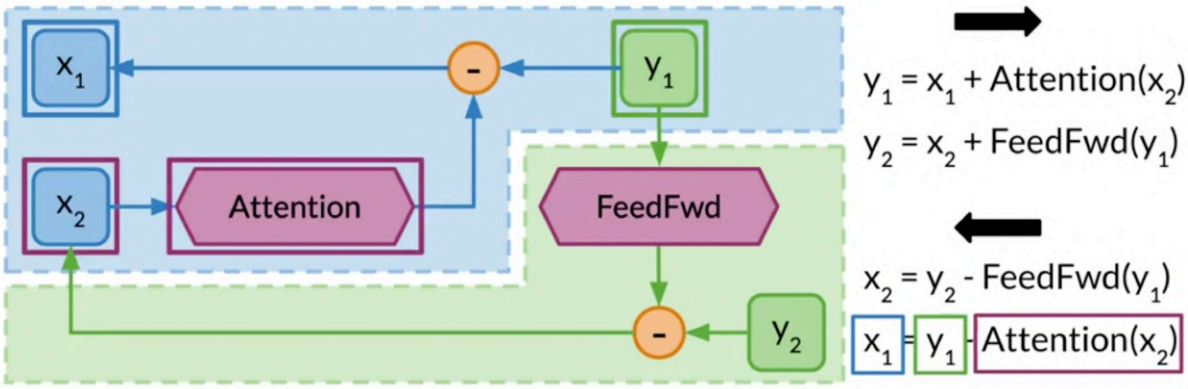
The second step is to calculate x_1. The formula for x_1 has a dependency on x_2 that you just calculated, similar to the Y_2 dependency on Y_1 in the forward pass formulas above.

Reversible layers equations

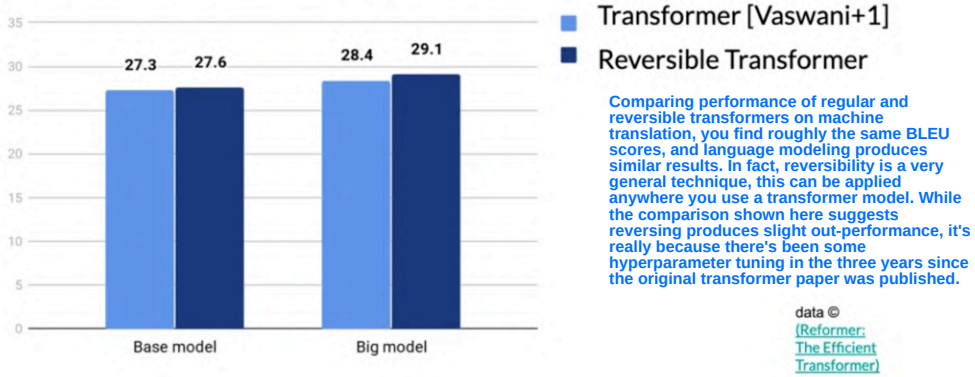


So x_1 equals Y_1 minus the attention of x_2 that you already calculated in the first step. You now know how to compute a backward pass for a residual layer and a transformer model without the need for saving memory hungry activations in the forward pass. I showed you how to do this using reversible residual blocks.

Reversible layers equations



Reversible Transformer: BLEU Scores



Comparing performance of regular and reversible transformers on machine translation, you find roughly the same BLEU scores, and language modeling produces similar results. In fact, reversibility is a very general technique, this can be applied anywhere you use a transformer model. While the comparison shown here suggests reversing produces slight out-performance, it's really because there's been some hyperparameter tuning in the three years since the original transformer paper was published.

data © (Reformer: The Efficient Transformer)

Reformer

The Reversible Transformer

L = 1 million tokens

1 GPU
(16 GB)

You learned how to solve the complexity and memory issues with transformer on long sequences. Now you'll put the solutions together and create an efficient transformer model called reformer.

Let me introduce you to reformer, the reversible transformer. Using this model, you can fit up to 1 million tokens on a single 16 gigabyte GPU. That's enough to fit an entire book, like all of crime, punishments, for example.

Reformer

- LSH Attention
- Reversible Layers

The reformer is a transformer model designed to handle context windows of up to 1 million words. It's combines two techniques to solve the crucial problems of attention, and memory allocation that limit transformers, application to long contexts windows. 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.

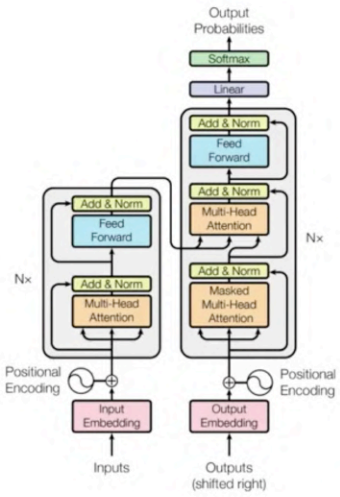
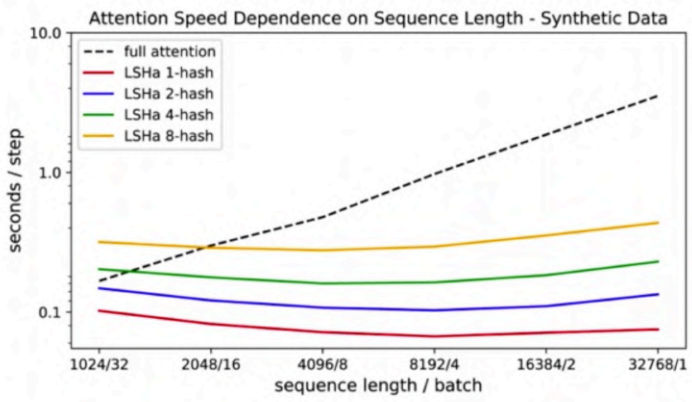


image © (Attention Is All You Need)

Reformer



These plots from the reformer paper, highlights how standard attention takes longer as sequence length increases. However, LSH attention takes roughly the same amount of time, as sequence length increases. The only difference in the colored lines is the number of hashes. With more hashes taking slightly longer than fewer hashes, regardless of the sequence length.

image © (Reformer: The Efficient Transformer)

In this week's assignments, you will harness the reformer models power over large contexts windows, to build a working Chatbots that you interact with. I'll show you how to build and train a reformer model, on the MultiWOZ datasets using the Trax framework from the Google Brain Team. MultiWOZ is a very large datasets of human conversations, covering multiple domains and topics. When finished, you'll be able to ask your Chatbots questions, about almost anything, and it'll answer you. You have no seen how a reformer is built. This is a transformer that can handle very long contexts. Let's put it to use to build a Chatbot.

Optional Transformers beyond NLP

Jukebox - A neural network that generates music!

https://openai.com/blog/jukebox/

GPT-3 Can also help with auto-programming!

https://beta.openai.com/?app=productivity&example=4_2_0

References

This course drew from the following resources:

- Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer (Raffel et al, 2019)

- Reformer: The Efficient Transformer (Kitaev et al, 2020)

- Attention Is All You Need (Vaswani et al, 2017)

- Deep contextualized word representations (Peters et al, 2018)

- The Illustrated Transformer (Alammar, 2018)

- The Illustrated GPT-2 (Visualizing Transformer Language Models) (Alammar, 2019)

- BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (Devlin et al, 2018)

- How GPT3 Works - Visualizations and Animations (Alammar, 2020)