Transformers

Inside Out - Parts 1 & 2

Sam Joseph, Ph.D. ~~November 7th, 2023,~~ ~~March 19th 2024,~~ June 6th 2025

Remember to Video the Session

Two cameras

Don’t Forget!

Who am I?

Sam Joseph

- PhD in Machine Learning (Neural Nets)
- Taught Computer Science in Universities, MOOCs and Bootcamps
- Lead AI Engineer (QualisFlow)
- Author of “How to Interact with Humans after Spending a lot of Time with Computers”
- Techie by day, Standup Comedian by night

LinkedIn

Amazon

Instagram

Disclaimer

It’s all about the journey …

- To visualise/understand how a transformer works
- Still not quite finished that journey
- Is there other better work? I haven’t reviewed all the research …
- But it’s still my journey
- Any errors due to me and how I interact with the AIs

In this Tutorial …

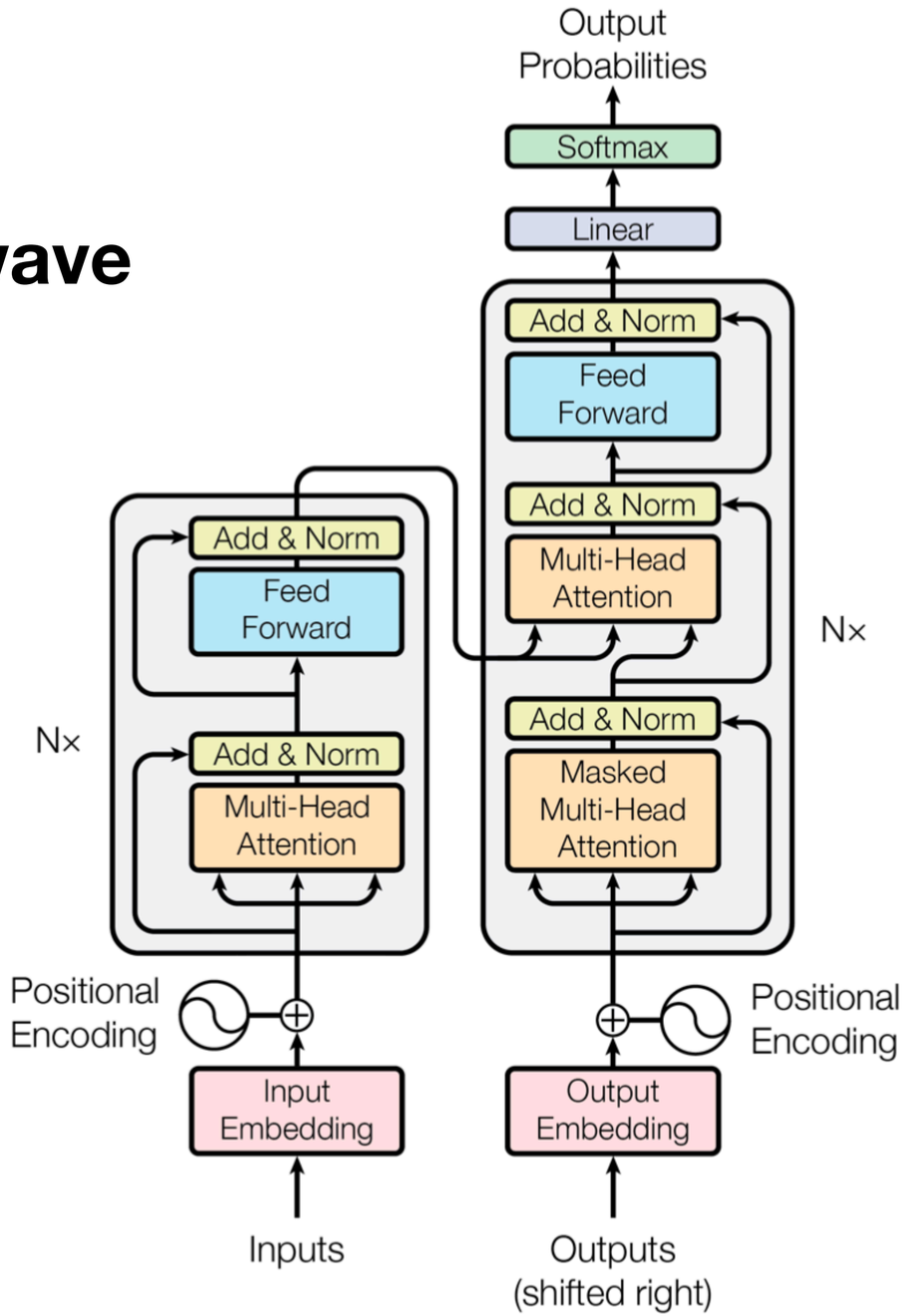
Objectives?

- Assuming you know some Python
- Get you running a GPT on your laptop (Andrej Karpathy's nanoGPT)
  - Or at least see it on your partner's machine
- Pair up with someone who has it working (get it and review README now)
- I'll occasionally take us through
  - Transformer architecture
  - Code on the debugger
  - Visualisations representing data going through the transformer
  - My data set and training results

Attention is All You Need

Google Paper that kicked off the LLM wave

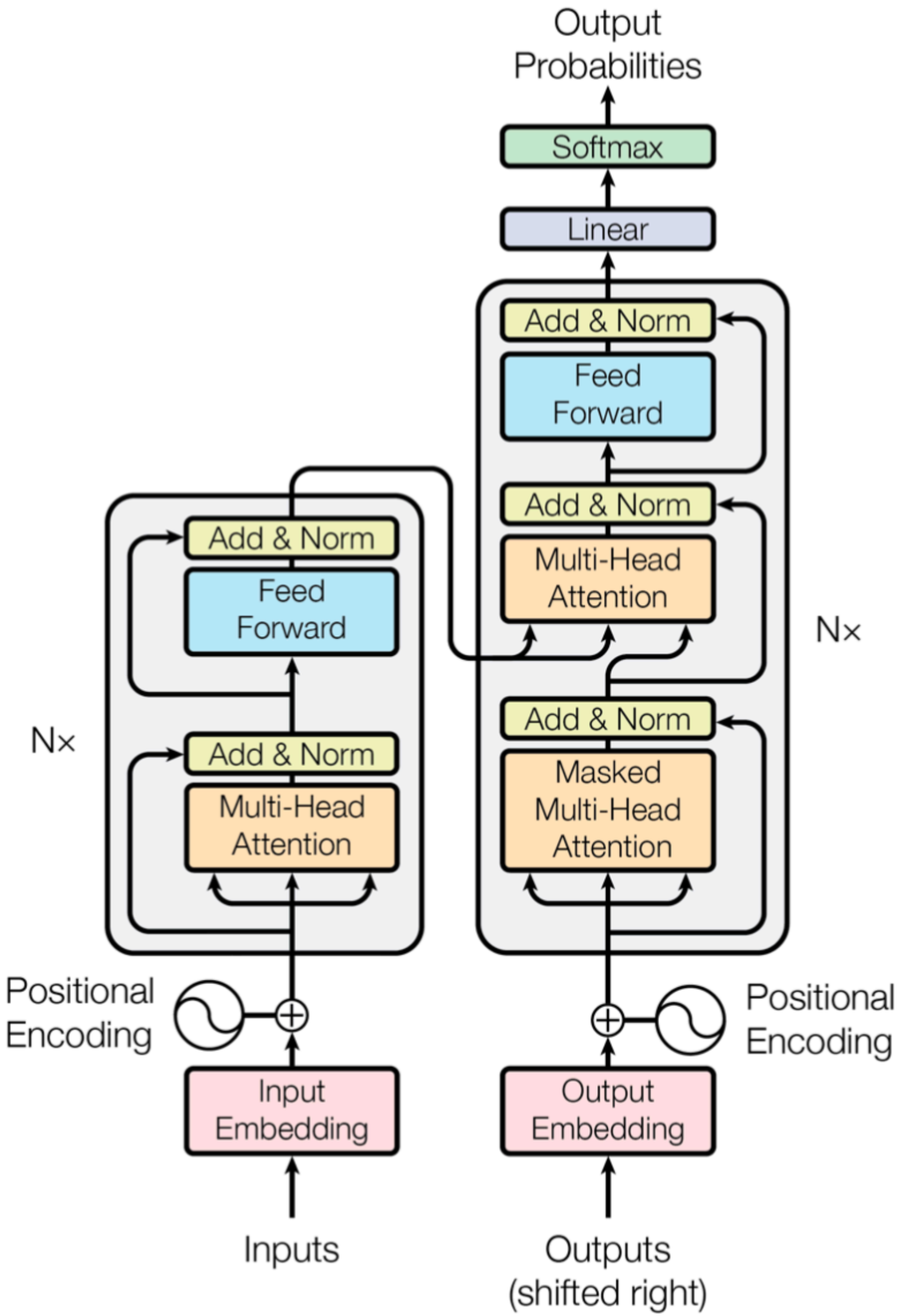
- Vaswani et al. (2017)
- Avoided recurrence and convolutions
- Replaced with parallelizable attention
- Big improvement in English to French translation
- Encoder on left takes in English, converts to internal representation
- Decoder on right converts internal representation into French, word by word



Children of Attention

Encoders and Decoders split up

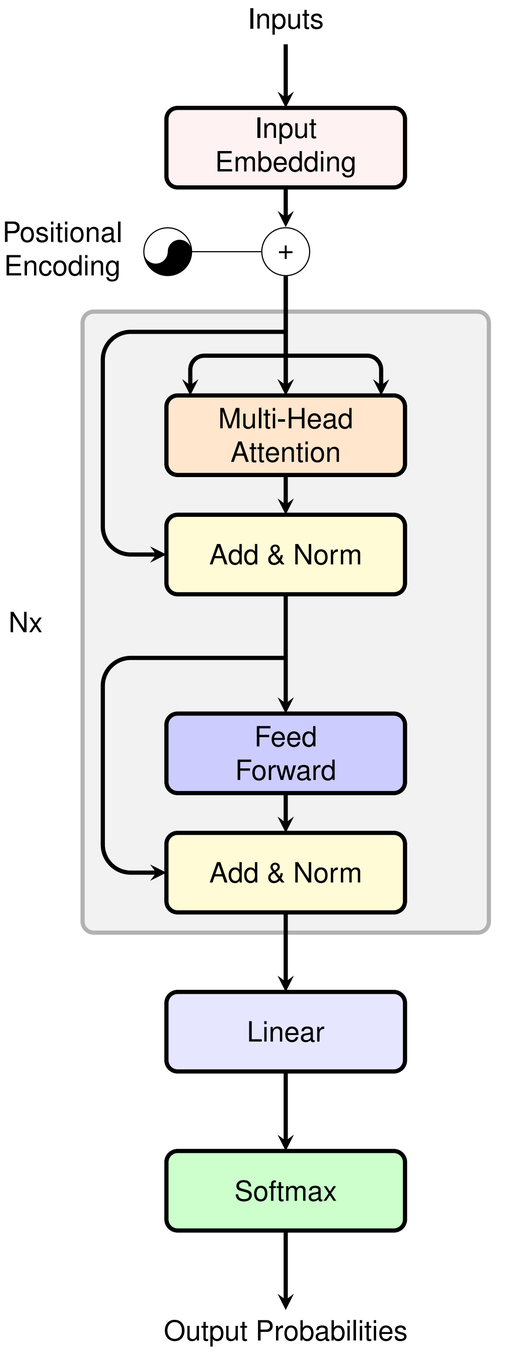
- GPT - “Generative Pre-trained Transformer” architecture
  - Based on decoder (on right)
  - Self-attention (future tokens masked)
  - Good for generating text
- BERT - “Bidirectional Encoder Representations from Transformers”
  - Based on encoder (on left)
  - “masked” Training objective
  - Good for classifying text



Trying to Understand GPT

First Step

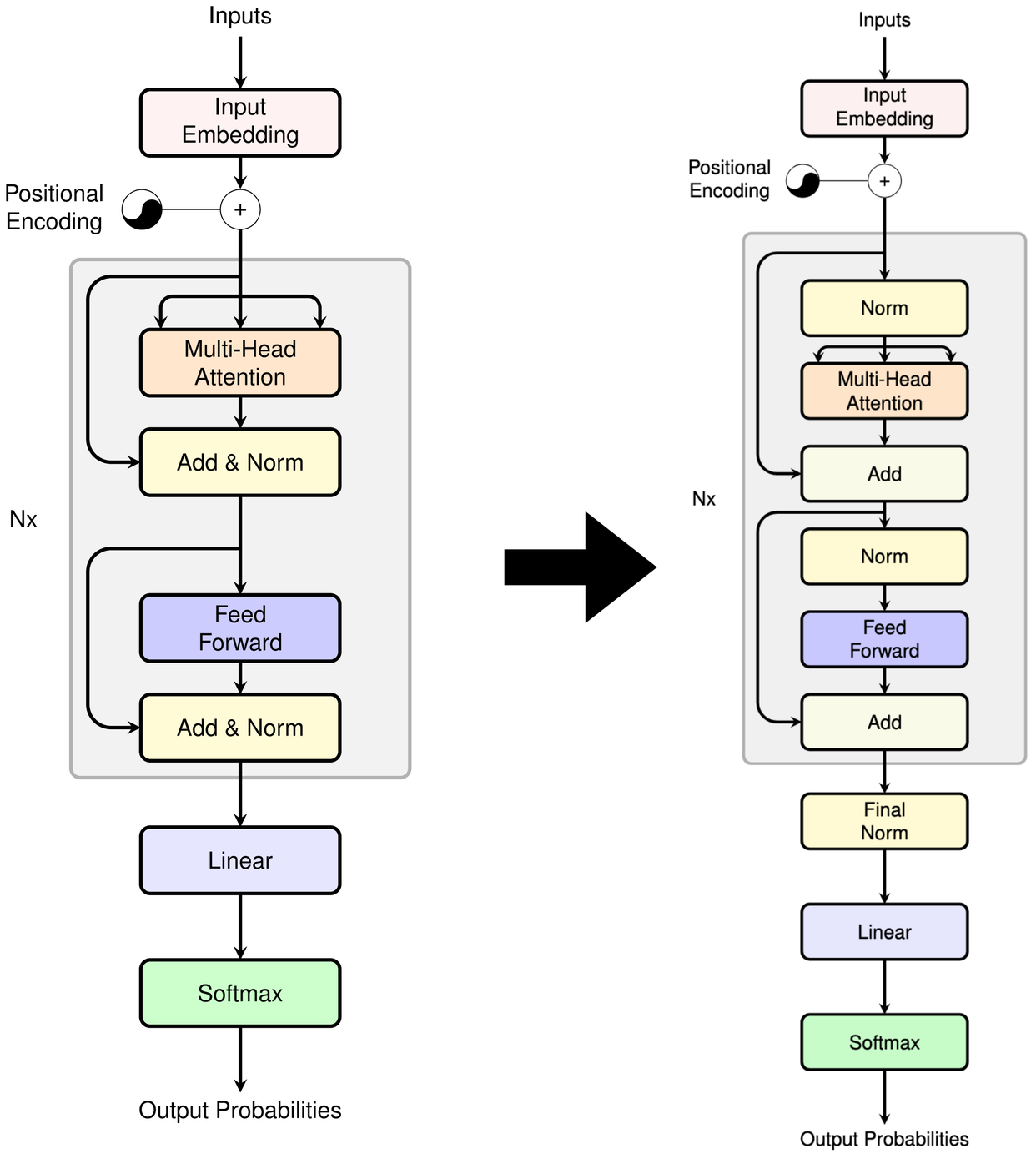
- Doing Coursera's NLP specialization I coded transformers in Trax
- Diagrams flow up
- Code flows down
- Can we fix that?
- Emailed Vaswani et al. for their diagrams
- No response
- Learn tikz?
- I "paired" with GPT4 to create the tikz and the code (pytorch)
- Could I train a simple 1 layer GPT model on jokes in order for me to better understand GPT?
- No
- Fix was moving to multi-layer distilGPT2 architecture - 6 layer, 1 head, 36 embed dimension



GPT2

Slightly different sequencing

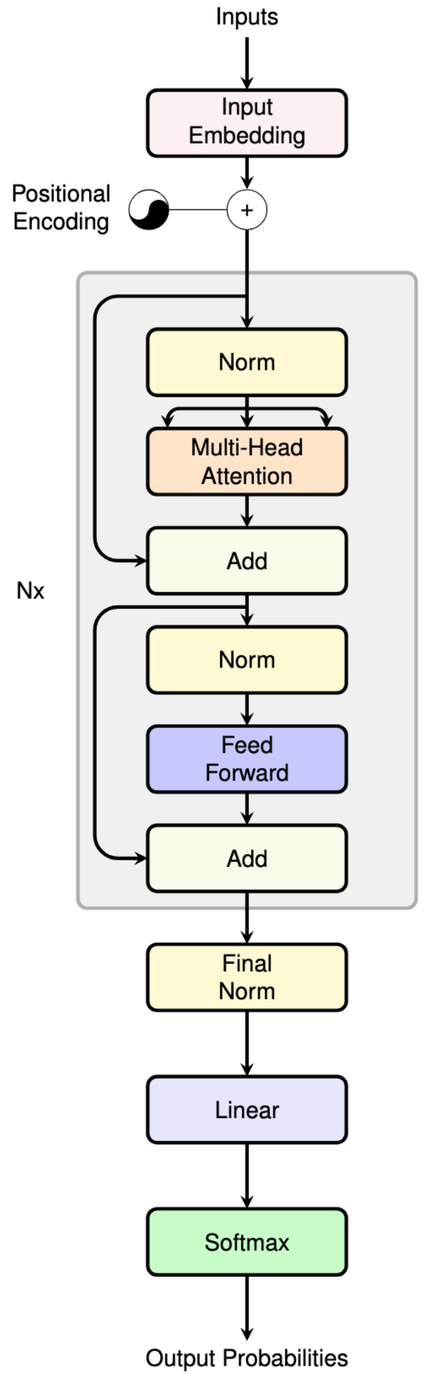
- GPT2 puts normalisation steps before attention and feed forward layers
- PreNorm is widely preferred for training stability, especially in deep transformers, because:
  - Gradients flow better during backpropagation (training)
  - It's less sensitive to initialisation and optimiser settings
  - It avoids certain exploding/vanishing behaviours in very deep networks



Joke Data Set

Knock knock …

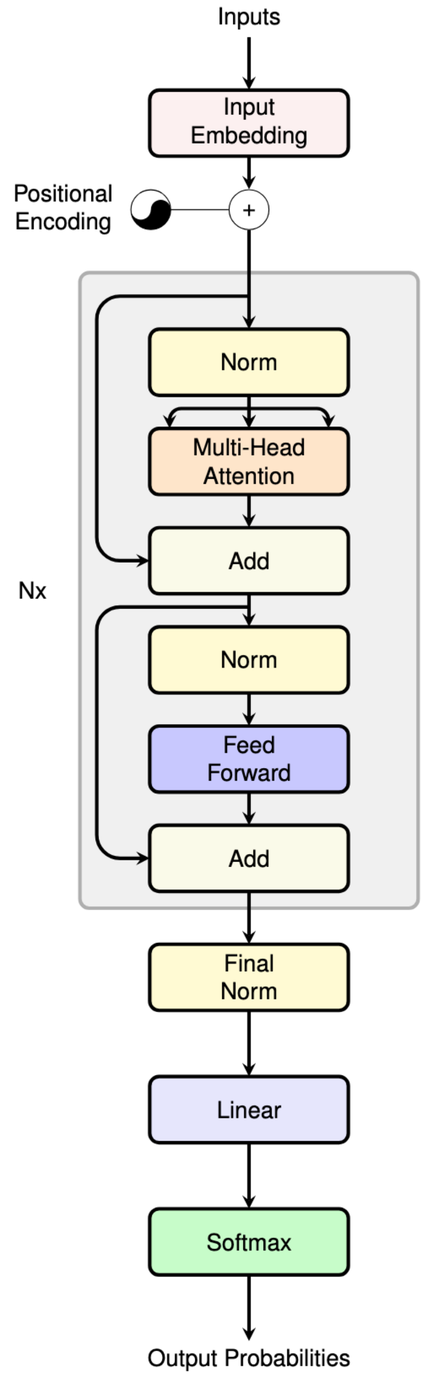
- 35,347 generated “knock knock” jokes
- Taken from a corpus of common sayings
  - e.g. “bob’s your uncle” becomes
  - “Knock knock. Who’s there? Bob. Bob who? Bob’s your uncle”
  - Not saying it’s funny, but it is technically a knock knock joke
- Data set has maximum sequence length of 15
- And fixed vocabulary size 8368



Input Tokenizing

Getting started

- Inputs have to be tokenized
- Sub-word tokenizing is common, but I used single words as tokens to help with visualisation
- “Knock knock whos there”
- [‘knock’, ‘knock’, ‘whos’, …]
- [241, 241, 223, …]
- Added End Of Sentence [EOS] token

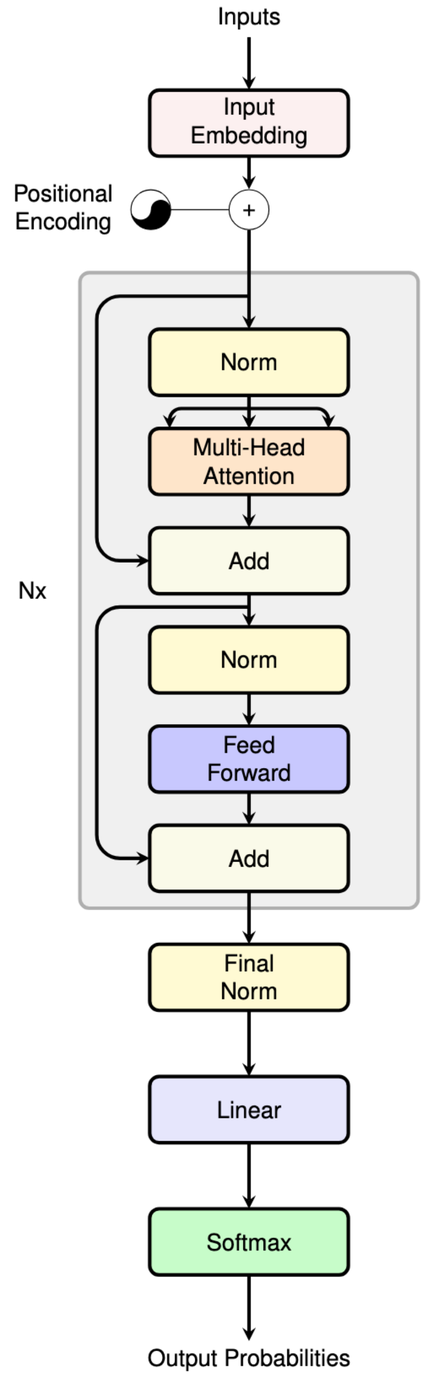


Input Tokenizing

Getting started (prepare.py)

```python
input_file_path = os.path.join(
    os.path.dirname(__file__), "knock-knock-jokes-cleaned.txt"
)
output_dir = os.path.dirname(input_file_path)
train_frac = 0.9

# Load the text
with open(input_file_path, "r", encoding="utf-8") as f:
    data = f.read()

# Split into train and val
n = len(data)
train_data = data[: int(train_frac * n)]
val_data = data[int(train_frac * n) :]
def tokenize_words(text):
    # First, handle <|endoftext|> as a single token
    text = text.replace("<|endoftext|>", " SPECIALENDOFTEXTTOKEN ")
    tokens = re.findall(r"\w+|[^\w\s]", text, re.UNICODE)
    # Replace placeholder with the real token
    return [t if t != "SPECIALENDOFTEXTTOKEN" else "<|endoftext|>" for t in tokens]


train_words = tokenize_words(train_data)
val_words = tokenize_words(val_data)
```

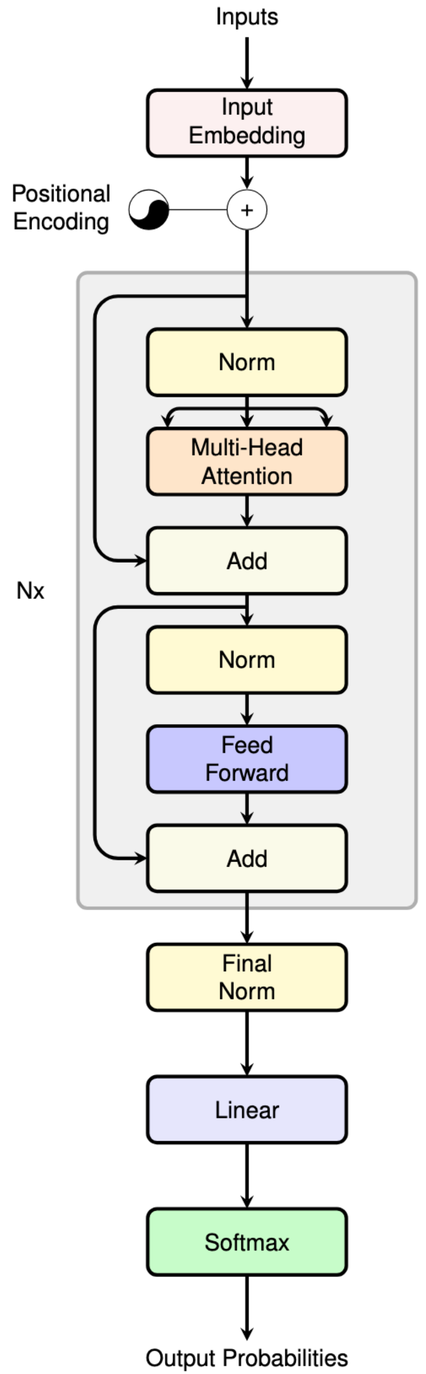
- “Knock knock whos there” → [241, 241, 223, …]



Training

On knock knock jokes

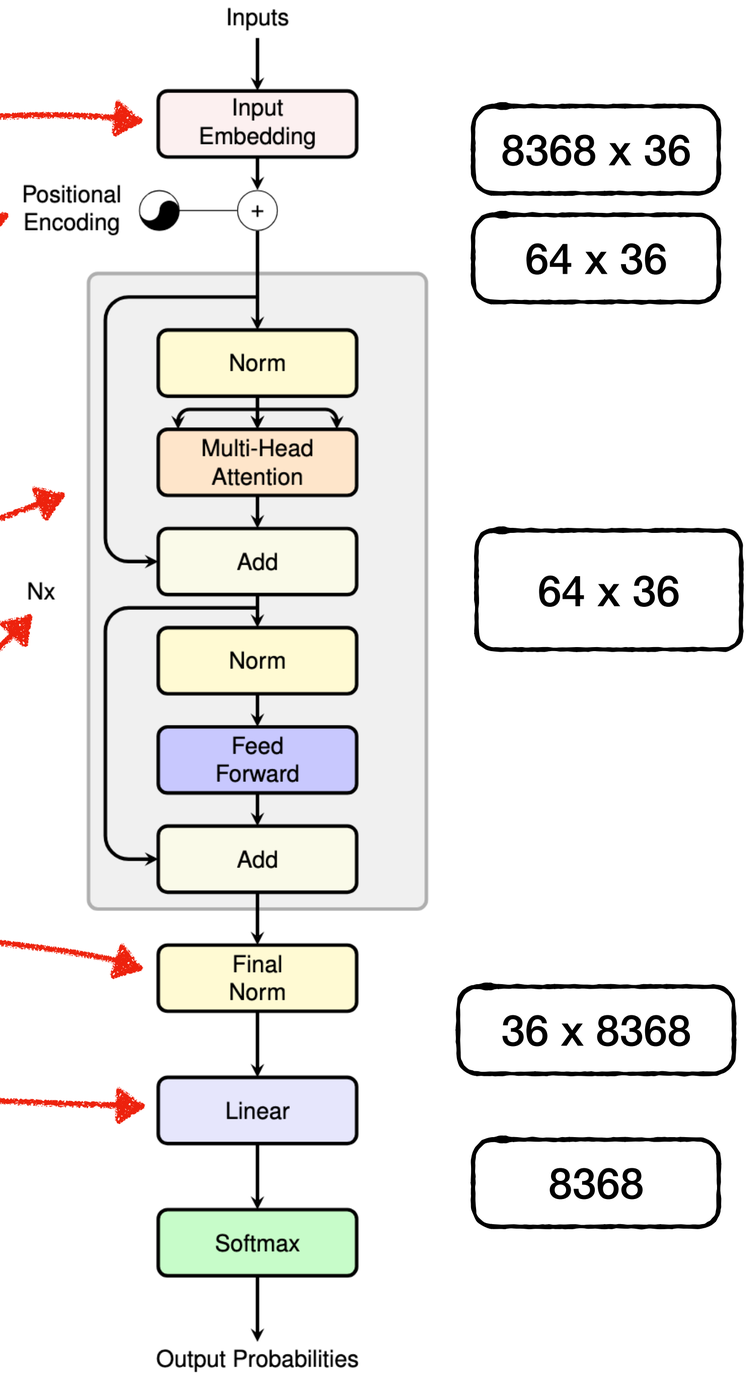
- Divide the jokes into training set and validation set
- Run the training set jokes through in batches
- Update the weights in the network via backpropagation
- Use validation set to assess progress
- First let’s work through transformer architecture



Transformers

Upside Down: n_embed = 36

```python
class GPT(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.vocab_size is not None
        assert config.block_size is not None
        self.config = config

        self.transformer = nn.ModuleDict(dict(
            wte = nn.Embedding(config.vocab_size, config.n_embd),
            wpe = nn.Embedding(config.block_size, config.n_embd),
            drop = nn.Dropout(config.dropout),
            h = nn.ModuleList([Block(config) for _ in range(config.n_layer)]),
            ln_f = LayerNorm(config.n_embd, bias=config.bias),
        ))
        self.lm_head = nn.Linear(config.n_embd, config.vocab_size, bias=False)
```

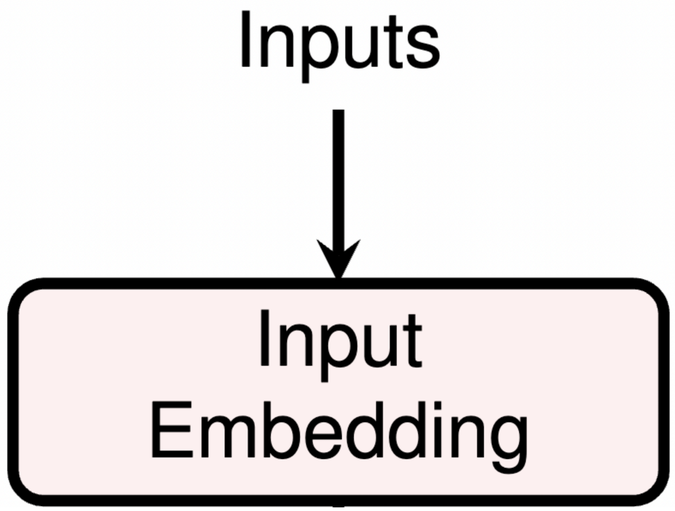


- This code is in root of nanoGPT **model.py**

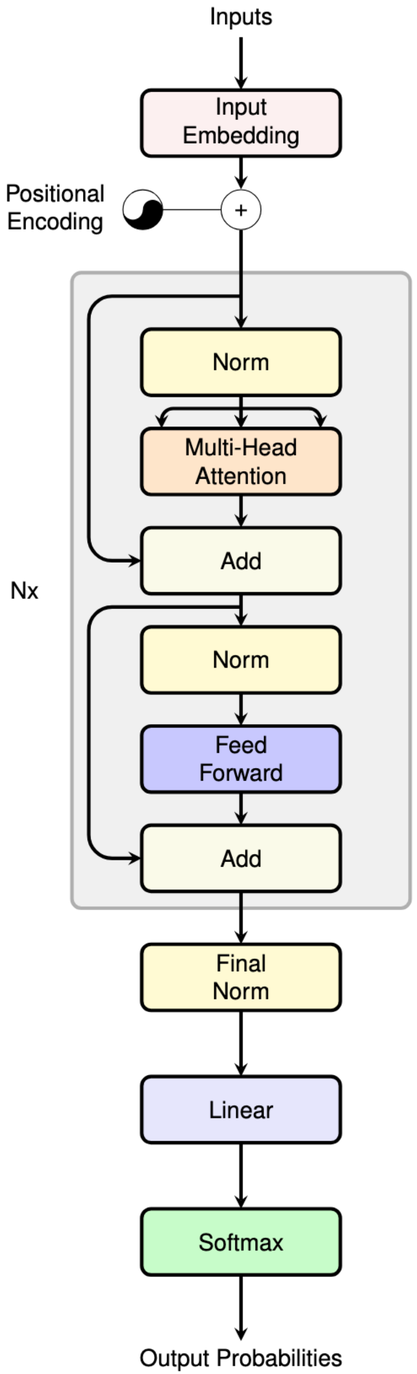
Input Embedding

First layer (8368 x 36)

- Embedding maps from vocabulary to some fixed model size, e.g. 36



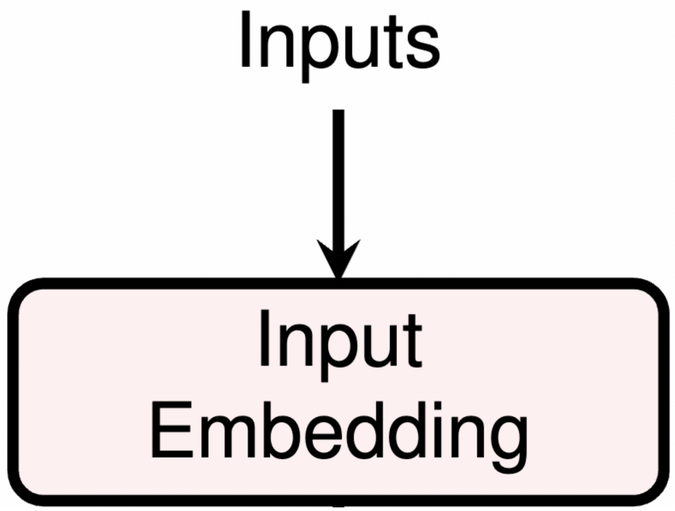
- Bigger black words represent larger weights
- Bigger red words represent larger negative weights
- Show HTML UI



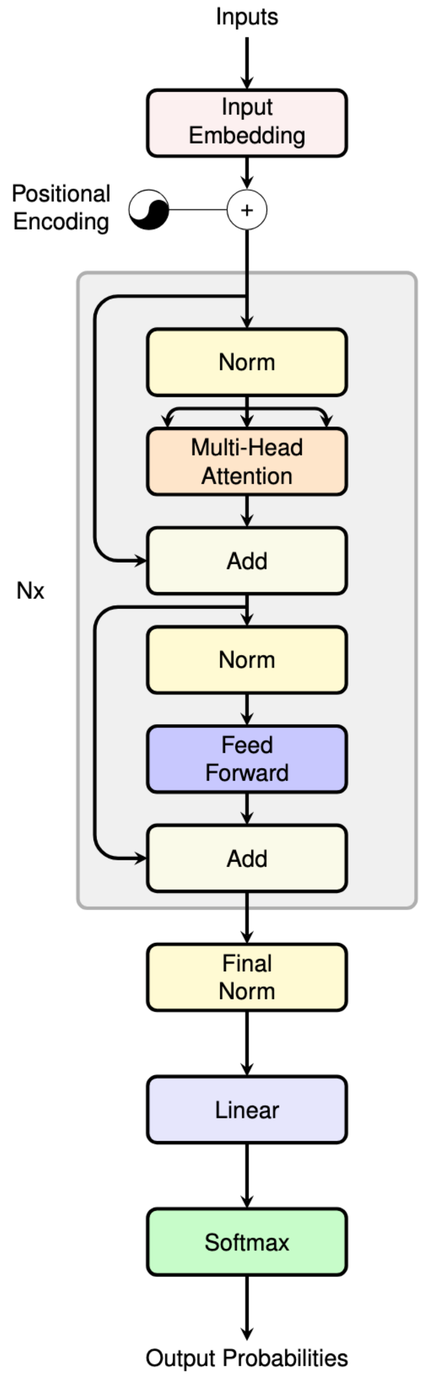
Input Embedding

First layer (8368 x 36)

- Allows us to represent a sentence like so

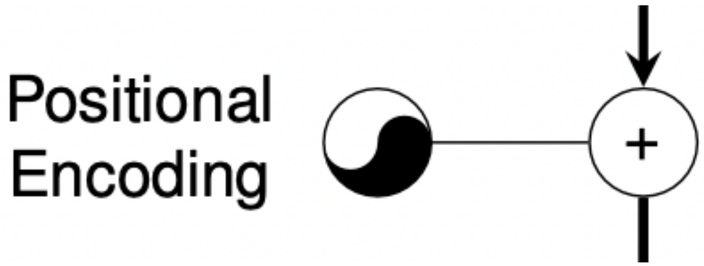


- More opaque means more active
- Inverted colour scheme means negatively active
- Show HTML UI



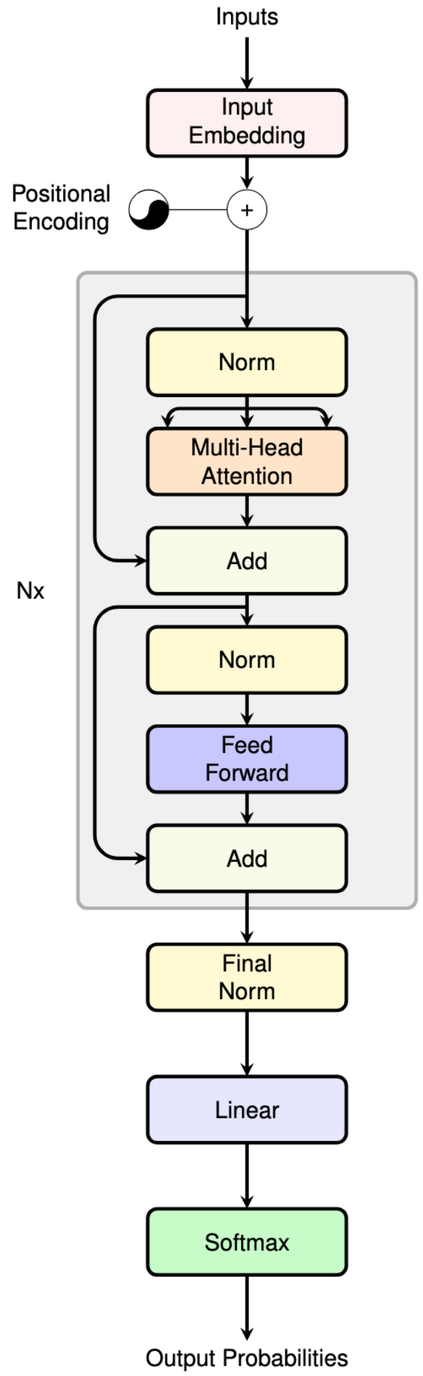
Positional Encoding

Added in (64x36)



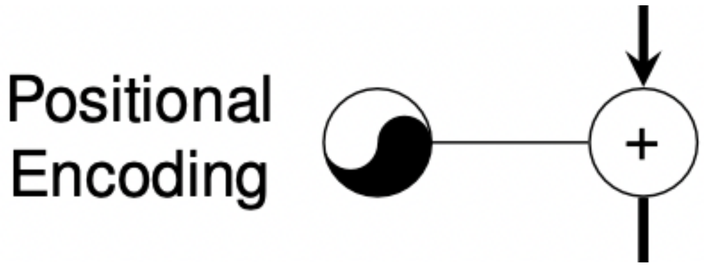
- Input embedding has no sense of positions
- GPT2 learns a positional encoding

- Bigger black numbers represent larger weights
- Bigger red numbers represent larger negative weights
- Show HTML UI



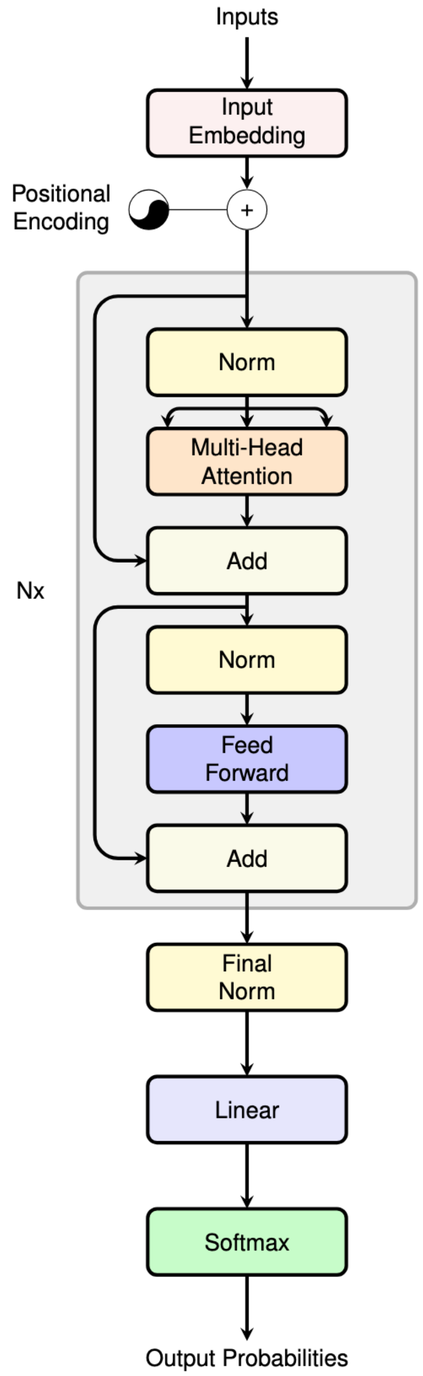
Positional Encoding

Added in (64x36)



- Allows us to represent a probe sentence like so

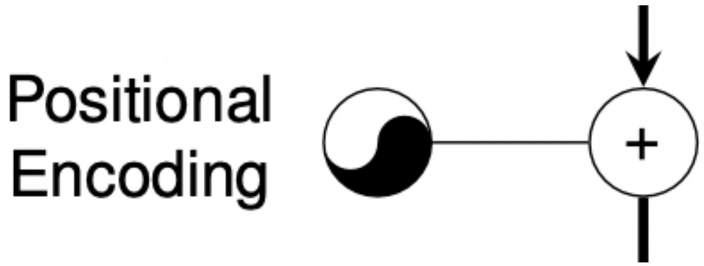
- More opaque means more active
- Inverted colour scheme means negatively active
- Show HTML UI



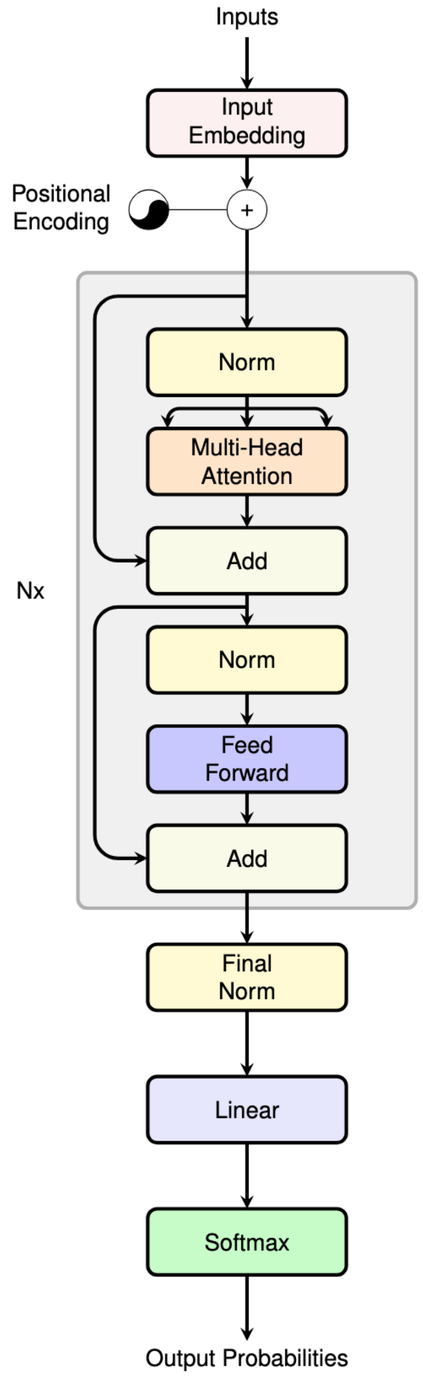
Positional Encoding

Added in (64x36)

- Both embeddings combine like so



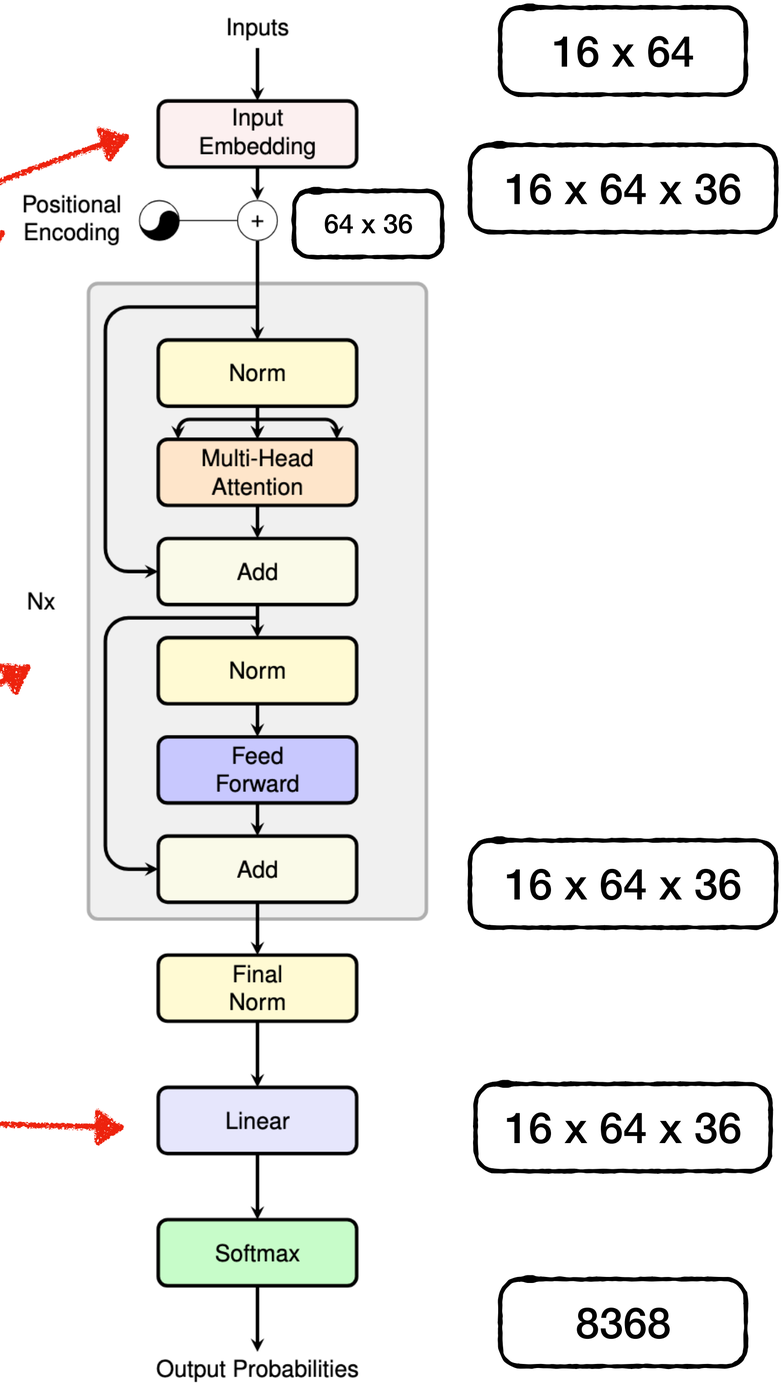
- More opaque means more active
- Inverted colour scheme means negatively active
- Show HTML UI



Forward Pass

Upside Down: batch_size (b) = 16, block_size (t) = 64

```
1.class GPT(nn.Module):

3.    def forward(self, idx, targets=None):
4.        device = idx.device
5.        b, t = idx.size()

7.        pos = torch.arange(0, t, dtype=torch.long, device=device) # shape (t)

9.        # forward the GPT model itself
10.        tok_emb = self.transformer.wte(idx)
11.                  # token embeddings of shape (b, t, n_embd)
12.        pos_emb = self.transformer.wpe(pos)
13.                  # position embeddings of shape (t, n_embd)
14.        x = self.transformer.drop(tok_emb + pos_emb)
15.        for block in self.transformer.h:
16.            x = block(x)
17.        x = self.transformer.ln_f(x)
```

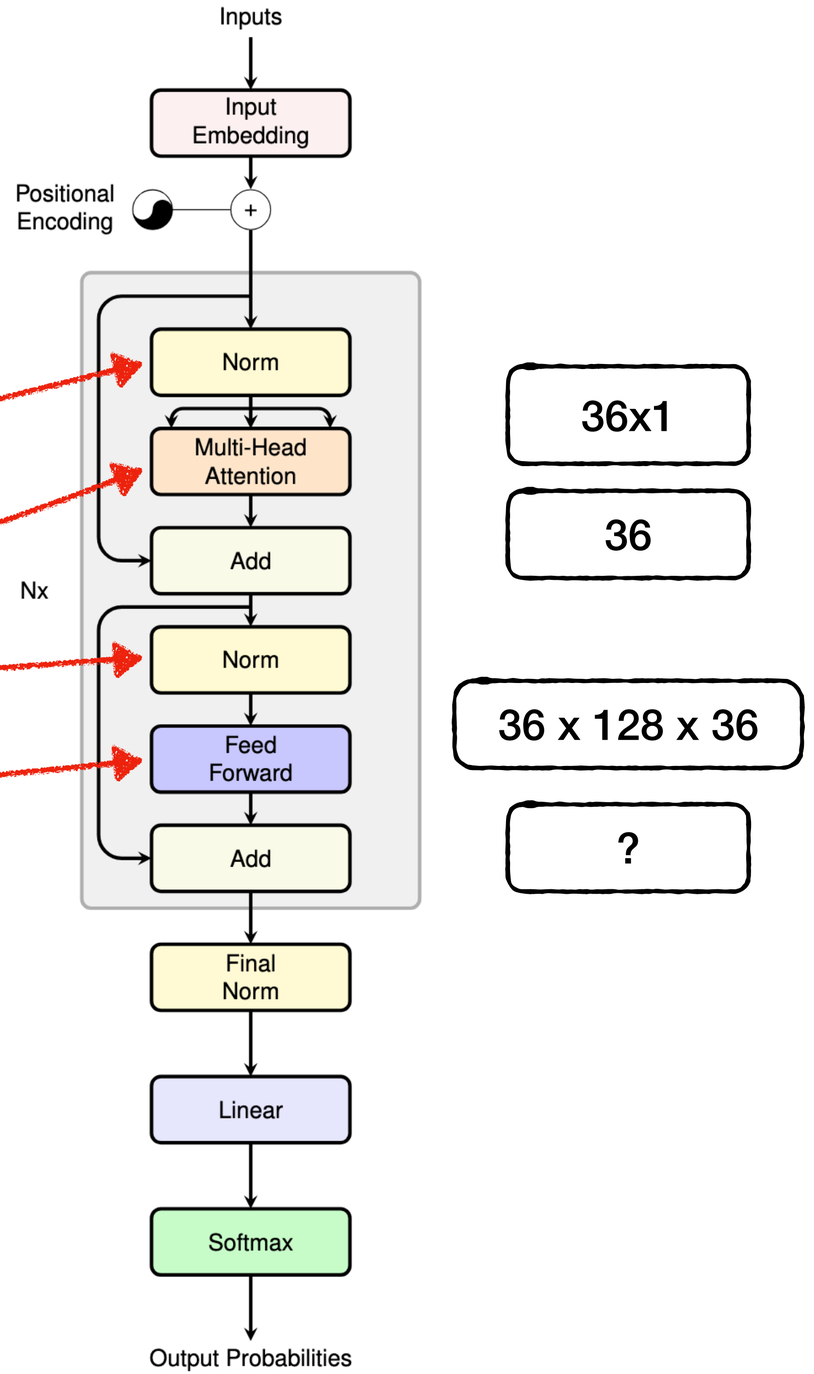


- This code is in root of nanoGPT **model.py**

Transformers (Block)

Upside Down: d_model = 36, nheads = 1

```
2. class Block(nn.Module):

4.     def __init__(self, config):
5.         super().__init__()
6.         self.ln_1 = LayerNorm(config.n_embd, bias=config.bias)
7.         self.attn = CausalSelfAttention(config)
8.         self.ln_2 = LayerNorm(config.n_embd, bias=config.bias)
9.         self.mlp = MLP(config)
```

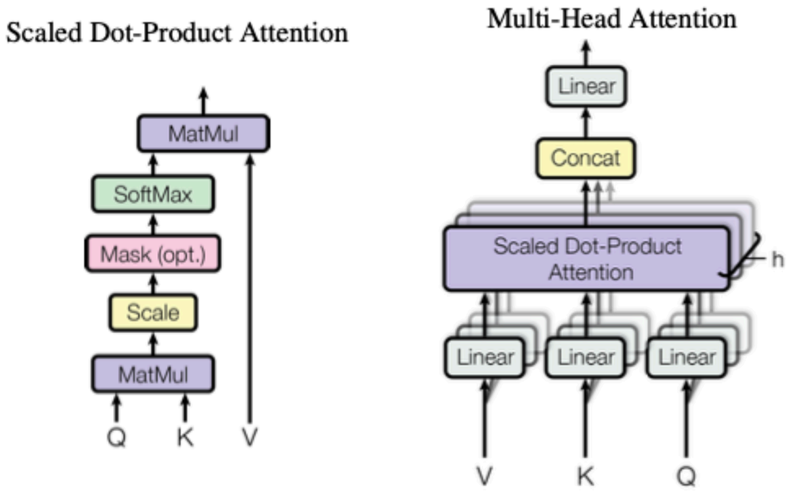


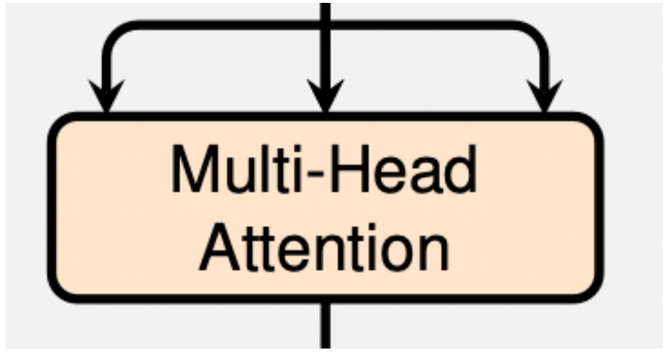
- This code is in root of nanoGPT **model.py**

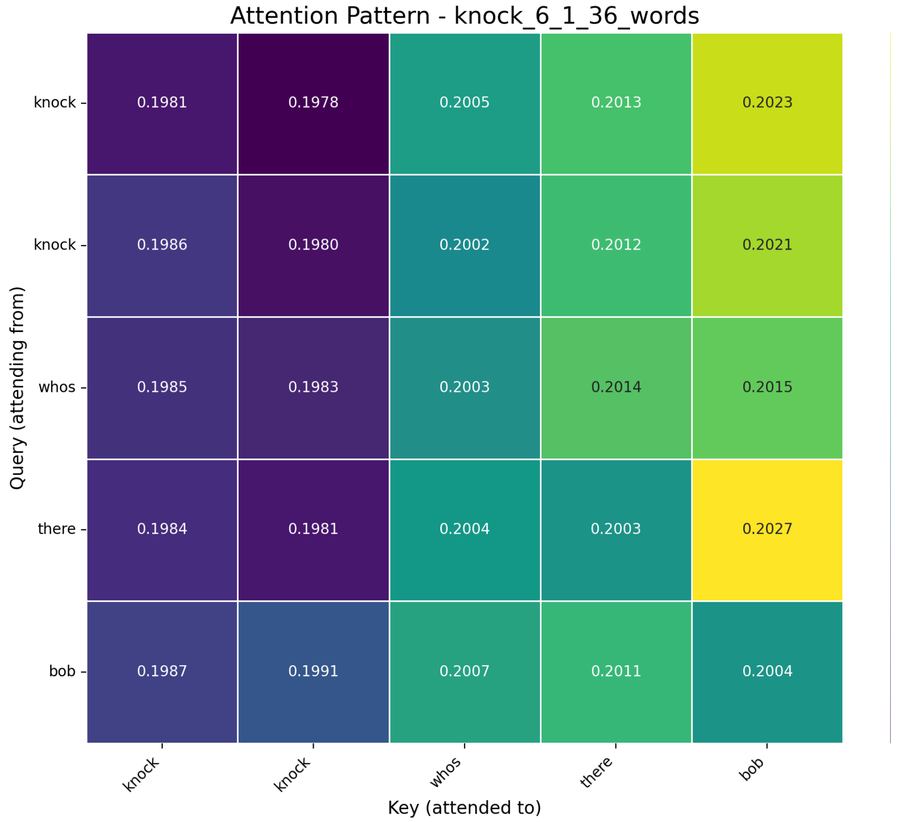
Transformer Block

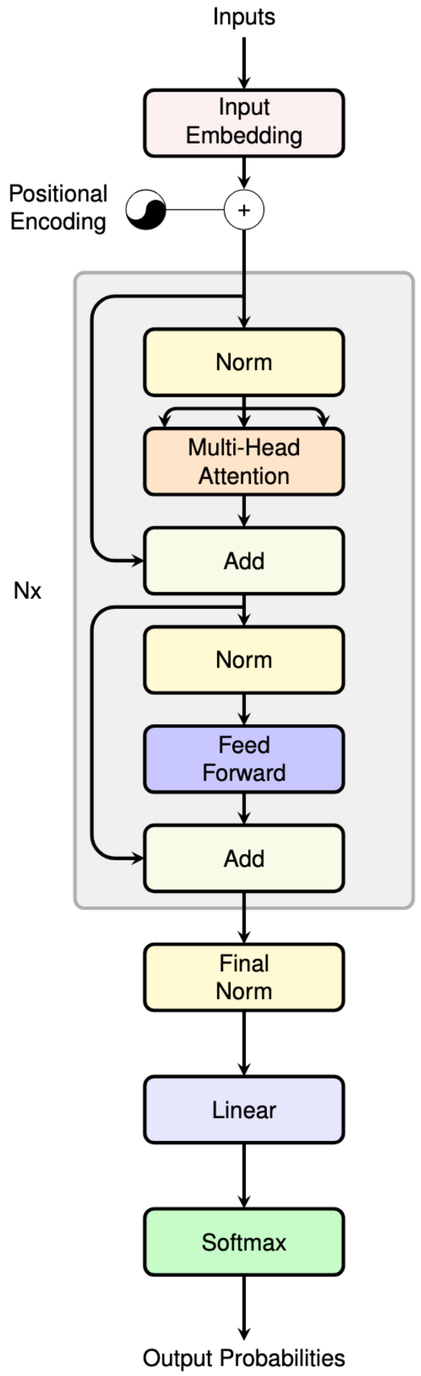
Multi-Head Attention

- Combined input and positional embedding is recombined with itself 3 times over
- Attention weights “activations” indicate tokens influence on each other





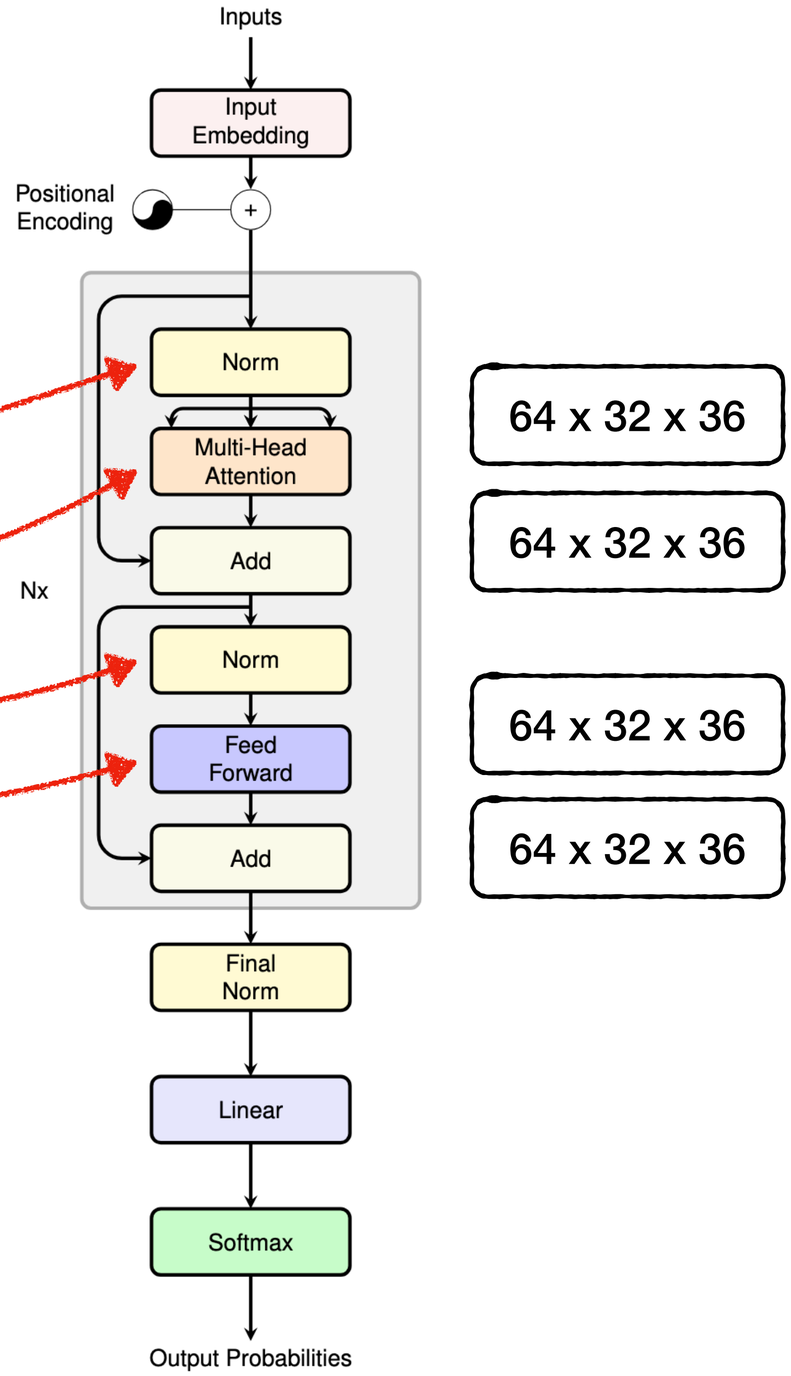




Forward Pass (Block)

Upside Down: d_model = 36, batch_size = 32

```
1.class Block(nn.Module):
2.    def forward(self, x):
3.        x = x + self.attn(self.ln_1(x))
4.        x = x + self.mlp(self.ln_2(x))
5.        return x
```

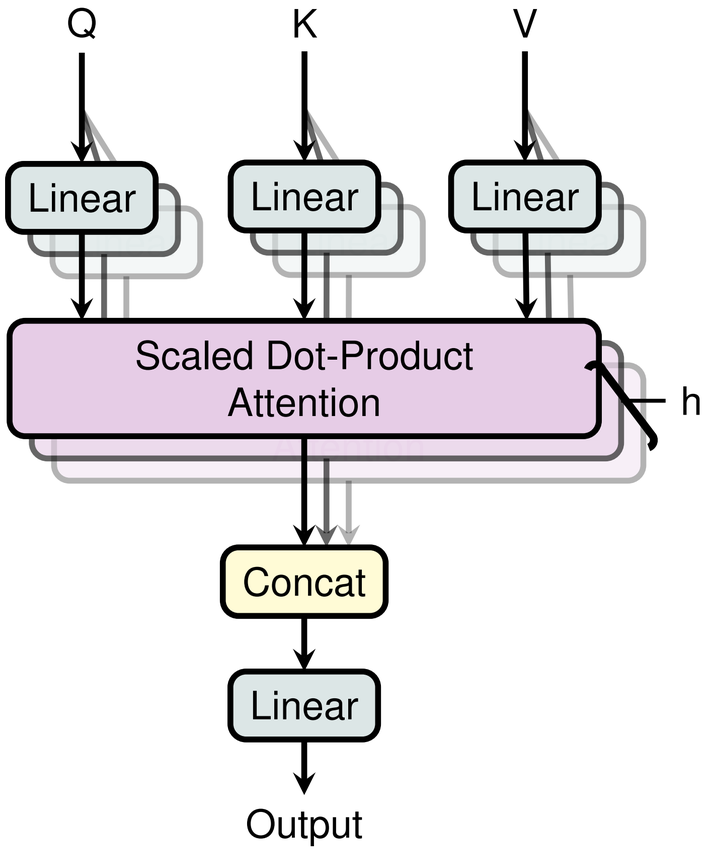


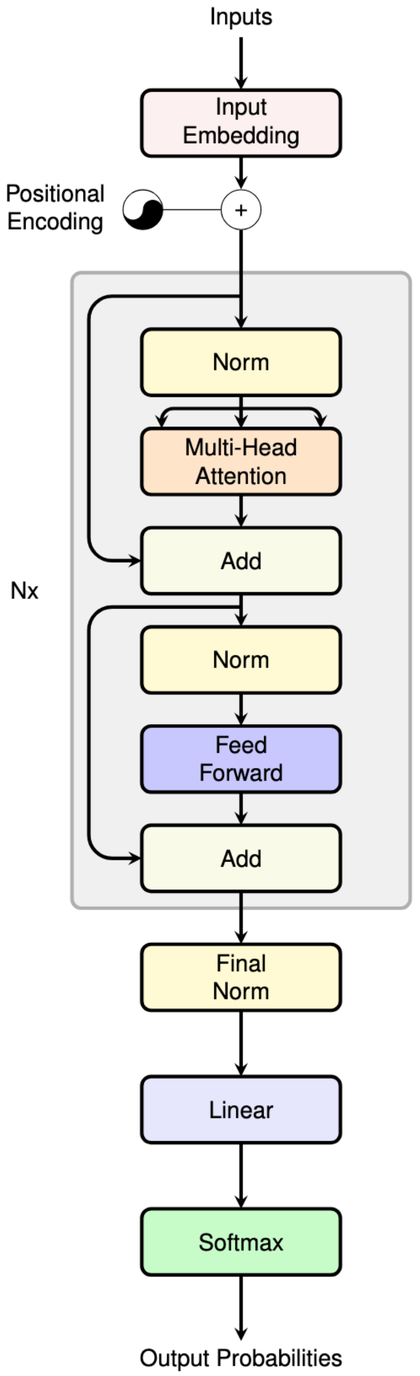
- This code is in root of nanoGPT **model.py**

Multi Head Attention

Another diagram reversed!

- Here we see “h” heads
- Separate linear transform for each of Q K and V for each head
- Q, K and V are are each transformed by linear networks
- i.e. each element of q, k and v becomes a weighted sum of all the elements in the input embedding

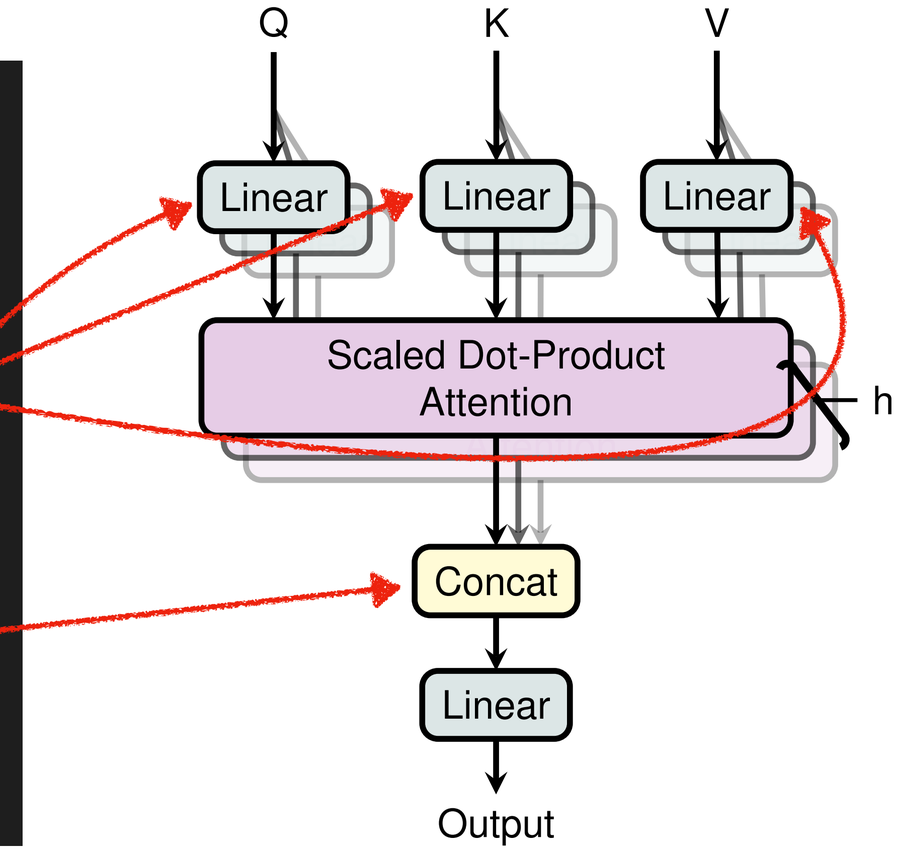




Transformers (Multi-Head)

Upside Down: batch size = 16, sequence length = 64, n_embed = 36

```
1.class CausalSelfAttention(nn.Module):

3.     def forward(self, x):
4.         B, T, C = x.size() # batch size, sequence length, embedding dim (n_embd)


7.         # calculate Q K V for all heads in batch & move head forward to be the batch dim
8.         q, k, v  = self.c_attn(x).split(self.n_embd, dim=2)
9.         k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
10.        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
11.        v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)


14.        # causal self-attention; (B, nh, T, hs) x (B, nh, hs, T) -> (B, nh, T, T)
15.        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
16.        att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
17.        att = F.softmax(att, dim=-1)
18.        att = self.attn_dropout(att)
19.        y = att @ v # (B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
20.        y = y.transpose(1, 2).contiguous().view(B, T, C)
21.        # re-assemble all head outputs side by side
```

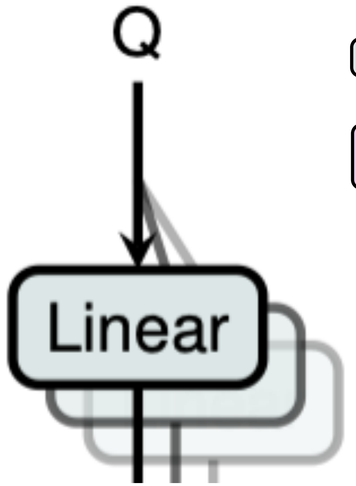


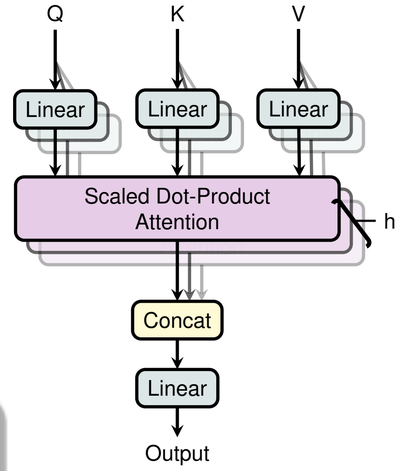
- This code is in root of nanoGPT **model.py** - spreadsheet?

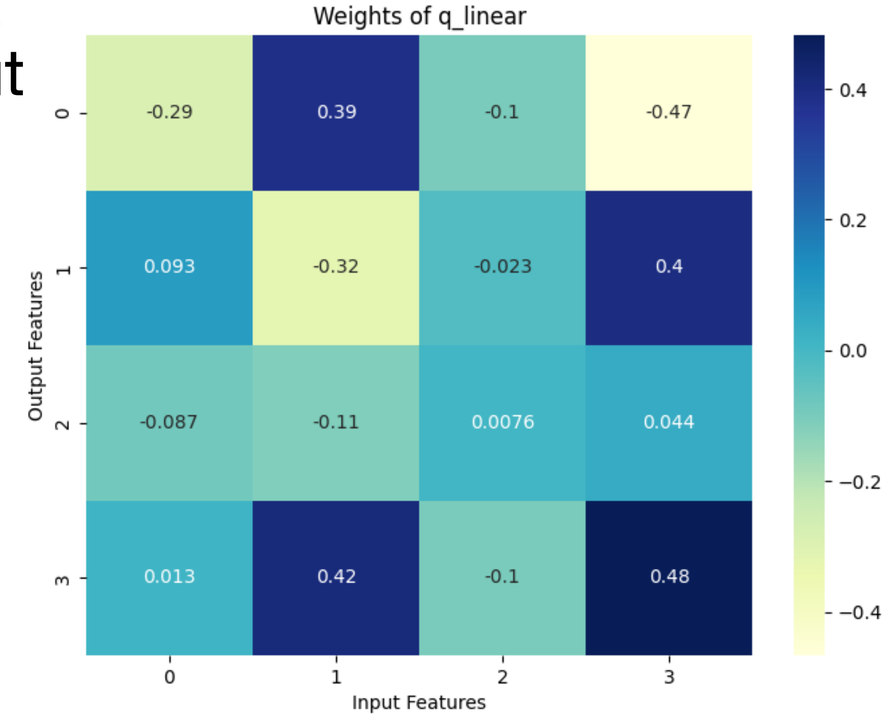
Linear Transformations

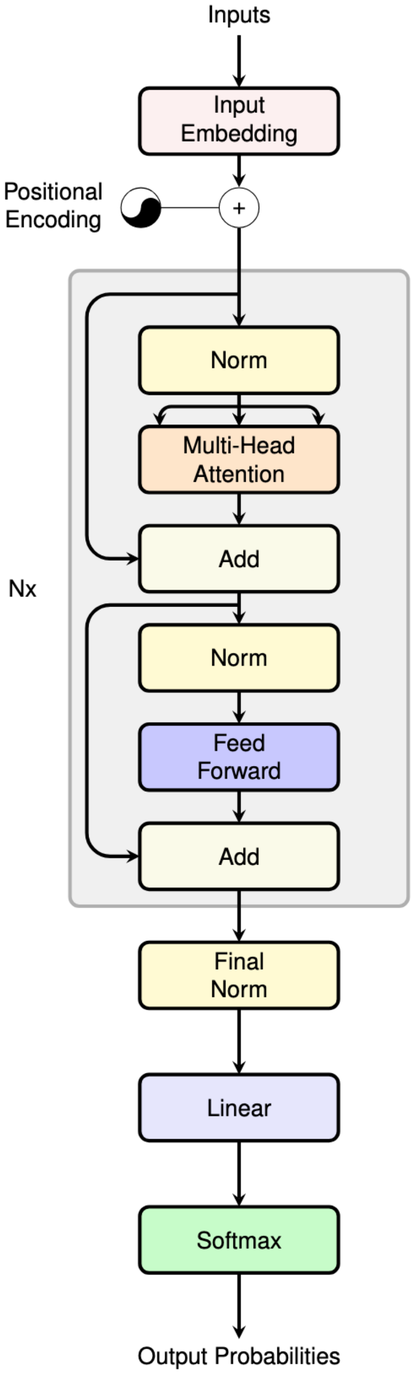
Think different!

- Recombination of all elements of the embedding
- Each output feature will now be a weighted combination of the input features





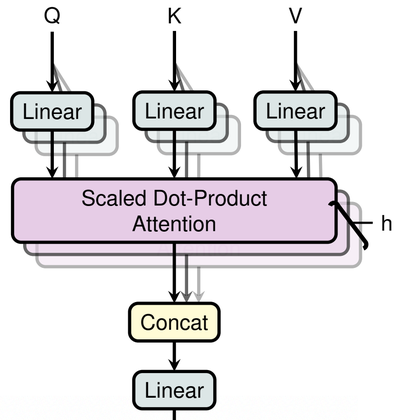


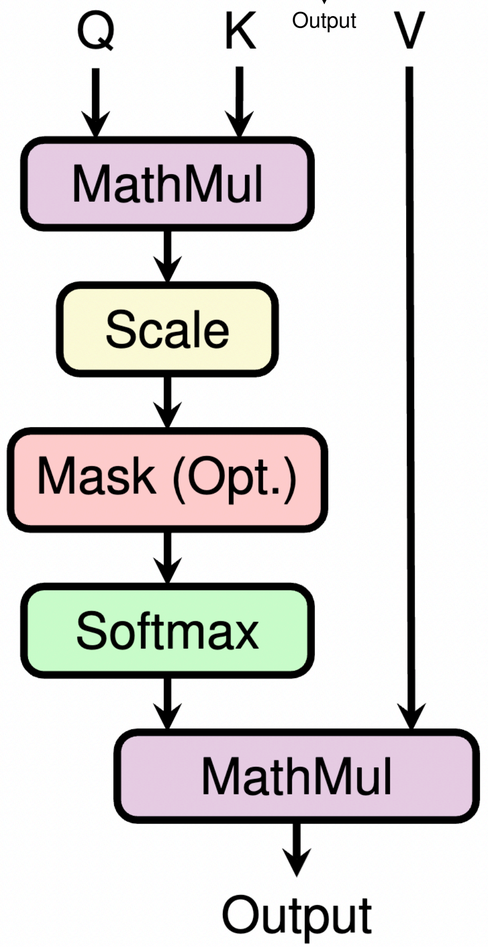


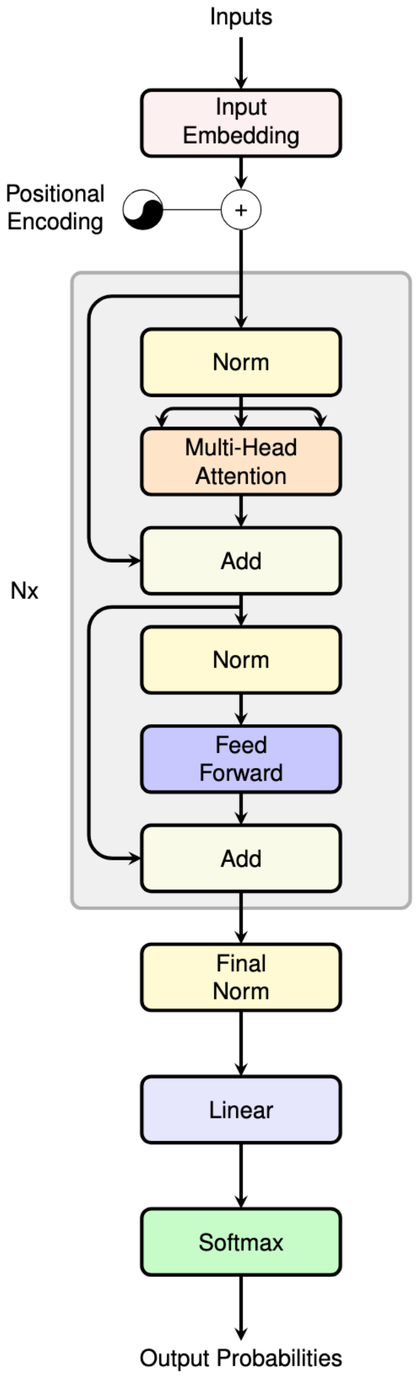
Scaled Dot Product Attention

The final upside down image!

- Q, K and V are linear transformations of the original embeddings
- And we could visualise the linear transformation by combining the word clouds in different combinations … art installation!



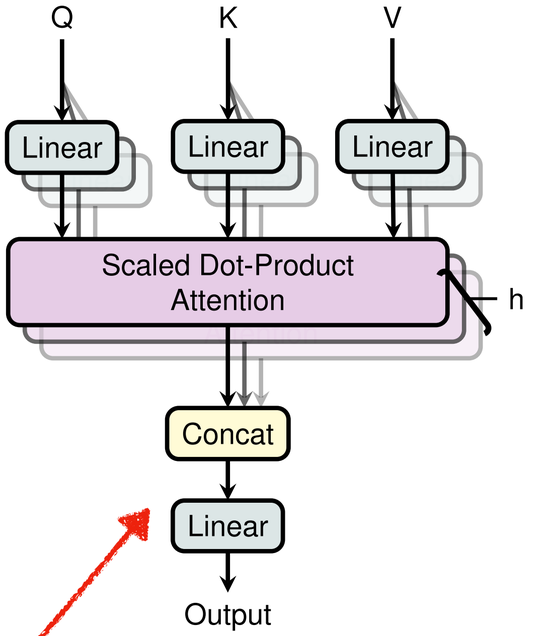


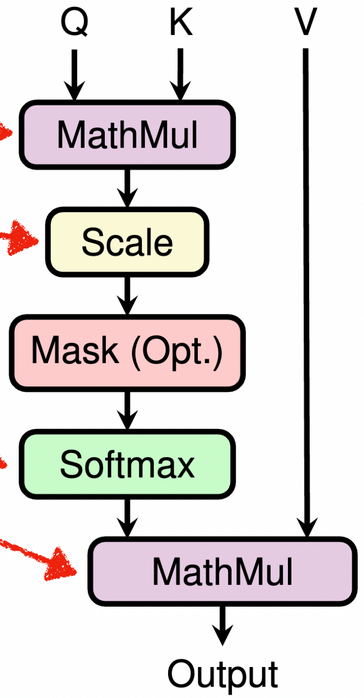


Transformers (Multi-Head)

Upside Down: n_embed=36

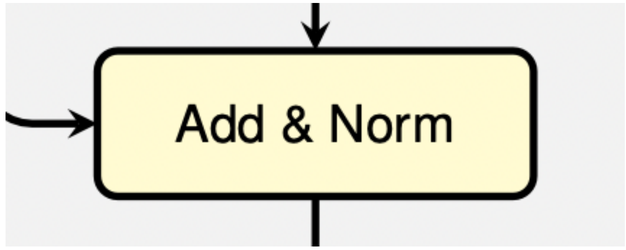
```python
class CausalSelfAttention(nn.Module):

    def forward(self, x):
        B, T, C = x.size() # batch size, sequence length, embedding dim (n_embd)


        # calculate Q K V for all heads in batch and move head forward to be the batch dim
        q, k, v  = self.c_attn(x).split(self.n_embd, dim=2)
        k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
        v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)


        # causal self-attention; (B, nh, T, hs) x (B, nh, hs, T) -> (B, nh, T, T)
        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
        att = att.masked_fill(self.bias[:,:,:T,:T] == 0, float('-inf'))
        att = F.softmax(att, dim=-1)
        att = self.attn_dropout(att)
        y = att @ v # (B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
        y = y.transpose(1, 2).contiguous().view(B, T, C)
        # re-assemble all head outputs side by side
```



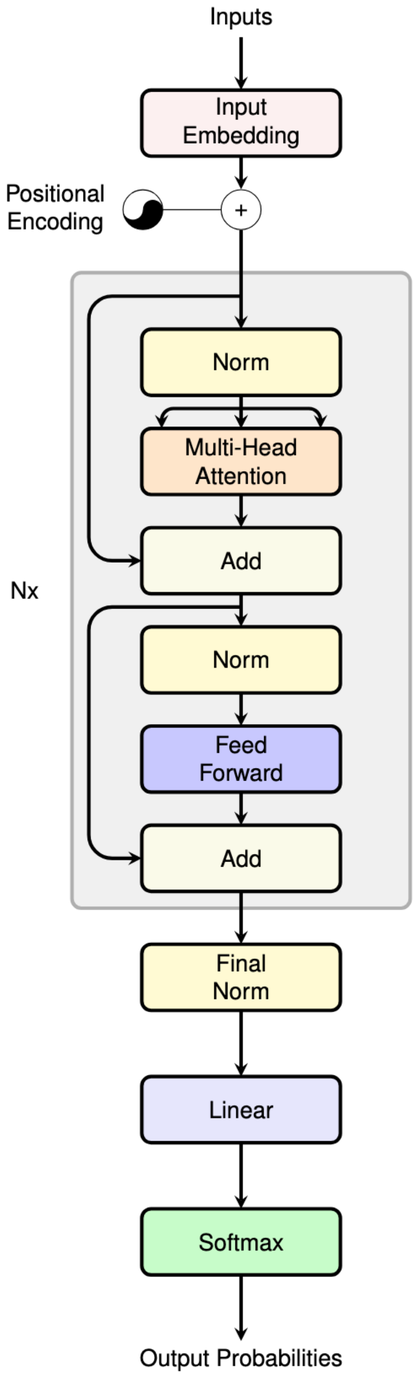


Transformer Block

Adds & Norms

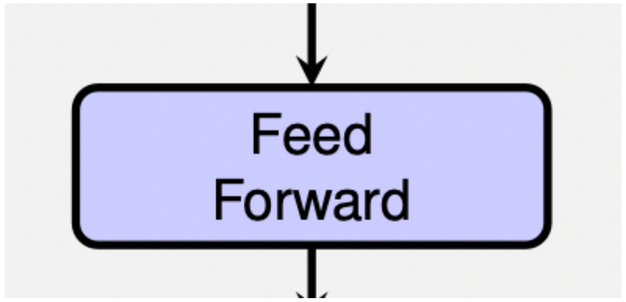


- Add:
  - Implements a residual connection, adding the input of a sub-layer (e.g., attention or feed-forward) to its output
  - Helps preserve original input information across layers
  - Aids in gradient flow, reducing vanishing gradient problems
- Norm (LayerNorm):
  - Applies layer normalisation to stabilise and scale inputs
  - Normalises across the feature dimension (per token)
  - Helps with training stability and faster convergence



Transformer Block

Feed Forward



- Anthropic circuits research shows presence of circuits across multiple layers

- Example here shows producing acronym for an organisations name

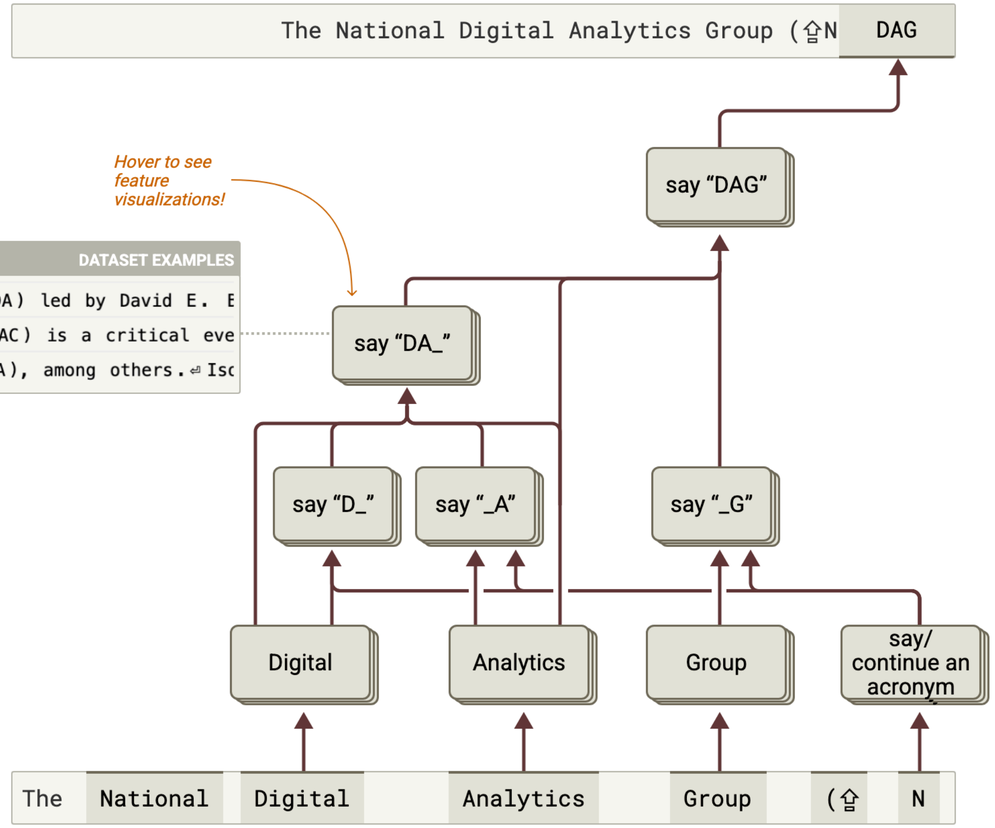
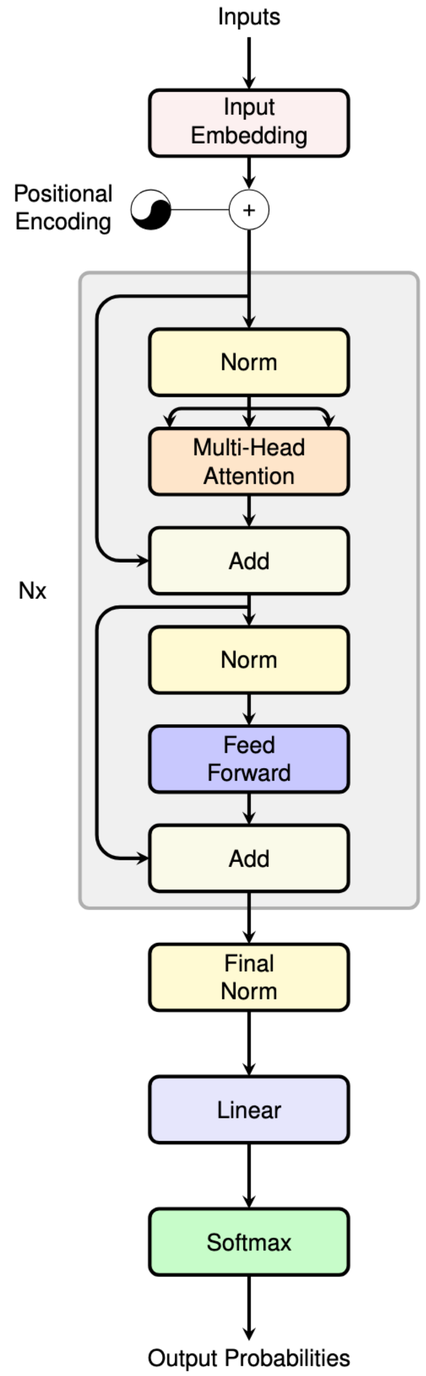
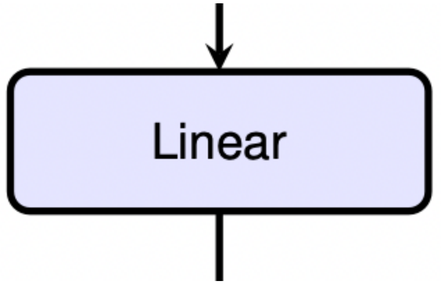


Figure 5: A simplified diagram of the attribution graph for 18L completing a fictional acronym.

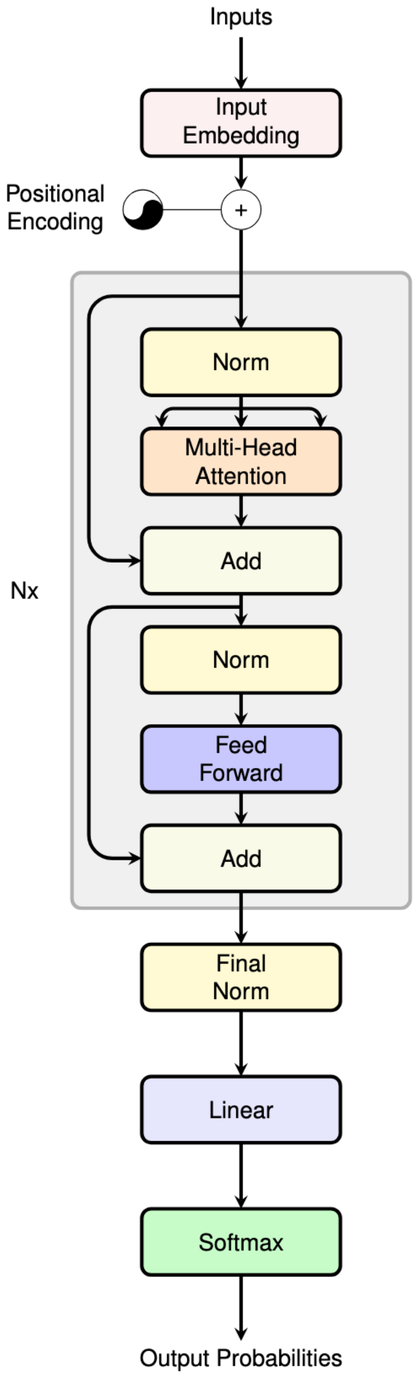


Linear

Back to tokens/words

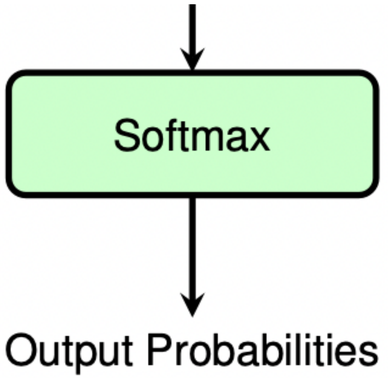


- Maps the hidden state (embedding size) back to the vocabulary size
- Computes logits for each token in the vocabulary (before softmax)
- Enables next-token prediction by assigning scores to each possible token
- Shares weights with the input embedding matrix (weight tying), reducing parameter count and improving generalization

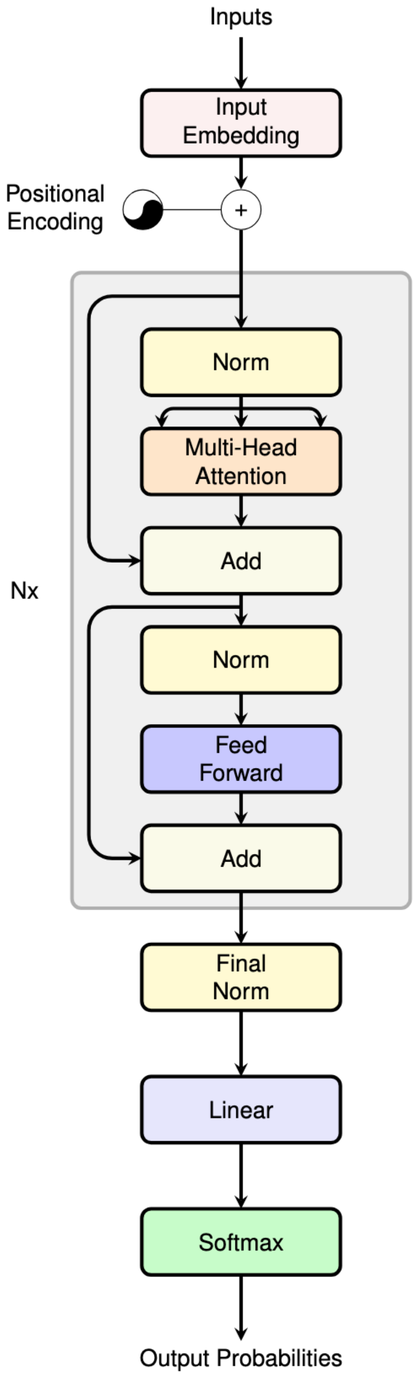


Softmax

Choosing the next token/word



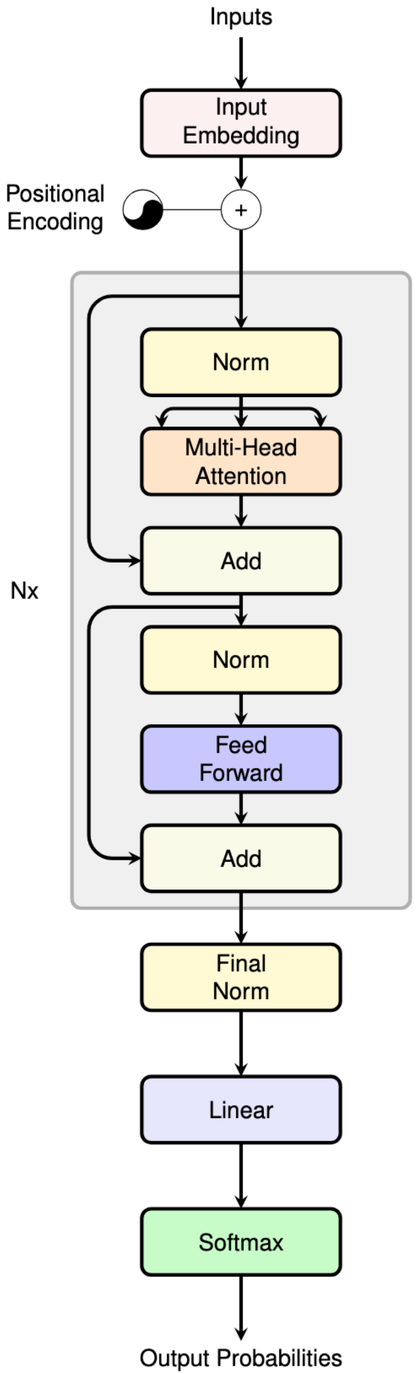
- Converts the logits (raw scores from the final linear layer) into a probability distribution over the vocabulary
- Ensures all probabilities are positive and sum to 1
- Enables sampling or greedy selection of the next token based on likelihood
- Makes the model's output interpretable as a prediction



Backward Pass

Backpropagation

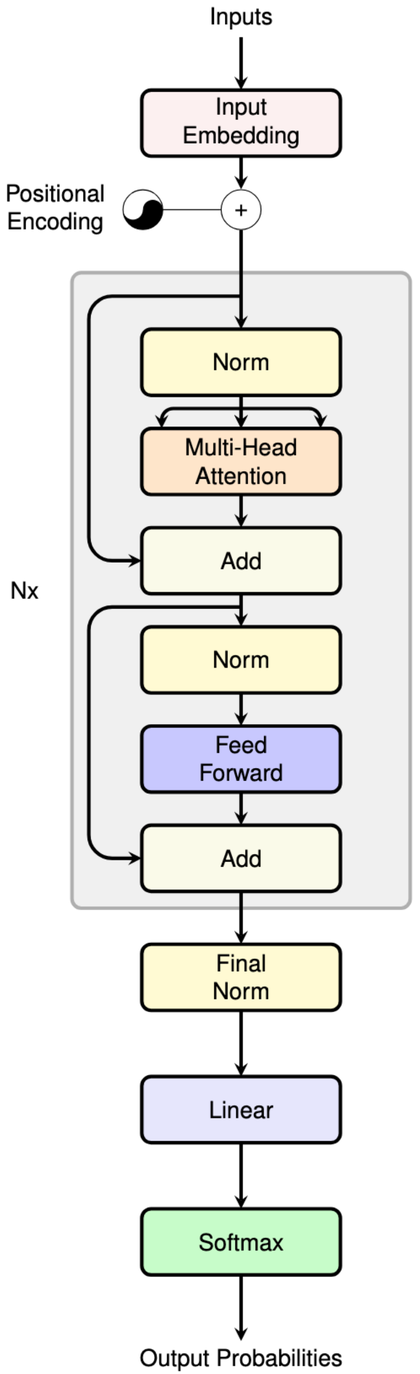
- Blame assignment for fails on training set
- Not really got time to go into in depth today



Demo

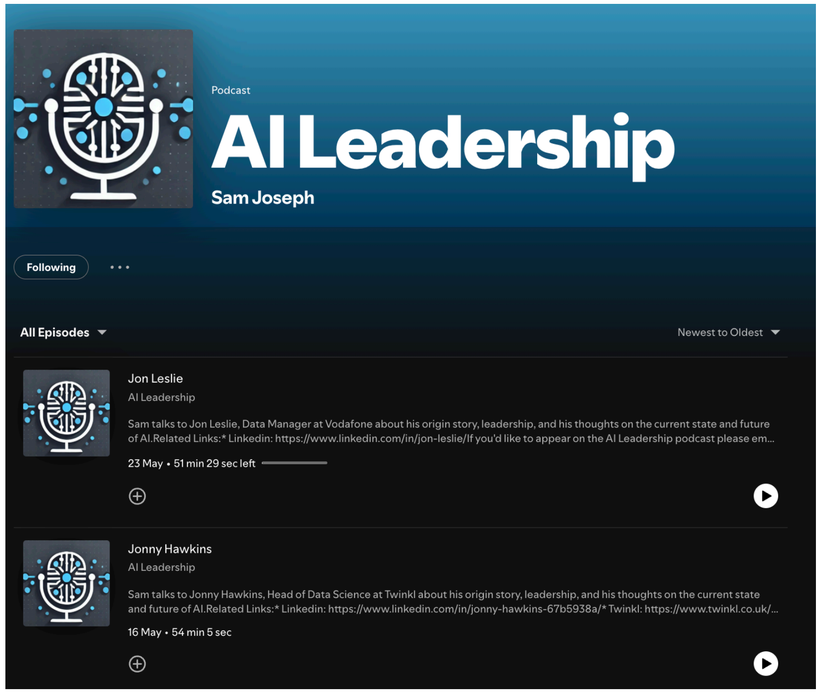
The Code So Far …

- DistilGPT2 6 layer 768 dimensions, 12 heads works 100%
- 6 layer 36 dimensions 1 head works 50% and others close
- Will generate text …
- E.g. for prompt “Knock knock whos there bob” a current completion is
- “Knock knock whos there bob bob who bob and weave”



Questions?

And listen to my podcast!



And come to my Python Engineering Excellence BOF! Sat 3:30pm!

LinkedIn

Amazon

Instagram

And read my book!