I made a transformer by hand (no training!)

September 11, 2023

> *Intended audience: some familiarity with language models, interested in how transformers do stuff (but might be a bit rusty on matrices)*

I've been wanting to understand transformers and attention better for awhile now—I'd read The Illustrated Transformer, but still didn't feel like I had an intuitive understanding of what the various pieces of attention were *doing*. What's the difference between `q` and `k`? And don't even get me started on `v`!

So I decided to make a transformer to predict a simple sequence (specifically, a decoder-only transformer with a similar architecture to GPT-2) manually—not by training one, or using pretrained weights, but instead by *assigning each weight, by hand*, over an evening. And—it worked! I feel like I understand transformers much better now, and hopefully after reading this, so will you.

The basic outline of what we need to do is:

- Pick a task—not too easy, but not as hard as "write fluent English text", since that needs at least a few hundred million parameters!
- Pick our model dimensions for the task
- Design position (`wpe`) and token (`wte`) embedding weights
- The hard part—design a transformer block to do the actual computation!
  - First, we'll need a `c_attn` layer to generate `q`, `k`, and `v` matrices
  - Then, we'll need a `c_proj` layer to project that result back to an embedding
- Finally, use the token embedding weights from before (`wte`) to project that back to a set of next token logits!

▶ Quick refresher on matrices

🔗 Picking a task

I originally started with just predicting sequences like `"abababababab"`, but quickly realized that because transformers predict the *shifted* sequence, this would be too easy—it wouldn't require using the position embeddings. (Instead, the algorithm would just be "If I am an `a`, then predict `b`, otherwise predict `a`.")

The more involved task I settled on was predicting the sequence `"aabaabaabaab..."` (that is to say, `(aab)*`), which requires querying the previous *two* tokens to know if the output should be an `a` (previous tokens are `ab` or `ba`) or a `b` (previous tokens are `aa`).

🔗 Sidetrack: Designing a tokenization scheme

Since we only have two symbols to worry about, I used a very simple scheme where `a` is `0` and `b` is `1`:

```
CHARS = ["a", "b"]
def tokenize(s): return [CHARS.index(c) for c in s]
def untok(tok): return CHARS[tok]

# examples:
tokenize("aabaa") # => [0, 0, 1, 0, 0]
untok(0) # => "a"
untok(1) # => "b"
```

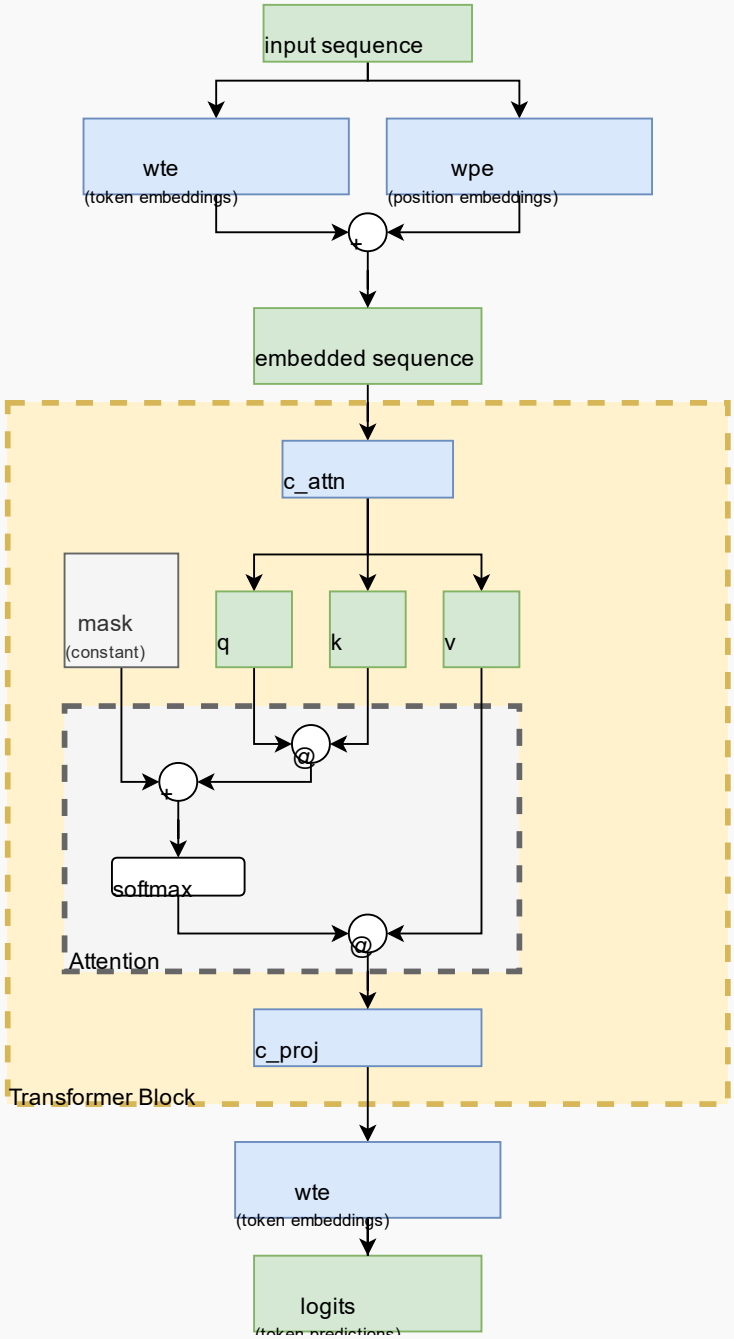
🔗 Picking a model

I based the code for the model off jaymody's picoGPT implementation of GPT-2, with a few changes to make things simpler[1]. This left me with the following architecture for assigning weights. Don't worry if this doesn't make sense yet, I'll be explaining it as we go!

```
# based on https://github.com/jaymody/picoGPT/blob/main/gpt2.py (MIT license)

import numpy as np

def softmax(x):
  exp_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
```

```python
  return exp_x / np.sum(exp_x, axis=-1, keepdims=True)

# [m, in], [in, out], [out] -> [m, out]
def linear(x, w, b):
  return x @ w + b

# [n_q, d_k], [n_k, d_k], [n_k, d_v], [n_q, n_k] -> [n_q, d_v]
def attention(q, k, v, mask):
  return softmax(q @ k.T / np.sqrt(q.shape[-1]) + mask) @ v

# [n_seq, n_embd] -> [n_seq, n_embd]
def causal_self_attention(x, c_attn, c_proj):
  # qkv projections
  x = linear(x, **c_attn)  # [n_seq, n_embd] -> [n_seq, 3*n_embd]

  # split into qkv
  q, k, v = np.split(x, 3, axis=-1)  # [n_seq, 3*n_embd] -> 3 of [n_seq, n_embd]

  # causal mask to hide future inputs from being attended to
  causal_mask = (1 - np.tri(x.shape[0], dtype=x.dtype)) * -1e10  # [n_seq, n_seq]

  # perform causal self attention
  x = attention(q, k, v, causal_mask)  # [n_seq, n_embd] -> [n_seq, n_embd]

  # out projection
  x = linear(x, **c_proj)  # [n_seq, n_embd] @ [n_embd, n_embd] = [n_seq, n_embd]

  return x

# [n_seq, n_embd] -> [n_seq, n_embd]
def transformer_block(x, attn):
  x = x + causal_self_attention(x, **attn)
  return x

# [n_seq] -> [n_seq, n_vocab]
def gpt(inputs, wte, wpe, blocks):
  # token + positional embeddings
  x = wte[inputs] + wpe[range(len(inputs))]  # [n_seq] -> [n_seq, n_embd]

  # forward pass through n_layer transformer blocks
  for block in blocks:
    x = transformer_block(x, **block)  # [n_seq, n_embd] -> [n_seq, n_embd]

  # project to vocab
  return x @ wte.T  # [n_seq, n_embd] -> [n_seq, n_vocab]
```

This roughly corresponds to the following architecture diagram:



🔗 Picking the model dimensions

There were 3 model parameters to choose here:

- Context length
- Vocabulary size
- Embedding size

Context length is the maximum number of tokens the model will see at a time. Theoretically

this task only needs the previous 2 tokens—but let's go with 5 to make it a little more difficult, since then we'll also need to ignore the irrelevant tokens.

Vocabulary size is the number of distinct tokens the model will see. In a real model, there are tradeoffs between generalization, number of distinct tokens to learn, context length usage, etc. However, our task is much simpler, so in our model, we'll only need two tokens: `a` (`0`) and `b` (`1`).

Embedding size is the size of the vectors that the model will learn for each token/position, and will also use internally. I picked 8 rather arbitrarily, which ended up being exactly the size required :-)

In summary,

```
N_CTX = 5
N_VOCAB = 2
N_EMBED = 8
```

🔗 Designing the embedding weights

The first thing that happens is the list of token ids (`[0, 1, 0, ...]`) gets turned into a `seq_len x embedding_size` matrix that combines the position and type of each token:

```
def gpt(inputs, wte, wpe, blocks):  # [n_seq] -> [n_seq, n_vocab]
  # token + positional embeddings
  x = wte[inputs] + wpe[range(len(inputs))]  # [n_seq] -> [n_seq, n_embd]
  ...
```

That means the first things we need to design are `wte` (weights for token embeddings) and `wpe` (weights for position embeddings[2]). We'll use a 1-hot scheme for each, meaning each class of thing has a `1` in a unique position.

We'll use first five embedding elements for the position 1-hot embedding: position 0 will be represented as `[1, 0, 0, 0, 0]`, position 1 as `[0, 1, 0, 0, 0]`, and so on up to position 4 as `[0, 0, 0, 0, 1]`.

Likewise, we'll use the next two embedding elements for the token id 1-hot embeddings:

token `a` will be represented as `[1, 0]`, and token `b` as `[0, 1]`.

```
MODEL = {
  "wte": np.array(
    # one-hot token embeddings
    [
      [0, 0, 0, 0, 0, 1, 0, 0],  # token `a` (id 0)
      [0, 0, 0, 0, 0, 0, 1, 0],  # token `b` (id 1)
    ]
  ),
  "wpe": np.array(
    # one-hot position embeddings
    [
      [1, 0, 0, 0, 0, 0, 0, 0],  # position 0
      [0, 1, 0, 0, 0, 0, 0, 0],  # position 1
      [0, 0, 1, 0, 0, 0, 0, 0],  # position 2
      [0, 0, 0, 1, 0, 0, 0, 0],  # position 3
      [0, 0, 0, 0, 1, 0, 0, 0],  # position 4
    ]
  ),
  ...: ...,
}
```

If we encode an entire sequence `"aabaa"` with this scheme, we get the following embedding matrix of shape `5 x 8` (`seq_len x embedding_size`):

'a'		1	0	0	0	0	1	0	0	position 0, token `a`
'a'		0	1	0	0	0	1	0	0	position 1, token `a`
'b'	→	0	0	1	0	0	0	1	0	position 2, token `b`
'a'		0	0	0	1	0	1	0	0	position 3, token `a`
'a'		0	0	0	0	1	1	0	0	position 4, token `a`

This gives us the embedding matrix the rest of the model will work with, until it gets projected back to vocabulary-space at the end. Note that the 7th position isn't used—that will be our scratch space in the transformer block. Speaking of...

🔗 Designing the transformer block

The model code supports multiple transformer blocks, but we'll only be using one. Our block consists of two parts: an attention head, and a linear network that will project the attention result matrix back to the common `seq_len x embedding_size` matrix the network usually works with.

Let's focus on the attention head first.

🔗 Designing the attention head

The attention head is the main part of the transformer block—it's where attention happens, and as we all know, Attention Is All You Need!

(Most transformers nowadays have multiple parallel heads per transformer block ("multi-head attention"), which I believe is where the "attention head" terminology comes from, but we'll only use one for simplicity. For our purposes, "attention head" = attention layer.)

As you might expect, the main part of the attention head is attention, which in our case is defined as:

```
# [n_q, d_k], [n_k, d_k], [n_k, d_v], [n_q, n_k] -> [n_q, d_v]
def attention(q, k, v, mask):
  return softmax(q @ k.T / np.sqrt(q.shape[-1]) + mask) @ v
```

The parameters here are:

- `q`, or "query"
- `k`, or "key"
- `v`, or "value"
- and `mask`, which is a non-learned parameter used to prevent the model from cheating during training by looking at future tokens, which will make more sense later.

These are usually explained by analogy to a dictionary lookup:

```
DICT = { k: v }
DICT[q]
```

...but I never found this explanation very useful. I think working through how attention actually works helped it make a lot more sense—so let's do that!

In our model, the weights for attention are defined in `c_attn`:

```
Lg = 1024 # Large

MODEL = {
  ...: ...,
  "blocks": [
    {
      "attn": {
        "c_attn": {  # generates qkv matrix
          "b": np.zeros(N_EMBED * 3),
          "w": np.array(
            # this is where the magic happens
            # fmt: off
            [
              [Lg, 0., 0., 0., 0., 0., 0., 0.,  # q
                1., 0., 0., 0., 0., 0., 0., 0.,  # k
                  0., 0., 0., 0., 0., 0., 0., 0.], # v
              [Lg, Lg, 0., 0., 0., 0., 0., 0.,  # q
                0., 1., 0., 0., 0., 0., 0., 0.,  # k
                  0., 0., 0., 0., 0., 0., 0., 0.], # v
              [0., Lg, Lg, 0., 0., 0., 0., 0.,  # q
                0., 0., 1., 0., 0., 0., 0., 0.,  # k
                  0., 0., 0., 0., 0., 0., 0., 0.], # v
              [0., 0., Lg, Lg, 0., 0., 0., 0.,  # q
                0., 0., 0., 1., 0., 0., 0., 0.,  # k
                  0., 0., 0., 0., 0., 0., 0., 0.], # v
              [0., 0., 0., Lg, Lg, 0., 0., 0.,  # q
                0., 0., 0., 0., 1., 0., 0., 0.,  # k
                  0., 0., 0., 0., 0., 0., 0., 0.], # v
              [0., 0., 0., 0., 0., 0., 0., 0.,  # q
                0., 0., 0., 0., 0., 0., 0., 0.,  # k
                  0., 0., 0., 0., 0., 0., 0., 1.], # v
              [0., 0., 0., 0., 0., 0., 0., 0.,  # q
                0., 0., 0., 0., 0., 0., 0., 0.,  # k
                  0., 0., 0., 0., 0., 0., 0., -1], # v
              [0., 0., 0., 0., 0., 0., 0., 0.,  # q
                0., 0., 0., 0., 0., 0., 0., 0.,  # k
                  0., 0., 0., 0., 0., 0., 0., 0.], # v
            ]
            # fmt: on
```

```
          ),
        },
        ...: ...,
      }
    }
  ]
}
```

That looks intimidating! But `c_attn` is just a regular fully-connected layer, with dimension `embed_size x (embed_size * 3)`. When we take its product with the `seq_len x embed_size` embedding matrix we calculated above, we get a matrix of size `seq_len x (embed_size * 3)`—called the `qkv` matrix. We then split that `qkv` matrix into 3 matrices of size `seq_len x embed_size`—`q`, `k`, and `v`.

```
def causal_self_attention(x, c_attn, c_proj):
  # qkv projections
  x = linear(x, **c_attn) # [n_seq, n_embd] -> [n_seq, 3*n_embd]
  # split into qkv
  q, k, v = np.split(x, 3, axis=-1) # [n_seq, 3*n_embd] -> 3 of [n_seq, n_embd]
  ...
```

(In the weights above, I formatted the `c_attn` weights on different lines to show which ones generate the `q`, `k`, and `v` parts of the `qkv` matrix.)

So let's just run the embedding matrix from before through `c_attn` and see what happens! If we take the embedding...

1	0	0	0	0	1	0	0	position 0, token `a`
0	1	0	0	0	1	0	0	position 1, token `a`
0	0	1	0	0	0	1	0	position 2, token `b`
0	0	0	1	0	1	0	0	position 3, token `a`
0	0	0	0	1	1	0	0	position 4, token `a`

...and run it through `embedding @ c_attn["w"] + c_attn["b"]`, we get the following `5 x 24` (`seq_len x (embed_size * 3)`) `qkv` matrix. (The thick lines here show where we will `np.split` this matrix in a

second):

q	q	q	q	q	q	q	q	k	k	k	k	k	k	k	k	v	v	v	v	v	v	v	v
1024	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1024	1024	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	1024	1024	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	-1
0	0	1024	1024	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1024	1024	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1

Let's ignore `v` for now and focus on `q` and `k`.

`k` might make sense—it's just the 1-hot position embeddings, isolated from the combined embedding matrix. You can think of this as what each token is "offering"—its position.

But what is `q`? If `k` is what each token is offering, then `q` is what each token is looking for—but what does that mean in practice?[3] Well, in attention, `k` will be transposed and multiplied with `q` in `q @ k.T`, producing a `seq_len x seq_len` matrix:

q							
1024	0	0	0	0	0	0	0
1024	1024	0	0	0	0	0	0
0	1024	1024	0	0	0	0	0
0	0	1024	1024	0	0	0	0
0	0	0	1024	1024	0	0	0

@

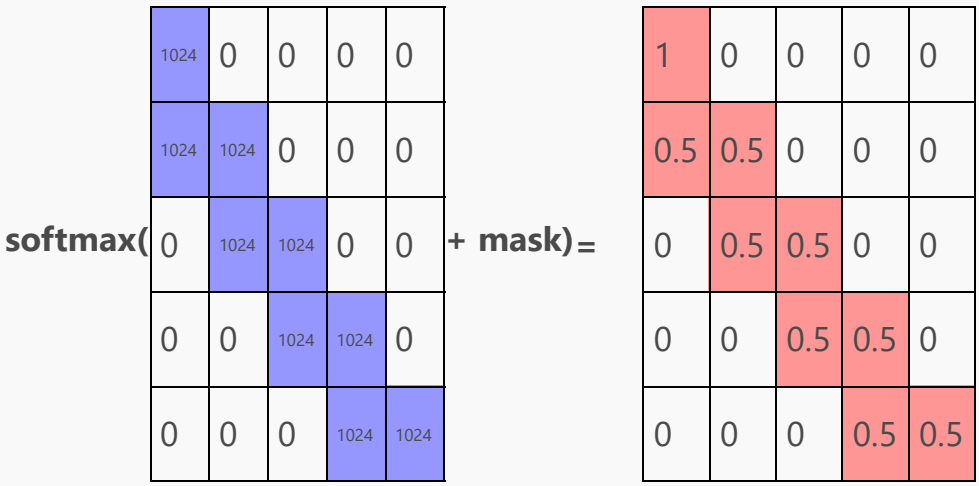
k.T				
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

=

1024	0	0	0	0
1024	1024	0	0	0
0	1024	1024	0	0
0	0	1024	1024	0
0	0	0	1024	1024

...and when we add the mask and softmax the whole thing (`softmax(q @ k.T + mask)`), suddenly it starts to make sense!

▶ Quick refresher on softmax

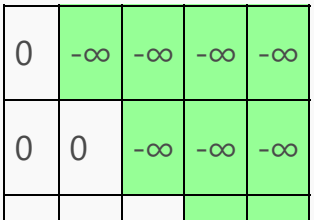


Think of each row as what's needed to generate the prediction for that row (e.g., row 0 is the information needed to generate the model's prediction after seeing token 0, the first token), and each column as which tokens to pay attention to. Furthermore, remember that the mask prevents the model from seeing into the future! (I'll explain how in a minute.)

That means the first prediction (row 0) isn't able to pay attention to any token except the first, so it puts 100% of its attention on that token—in other words, the first row has a `1` in the first column, and zeroes everywhere else.

But for all the other predictions, the model has at least two tokens to pay attention to, and for the `aabaabaab...` task, it doesn't ever need any more than two! So the model splits its attention for that token evenly between the latest two accessible (non-masked) tokens. That means the prediction for the second token (row 1) pays equal attention to token 0 and token 1, the prediction for the third token (row 3) pays equal attention to token 1 and token 2, and so on—so we see two non-zero cells, with `0.5` in each.

So what's the `mask` term added in `softmax(q @ k.T + mask)`? It's just the following matrix:



0	0	0	-∞	-∞
0	0	0	0	-∞
0	0	0	0	0

All this does is prevent the model from "cheating" during regular gradient-descent training—without this mask, the model would be incentivized to generate its prediction for the first token based on the value of the *second* token! By adding -∞[4], we force those positions down (see the softmax expando above for an explanation of why) so that the matrix coming out of `softmax` will have 0 in all masked (= future) token positions. This forces the model to actually *learn* how to predict those positions instead of cheating by looking ahead. In our case the mask isn't doing anything because this handmade transformer isn't designed to cheat, but leaving the mask in keeps things closer to the real GPT-2 architecture.

(Likewise, the scaling by `np.sqrt(q.shape[-1])` helps produce better gradients during real training, but doesn't affect our handmade transformer.)

But enough about the mask and scaling: all that really matters is the result of `softmax(q @ k.T / np.sqrt(q.shape[-1]) + mask)` is:

1	0	0	0	0
0.5	0.5	0	0	0
0	0.5	0.5	0	0
0	0	0.5	0.5	0
0	0	0	0.5	0.5

To recap, each row here represents the attention the model will pay to different token positions (columns) when making its prediction for that position—that is, predicting the next token. To make the prediction for the first token (predicting the second token), we can only pay attention to the first token. To make the prediction for the second token (predicting the third token), we split attention between the first and second tokens, and so on.

But what about `v`? The final step of attention is multiplying the matrix from above with v:

`softmax(q @ k.T / np.sqrt(q.shape[-1]) + mask) @ v`, note the `@ v`. So what's `v`?

Well, recall that passing the embedding matrix from before...

1	0	0	0	0	1	0	0	position 0, token `a`
0	1	0	0	0	1	0	0	position 1, token `a`
0	0	1	0	0	0	1	0	position 2, token `b`
0	0	0	1	0	1	0	0	position 3, token `a`
0	0	0	0	1	1	0	0	position 4, token `a`

...through `linear` with `c_attn`, we get the following `qkv` matrix:

q	q	q	q	q	q	q	q	k	k	k	k	k	k	k	k	v	v	v	v	v	v	v	v
1024	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1024	1024	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	1024	1024	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	-1
0	0	1024	1024	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1024	1024	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1

Looking at the `v` part, we can see that it only ever has one element (column 7) ever set, and that element is `1` when the row is an `a` token, and `-1` when the row is a `b` token. That means all that's happening in `v` is that the one-hot token encoding (`a = [1, 0], b = [0, 1]`) is being turned into a `1`/`-1` encoding!

That might sound pretty useless, but recall that our task is to predict `aabaab`, or in other words:

- if previous tokens are (a, a) => predict b
- if previous tokens are (a, b) => predict a

- if previous tokens are (b, a) => predict a
- if previous tokens are (b, b) => error, out of domain

Since we can safely ignore the (b, b) case as out of domain, this means we only want to predict a `b` token *if the tokens we're attending to are identical*! Since matrix multiplications involve sums, this means we can take advantage of additive cancellation, or in other words: `0.5 + 0.5 = 1`, and `0.5 + (-0.5) = 0`.

By encoding `a` as `1` and `b` as `-1`, this simple equation does exactly what we want. When the token prediction should be `a`, this equation equals `0`, and when the token prediction should be `b`, this equation equals `1`:

- a, b → 0.5 * 1 + 0.5 * (-1) = 0
- b, a → 0.5 * (-1) + 0.5 * 1 = 0
- a, a → 0.5 * 1 + 0.5 * 1 = 1

If we take our softmax result matrix from before and multiply it with the split-off v matrix from before, performing that exact calculation for each row, we get the following attention result for the input sequence `aabaa`:

1	0	0	0	0
0.5	0.5	0	0	0
0	0.5	0.5	0	0
0	0	0.5	0.5	0
0	0	0	0.5	0.5

@

0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	-1
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1

=

0	0	0	0	0	0	0	1	(1 * 1)
0	0	0	0	0	0	0	1	(0.5*1 + 0.5*1)
0	0	0	0	0	0	0	0	(0.5*1 + 0.5*(-1))
0	0	0	0	0	0	0	0	(0.5*(-1) + 0.5*1)
0	0	0	0	0	0	0	1	(0.5*1 + 0.5*1)

The first row has a spurious `b` prediction because it doesn't have enough data (with only a single `a` token to go on, the result could be either `a` or `b`). But the other two `b` predictions are right on: the second row predicts the next token should be `b`, which is correct, and the final row predicts the token following the sequence should also be `b`, which is also correct.

So to summarize, what the `c_attn` weights are doing is:

- Mapping the position embeddings into an "attention window" in `q`
- Extracting the position embeddings into `k`
- Transforming the token embeddings into a `1/-1` token encoding in `v`
- When `q` and `k` are combined in `softmax(q @ k.T / ... + mask)`, we get a `seq_len x seq_len` matrix that
  - in the first row, only attends to the first token
  - in the other rows result, attends equally to the two most recent tokens
- Finally, with `softmax(...) @ v`, we take advantage of additive cancellation to get
  - a `0` in position 7 of the row when the model should predict `a`,
  - and a `1` in position 7 of the row when the model should predict `b`

(Another, more biological way to think of this is in terms of repressors and promoters: seeing an a in the attended tokens promotes position 7, and seeing a b represses it.)

That all works, which means our attention head is done!

🔗 Projecting back to embedding space

Next, to finish the transformer block we need to project the attention result back to a regular embedding. Our attention head put its prediction in `embedding[row, 7]` (`1` for `b`, `0` for `a`), but elsewhere we use a one-hot scheme, with a positive value in `embedding[row, 5]` indicating `a` and a positive value in `embedding[row, 6]` indicating `b`.

For reasons, that will become clear in a bit, we don't want this layer to produce a plain 1-hot like `[..., 1, 0, ...]` or `[..., 0, 1, ...]`. Instead, we want to produce a *scaled* 1-hot of `[..., 1024, 0, ...]` or `[..., 0, 1024, ...]`.

To do this, all we have to do is use the bias of `c_proj` to set `embedding[row, 5]` (the one-hot for

token `a`) to `1024` by default, and then yoink in and appropriately scale the attention result's `embedding[row, 7]`:

```
Lg = 1024  # Large

MODEL = {
  "wte": ...,
  "wpe": ...,
  "blocks": [
    {
      "attn": {
        "c_attn": ...,
        "c_proj": {  # weights to project attn result back to embedding space
          "b": [0, 0, 0, 0, 0, Lg, 0, 0],
          "w": np.array([
            [0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, -Lg, Lg, 0],
          ]),
        },
      },
    },
  ],
}
```

In other words, after `c_proj`,

- `embedding[row, 5]` (corresponding to `a`) equals `Lg + (-Lg) * prediction`
- `embedding[row, 6]` (corresponding to `b`) equals `0 + Lg * prediction`

And after running the attention result from before through `c_proj`, we get this matrix, which is exactly what we want—one hot predictions, scaled by 1024!

0	0	0	0	0	0	1024	0
0	0	0	0	0	0	1024	0

0	0	0	0	0	1024	0	0
0	0	0	0	0	1024	0	0
0	0	0	0	0	0	1024	0

Now we're done with `c_proj`, and can project the transformer block result back to vocabulary space to make a prediction!

🔗 Projecting back to vocabulary space and extracting probabilities

We start out with the result of running the transformer block:

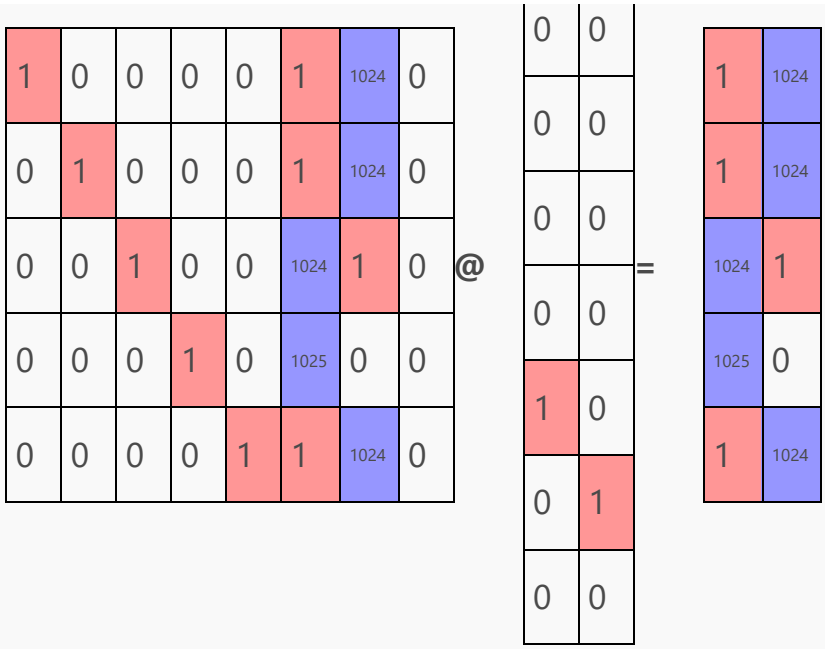
1	0	0	0	0	1	1024	0
0	1	0	0	0	1	1024	0
0	0	1	0	0	1024	1	0
0	0	0	1	0	1025	0	0
0	0	0	0	1	1	1024	0

This is the original embedding added to the `c_prog` result from above. The original embedding is added because of what's called a residual connection: in `transformer_block`, we do `x = x + causal_self_attention(x, ...)` (note the `x +`) instead of simply doing `x = causal_self_attention(x, ...)`.

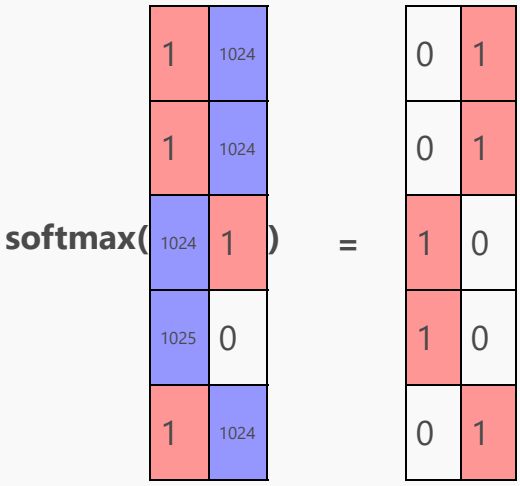
Residual connections can help deep networks maintain information flow through lots of layers, but in our case it just gets in the way. *This* is why the output of `c_proj` was scaled by`1024`: to drown out the unneeded residual signal.

The next step is to multiply the above matrix by the transposed token embedding weights (`wte`) we defined at the start to get the final logits:

0	0



The red squares in the logits show where the model has a slight bias to repeat a token, because of the residual connection, but the opposed `1024` drowns that out, so the final predictions after `softmax` are all 100% one way or the other:



Or in other words, when given the context sequence `aabaa`, the model predicts:

- The token following `a` is `b` (acceptable, could be either)
- The token following `aa` is `b` (correct!)
- The token following `aab` is `a` (correct!)
- The token following `aaba` is `a` (correct!)
- The token following `aabaa` is `b` (correct!)

Of course, for inference all we care about is that final prediction row: `b` follows `aabaa`. The other predictions are only useful for training the model.

With the finished model weights (below), we can write a small `complete` function and show that our handmade model always generates reasonable completions:

```
def complete(s, max_new_tokens=10):
  tokens = tokenize(s)
  while len(tokens) < len(s) + max_new_tokens:
    logits = gpt(np.array(tokens[-5:]), **MODEL)
    probs = softmax(logits)
    pred = np.argmax(probs[-1]) # greedy sample, but temperature sampling would give the s
    tokens.append(pred)
  return s + " :: " + "".join(untok(t) for t in tokens[len(s):])

print(complete("a")) # a :: baabaabaab
print(complete("ba")) # ba :: abaabaabaa
print(complete("abaab")) # abaab :: aabaabaaba
```

It can even recover from out of domain inputs!

```
print(complete("ababa")) # ababa :: abaabaabaa
print(complete("bbbbb")) # bbbbb :: aabaabaaba
```

If we write a small accuracy testing loop, the handmade model is 100% accurate (as long as it's given an unambiguous context):

```
test = "aab" * 10
total, correct = 0, 0
for i in range(2, len(test) - 1):
  ctx = test[:i]
  expected = test[i]
  total += 1
  if untok(predict(ctx)) == expected:
    correct += 1
print(f"ACCURACY: {correct / total * 100}% ({correct} / {total})")
# ACCURACY: 100.0% (27 / 27)
```

🔗 Conclusion

Thanks for reading! Hopefully you have a more intuitive understanding of transformers and attention now, and maybe even feel inspired to make your own model by hand as well!

If you enjoyed this post, you may also enjoy:

- My other blog posts, such as GPT-3 will ignore tools when it disagrees with them, Does GPT-4 think better in Javascript? and I'm worried about adversarial training data
- My other projects
- My Twitter, where I post about new blog posts, smaller AI-related thoughts (e.g. 1, 2), and other things.

If you have thoughts about this post, please feel free to get in touch! I love hearing from people who read my posts.

🔗 Thanks

Thanks to the following people for reviewing drafts of this post:

- MF FOOM—a great account to follow if you're interested in LLMs!
- Corinne
- Susan Vogel

🔗 Completed code

```python
# Model ops from https://github.com/jaymody/picoGPT/blob/main/gpt2.py (MIT license)

import numpy as np


def softmax(x):
    exp_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)

# [m, in], [in, out], [out] -> [m, out]
def linear(x, w, b):
    return x @ w + b

# [n_q, d_k], [n_k, d_k], [n_k, d_v], [n_q, n_k] -> [n_q, d_v]
def attention(q, k, v, mask):
    return softmax(q @ k.T / np.sqrt(q.shape[-1]) + mask) @ v

# [n_seq, n_embd] -> [n_seq, n_embd]
```

```python
def causal_self_attention(x, c_attn, c_proj):
    # qkv projections
    x = linear(x, **c_attn)  # [n_seq, n_embd] -> [n_seq, 3*n_embd]

    # split into qkv
    q, k, v = np.split(x, 3, axis=-1)  # [n_seq, 3*n_embd] -> 3 of [n_seq, n_embd]

    # causal mask to hide future inputs from being attended to
    causal_mask = (1 - np.tri(x.shape[0], dtype=x.dtype)) * -1e10  # [n_seq, n_seq]

    # perform causal self attention
    x = attention(q, k, v, causal_mask)  # [n_seq, n_embd] -> [n_seq, n_embd]

    # out projection
    x = linear(x, **c_proj)  # [n_seq, n_embd] @ [n_embd, n_embd] = [n_seq, n_embd]

    return x

# [n_seq, n_embd] -> [n_seq, n_embd]
def transformer_block(x, attn):
    x = x + causal_self_attention(x, **attn)
    # NOTE: removed ffn
    return x

# [n_seq] -> [n_seq, n_vocab]
def gpt(inputs, wte, wpe, blocks):
    # token + positional embeddings
    x = wte[inputs] + wpe[range(len(inputs))]  # [n_seq] -> [n_seq, n_embd]

    # forward pass through n_layer transformer blocks
    for block in blocks:
        x = transformer_block(x, **block)  # [n_seq, n_embd] -> [n_seq, n_embd]

    # projection to vocab
    return x @ wte.T  # [n_seq, n_embd] -> [n_seq, n_vocab]


N_CTX = 5
N_VOCAB = 2
N_EMBED = 8

Lg = 1024  # Large

MODEL = {
    # EMBEDDING USAGE
    #  P = Position embeddings (one-hot)
```

```
    #  T = Token embeddings (one-hot, first is `a`, second is `b`)
    #  V = Prediction scratch space
    #
    #       [P, P, P, P, P, T, T, V]
    "wte": np.array(
        # one-hot token embeddings
        [
            [0, 0, 0, 0, 0, 1, 0, 0],  # token `a` (id 0)
            [0, 0, 0, 0, 0, 0, 1, 0],  # token `b` (id 1)
        ]
    ),
    "wpe": np.array(
        # one-hot position embeddings
        [
            [1, 0, 0, 0, 0, 0, 0, 0],  # position 0
            [0, 1, 0, 0, 0, 0, 0, 0],  # position 1
            [0, 0, 1, 0, 0, 0, 0, 0],  # position 2
            [0, 0, 0, 1, 0, 0, 0, 0],  # position 3
            [0, 0, 0, 0, 1, 0, 0, 0],  # position 4
        ]
    ),
    "blocks": [
        {
            "attn": {
                "c_attn": {  # generates qkv matrix
                    "b": np.zeros(N_EMBED * 3),
                    "w": np.array(
                        # this is where the magic happens
                        # fmt: off
                        [
                          [Lg, 0., 0., 0., 0., 0., 0., 0.,  # q
                            1., 0., 0., 0., 0., 0., 0., 0.,  # k
                              0., 0., 0., 0., 0., 0., 0., 0.], # v
                          [Lg, Lg, 0., 0., 0., 0., 0., 0.,  # q
                            0., 1., 0., 0., 0., 0., 0., 0.,  # k
                              0., 0., 0., 0., 0., 0., 0., 0.], # v
                          [0., Lg, Lg, 0., 0., 0., 0., 0.,  # q
                            0., 0., 1., 0., 0., 0., 0., 0.,  # k
                              0., 0., 0., 0., 0., 0., 0., 0.], # v
                          [0., 0., Lg, Lg, 0., 0., 0., 0.,  # q
                            0., 0., 0., 1., 0., 0., 0., 0.,  # k
                              0., 0., 0., 0., 0., 0., 0., 0.], # v
                          [0., 0., 0., Lg, Lg, 0., 0., 0.,  # q
                            0., 0., 0., 0., 1., 0., 0., 0.,  # k
                              0., 0., 0., 0., 0., 0., 0., 0.], # v
                          [0., 0., 0., 0., 0., 0., 0., 0.,  # q
```

```
                        0., 0., 0., 0., 0., 0., 0., 0.,  # k
                          0., 0., 0., 0., 0., 0., 0., 1.], # v
                      [0., 0., 0., 0., 0., 0., 0., 0.,  # q
                        0., 0., 0., 0., 0., 0., 0., 0.,  # k
                          0., 0., 0., 0., 0., 0., 0., -1], # v
                      [0., 0., 0., 0., 0., 0., 0., 0.,  # q
                        0., 0., 0., 0., 0., 0., 0., 0.,  # k
                          0., 0., 0., 0., 0., 0., 0., 0.], # v
                  ]
                  # fmt: on
              ),
          },
          "c_proj": {  # weights to project attn result back to embedding space
              "b": [0, 0, 0, 0, 0, Lg, 0, 0],
              "w": np.array(
                  [
                      [0, 0, 0, 0, 0, 0, 0, 0],
                      [0, 0, 0, 0, 0, 0, 0, 0],
                      [0, 0, 0, 0, 0, 0, 0, 0],
                      [0, 0, 0, 0, 0, 0, 0, 0],
                      [0, 0, 0, 0, 0, 0, 0, 0],
                      [0, 0, 0, 0, 0, 0, 0, 0],
                      [0, 0, 0, 0, 0, 0, 0, 0],
                      [0, 0, 0, 0, 0, -Lg, Lg, 0],
                  ]
              ),
          },
      },
  }
  ],
}

CHARS = ["a", "b"]
def tokenize(s): return [CHARS.index(c) for c in s]
def untok(tok): return CHARS[tok]

def predict(s):
  tokens = tokenize(s)[-5:]
  logits = gpt(np.array(tokens), **MODEL)
  probs = softmax(logits)

  for i, tok in enumerate(tokens):
      pred = np.argmax(probs[i])
      print(
          f"{untok(tok)} ({tok}): next={untok(pred)} ({pred}) probs={probs[i]} logits={l
      )
```

```
    return np.argmax(probs[-1])

def complete(s, max_new_tokens=10):
    tokens = tokenize(s)
    while len(tokens) < len(s) + max_new_tokens:
        logits = gpt(np.array(tokens[-5:]), **MODEL)
        probs = softmax(logits)
        pred = np.argmax(probs[-1])
        tokens.append(pred)
    return s + " :: " + "".join(untok(t) for t in tokens[len(s):])

test = "aab" * 10
total, correct = 0, 0
for i in range(2, len(test) - 1):
    ctx = test[:i]
    expected = test[i]
    total += 1
    if untok(predict(ctx)) == expected:
        correct += 1
print(f"ACCURACY: {correct / total * 100}% ({correct} / {total})")
```

🔗 Bonus: Efficiency

For a full 5-token context, our model needs ~4,000 floating point operations to predict a single token, the majority used in the attention calculation. This can be reduced by shrinking the context window, using fused multiply-adds, kv caching, and other techniques, but predicting a single token still needs ~hundreds of machine instructions.

In comparison, handwritten (x64) assembly needs eight:

```
; dl: next token
; rax: context addr
; rcx: context len
.next_token
  mov dl, 'a'
  cmp byte ptr [rax + rcx - 1], 'a'
  jne .done
  cmp rcx, 1
  je .return_b
  cmp byte ptr [rax + rcx - 2], 'a'
  jne .done
.return_b:
```

```
  mov dl, 'b'
.done:
```

Could we somehow train language models that are 1000x more efficient—that's to say, as relatively efficient to current models at generating natural language as this assembly is at generating `(aab)*`? If you figure out how, email me. I'll send the first person $10 ;-)

1

The specific changes I made, if you're interested, are:

- I removed the layer norms, which were annoying to work around. I wanted to pass around nice matrices with lots of 0s and 1s that were easy to reason about, and the layer norms wanted to turn my `1`s and `0`s into `1.73200462`s and `-0.57733487`s:

```
def layer_norm(x, g, b, eps: float = 1e-5):
  mean = np.mean(x, axis=-1, keepdims=True)
  variance = np.var(x, axis=-1, keepdims=True)
  # normalize x to have mean=0 and var=1 over last axis
  x = (x - mean) / np.sqrt(variance + eps)
  return g * x + b  # scale and offset with gamma/beta params
```

(I could have reversed the effect by assigning gamma to revert the `np.sqrt(...)` scaling, and beta to revert the `(x - mean)` offset—but it was easier to just remove the layer norms entirely instead of fixing them up every time I made an unrelated change.)

- I used single-headed attention instead of multi-headed attention, since I didn't need multiple heads.
- I removed the `mlp` feed-forward layer in the transformer block, since I didn't need it.
  (Though I could have just set it to the identity matrix instead.)

2

Unlike some other transformer architectures, GPT-2 uses exclusively learned position embeddings, so there aren't any sines or RoPEs here.

3

Actually, there's no need for `q` or `k` to have these roles—in the original version of this post, I had them flipped, and `q` was the extracted position embeddings and `k` was the query.

However, the way it is in the post now is more common and better fits the matrix names—not that GPT-2 has ever heard those names ;-)

4

In reality, instead of `-∞`, real code uses a number like `-1e10` to avoid issues with `NaN`s, but the practical effect is the same.

Previous entry: Writing a C compiler in 500 lines of Python