Let's build GPT: from scratch, in code, spelled out

https://www.youtube.com/watch?v=kCc8FmEb1nY

Attention Is All You Need (2017)

Model architecture

- Encoder-decoder structure
- Encoder maps input sequence of symbol representations to a sequence of continuous representations
- Decoder then uses these representations to create an output sequence of symbols, one at a time
- At each step the model is auto-regressive, using the generated symbols as additional input
- The transformer follows this architecture using self-attention and point wise, fuilly connected layers for encoder and decoder

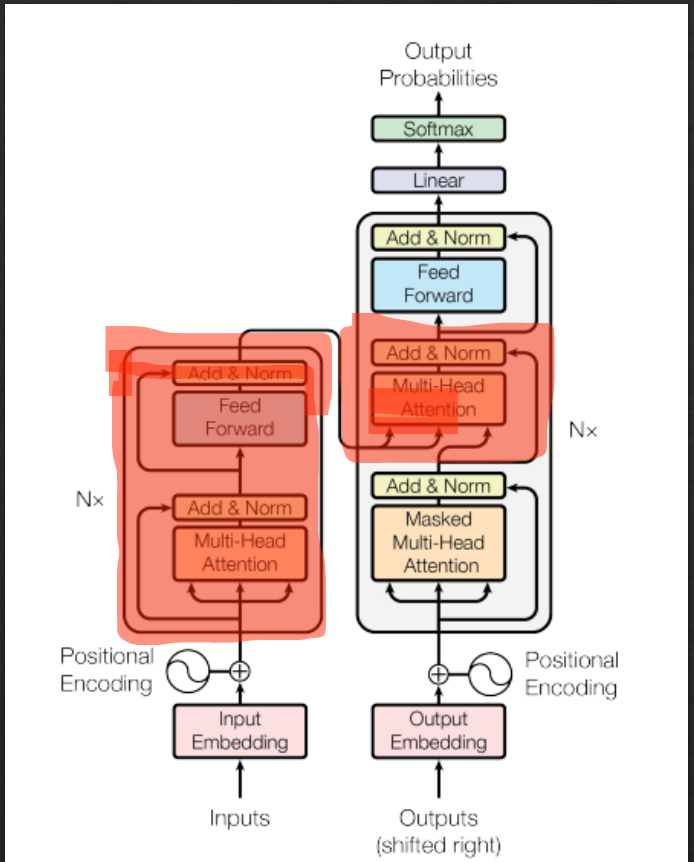


Figure 1: The Transformer - model architecture.

The red parts (encoder and cross attention) are missing in the model
For generating text the decoder is enough

Encoding/Decoding

- Translating characters into integers
- A string will be represented with integers
- This procedure can be made in the opposite direction
- Both have N = 6 identical layers
- Encoder for each layer 2 sublayers, Decoder 3 sublayers

```
[ ]  # create a mapping from characters to integers
     stoi = { ch:i for i,ch in enumerate(chars) }
     itos = { i:ch for i,ch in enumerate(chars) }
     encode = lambda s: [stoi[c] for c in s] # encoder: take a string, output a list of integers
     decode = lambda l: ''.join([itos[i] for i in l]) # decoder: take a list of integers, output a string

     print(encode("hii there"))
     print(decode(encode("hii there")))
```

```
[46, 47, 47, 1, 58, 46, 43, 56, 43]
hii there
```

Encoding

```
[ ]  # let's now encode the entire text dataset and store it into a torch.Tensor
     import torch # we use PyTorch: https://pytorch.org
     data = torch.tensor(encode(text), dtype=torch.long)
     print(data.shape, data.dtype)
     print(data[:1000]) # the 1000 characters we looked at earier will to the GPT look like this
```

```
torch.Size([1115394]) torch.int64
tensor([18, 47, 56, 57, 58,  1, 15, 47, 58, 47, 64, 43, 52, 10,  0, 14, 43, 44,
        53, 56, 43,  1, 61, 43,  1, 54, 56, 53, 41, 43, 43, 42,  1, 39, 52, 63,
         1, 44, 59, 56, 58, 46, 43, 56,  6,  1, 46, 43, 39, 56,  1, 51, 43,  1,
        57, 54, 43, 39, 49,  8,  0,  0, 13, 50, 50, 10,  0, 31, 54, 43, 39, 49,
         6,  1, 57, 54, 43, 39, 49,  8,  0,  0, 18, 47, 56, 57, 58,  1, 15, 47,
        58, 47, 64, 43, 52, 10,  0, 37, 53, 59,  1, 39, 56, 43,  1, 39, 50, 50,
         1, 56, 43, 57, 53, 50, 60, 43, 42,  1, 56, 39, 58, 46, 43, 56,  1, 58,
        53,  1, 42, 47, 43,  1, 58, 46, 39, 52,  1, 58, 53,  1, 44, 39, 51, 47,
        57, 46, 12,  0,  0, 13, 50, 50, 10,  0, 30, 43, 57, 53, 50, 60, 43, 42,
         8,  1, 56, 43, 57, 53, 50, 60, 43, 42,  8,  0,  0, 18, 47, 56, 57, 58,
         1, 15, 47, 58, 47, 64, 43, 52, 10,  0, 18, 47, 56, 57, 58,  6,  1, 63,
        53, 59,  1, 49, 52, 53, 61,  1, 15, 39, 47, 59, 57,  1, 25, 39, 56, 41,
        47, 59, 57,  1, 47, 57,  1, 41, 46, 47, 43, 44,  1, 43, 52, 43, 51, 63,
         1, 58, 53,  1, 58, 46, 43,  1, 54, 43, 53, 54, 50, 43,  8,  0,  0, 13,
        50, 50, 10,  0, 35, 43,  1, 49, 52, 53, 61,  5, 58,  6,  1, 61, 43,  1,
        49, 52, 53, 61,  5, 58,  8,  0,  0, 18, 47, 56, 57, 58,  1, 15, 47, 58,
        47, 64, 43, 52, 10,  0, 24, 43, 58,  1, 59, 57,  1, 49, 47, 50, 50,  1,
        46, 47, 51,  6,  1, 39, 52, 42,  1, 61, 43,  5, 50, 50,  1, 46, 39, 60,
        43,  1, 41, 53, 56, 52,  1, 39, 58,  1, 53, 59, 56,  1, 53, 61, 52,  1,
        54, 56, 47, 41, 43,  8,  0, 21, 57,  5, 58,  1, 39,  1, 60, 43, 56, 42,
        47, 41, 58, 12,  0,  0, 13, 50, 50, 10,  0, 26, 53,  1, 51, 53, 56, 43,
         1, 58, 39, 50, 49, 47, 52, 45,  1, 53, 52,  5, 58, 11,  1, 50, 43, 58,
         1, 47, 58,  1, 40, 43,  1, 42, 53, 52, 43, 10,  1, 39, 61, 39, 63,  6,
         1, 39, 61, 39, 63,  2,  0,  0, 31, 43, 41, 53, 52, 42,  1, 15, 47, 58,
        47, 64, 43, 52, 10,  0, 27, 52, 43,  1, 61, 53, 56, 42,  6,  1, 45, 53,
        53, 42,  1, 41, 47, 58, 47, 64, 43, 52, 57,  8,  0,  0, 18, 47, 56, 57,
        58,  1, 15, 47, 58, 47, 64, 43, 52, 10,  0, 35, 43,  1, 39, 56, 43,  1,
        39, 41, 41, 53, 59, 52, 58, 43, 42,  1, 54, 53, 53, 56,  1, 41, 47, 58,
        47, 64, 43, 52, 57,  6,  1, 58, 46, 43,  1, 54, 39, 58, 56, 47, 41, 47,
```

The entire data set will now be represented with integers

Train -Validation split

- 90% is train set
- 10% validation set

```
# Let's now split up the data into train and validation sets
n = int(0.9*len(data)) # first 90% will be train, rest val
train_data = data[:n]
val_data = data[n:]
```

Bigram language model

- A token embedding table of the size vocab_size by vocab_size is created
- It will be predicted what comes next based on the individual identity of a single token
- So if the token is 5, it is possible to predict what might come next

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

    def __init__(self, vocab_size):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, idx, targets=None):

        # idx and targets are both (B,T) tensor of integers
        logits = self.token_embedding_table(idx) # (B,T,C)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss
```

Generate

- Generate continues the batch dimensions so B, T+x (1, 2, 3, …)
- New idx = B, T+1
- Output is not yet usable, becuase at the moment ist a random model

```python
def generate(self, idx, max_new_tokens):
    # idx is (B, T) array of indices in the current context
    for _ in range(max_new_tokens):
        # get the predictions
        logits, loss = self(idx)
        # focus only on the last time step
        logits = logits[:, -1, :] # becomes (B, C)
        # apply softmax to get probabilities
        probs = F.softmax(logits, dim=-1) # (B, C)
        # sample from the distribution
        idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
        # append sampled index to the running sequence
        idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
    return idx
```

```
torch.Size([32, 65])
tensor(4.8786, grad_fn=<NllLossBackward0>)

SKIcLT;AcELMoTbvZv C?nq-QE33:CJqkOKH-q;:la!oiywkHjgChzbQ?u!3bLIgwevmyFJGUGp
wnYWmnxKWWev-tDqXErVKLgJ
```

Matrix multiplication for weighted aggregation

- C : 14 = 2+6+6 (a*b=c)
- 16 = 7+4+5

```
# toy example illustrating how matrix multiplication can be used for a "weighted aggregation"
torch.manual_seed(42)
a = torch.tril(torch.ones(3, 3))
a = a / torch.sum(a, 1, keepdim=True)
b = torch.randint(0,10,(3,2)).float()
c = a @ b
print('a=')
print(a)
print('--')
print('b=')
print(b)
print('--')
print('c=')
print(c)
```

```
a=
tensor([[1.0000, 0.0000, 0.0000],
        [0.5000, 0.5000, 0.0000],
        [0.3333, 0.3333, 0.3333]])
--
b=
tensor([[2., 7.],
        [6., 4.],
        [6., 5.]])
--
c=
tensor([[2.0000, 7.0000],
        [4.0000, 5.5000],
        [4.6667, 5.3333]])
```

Weighted aggregation with Softmax

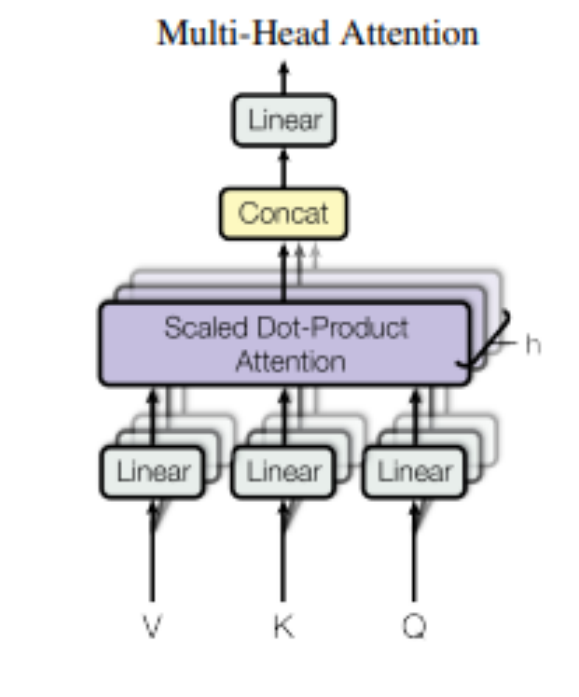
- The tokens interact with each other and are differently attracted/attended to each other

```
# version 3: use Softmax
tril = torch.tril(torch.ones(T, T))
wei = torch.zeros((T,T))
wei = wei.masked_fill(tril == 0, float('-inf'))
wei = F.softmax(wei, dim=-1)
xbow3 = wei @ x
torch.allclose(xbow, xbow3)
```

Self-attention

- Each token in the input creates three vectors: key-, query- and value-vector
- Attention between tokens: dot products between query vector of a token and key vectors of all other tokens
- Attention scores are then scaled (to prevent extrem large products) and passed through a softmax function for normalization (product of 1)
- These softmax scores are then multiplied with the value vectors and the result is summed up. This weighted sum is the output of self-attention for a token
- Multi-Head attention: the self-attention mechanism is performed multiple times
- Output: The output can be further processed to produce a representation of the input sequence. This can then be used for text generation

Multi head attention



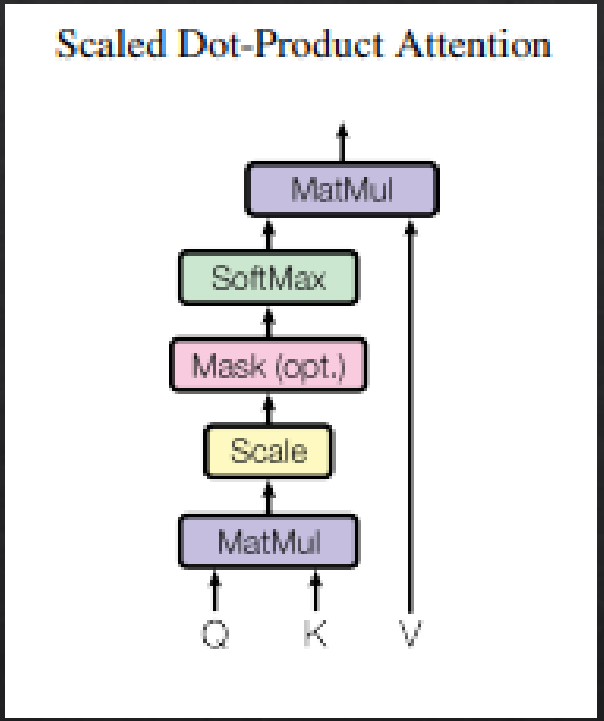

```python
class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
        self.proj = nn.Linear(n_embd, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out
```

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Scaled dot product attention



$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

```
# version 4: self-attention!
torch.manual_seed(1337)
B,T,C = 4,8,32 # batch, time, channels
x = torch.randn(B,T,C)

# let's see a single Head perform self-attention
head_size = 16
key = nn.Linear(C, head_size, bias=False)
query = nn.Linear(C, head_size, bias=False)
value = nn.Linear(C, head_size, bias=False)
k = key(x)   # (B, T, 16)
q = query(x) # (B, T, 16)
wei =  q @ k.transpose(-2, -1) # (B, T, 16) @ (B, 16, T) ---> (B, T, T)

tril = torch.tril(torch.ones(T, T))
#wei = torch.zeros((T,T))
wei = wei.masked_fill(tril == 0, float('-inf'))
wei = F.softmax(wei, dim=-1)

v = value(x)
out = wei @ v
#out = wei @ x

out.shape
```

Position-wise Feed-Forward Networks

- Each layer of encoder and decoder contains a fully connected feed-forward network.
- Input: Can be word embeddings or subword embeddings
- Position-wise: Each position is processed independently -> local patterns and dependencies in the sequence
- Two linear layers: Consists of two linear layers, and a ninlinear activation function.
  - First layer: projects input vector at each position to a higher dimensional space
  - Second layer: maps intermediate representations back to the original dimension
- Activation function: After the first linear layer, to get nonlinearity into the network
- Layer normalization: applied to the output to help normalize the activations and improve the stability

Position-wise Feed-Forward Networks

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- x = input vector
- W1, W2 = weight matrices
- b1, b2 = bias vectors
- max(o, x) = denotes the activation function

References

- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.
- Let's build GPT: from scratch, in code, spelled out.
  - https://www.youtube.com/watch?v=kCc8FmEb1nY