Building GPT from scratch

Problem: Building a next character predictor from scratch. From basic implementation, then adding things sequentially to achieve GPT-like architecture.

Concepts

Tokenization: Creating embeddings mapping the textual data to machine interpretable sequence of numbers, that a neural network can learn from. These embeddings themselves can be created by a neural network. Usually these are high dimensional vector representation of presenting the same information.

Encoder – "Hello" -> [3,5,39, 39, 120]

Decoder – [3,5,39, 39, 120] -> "Hello"

Bigram Model: The most basic implementation of this problem, using (vocab, vocab) embedding matrix with probability of each given character based on past one character. Naturally, increasing the context length increases the embedding dimensionality linearly if you concatenate embeddings of previous tokens, or exponentially if you explicitly model all token combinations, since the model must represent probabilities conditioned on more past tokens.

Attention Trick: Smarter way to increase context. It works by dynamically weighting the past tokens without explicitly storing or concatenating them. Its main benefits:

- Dynamic Context handling (Can look at any prev token instead of just last n tokens)
- Parameter Growth stays constant wrt context

How it works:

1. Create a **lower-triangular mask** (tril) to prevent looking at future tokens.
2. Replace masked positions with –∞ so softmax ignores them.
3. Apply **softmax** row-wise → get normalized attention weights over past tokens.
4. Multiply weights with token embeddings → get a **weighted sum of past embeddings**.

Positional Embeddings: Before using attention, we need to add another type of embedding called positional embedding. They help provide the model latent information about the position of tokens, which attention alone cant capture.

- X' = X + P
  X = token embeddings(different from earlier)
  P = Positional Embeddings

  X' is the embeddings models learn during training

Attention (V. IMP):

Attention is a mechanism that allows tokens to dynamically gather information from all the other relevant tokens, resulting in **Context-aware embeddings**.

Attention works by dividing information in 3 vectors (Q, K, V):

- Q : Queries -> "What this token is looking for?"
- K : Keys -> "What each token contains"
- V : Value -> "The actual content to use"

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

- QK'T is dot product
- Division by sqrt(dk) prevents large values
- dk = head size

Important Notes:

1. Attention is a communication mechanism between the tokens. It can be seen as nodes in a directed graph aggregating the weighted sum from all nodes that point to a node.
2. Attentions blocks have no concept of space. That's why positional encodings need to be used.
3. In encoder block, we don't use triangular mask because we want the encoder to look at all the tokens. In decoder, we use tril mask to make sure past tokens don't communicate with future tokens.

4. **Self-attention** means keys and values are produced from same source as queries. **Cross-attention** means Q gets produced from x, but k and v comes from other sources(eg encoder module)

Multi-Head Attention: Using multiple smaller heads in parallel, each focusing on different relationship or features, then concatenates the output.

Feed Forward:

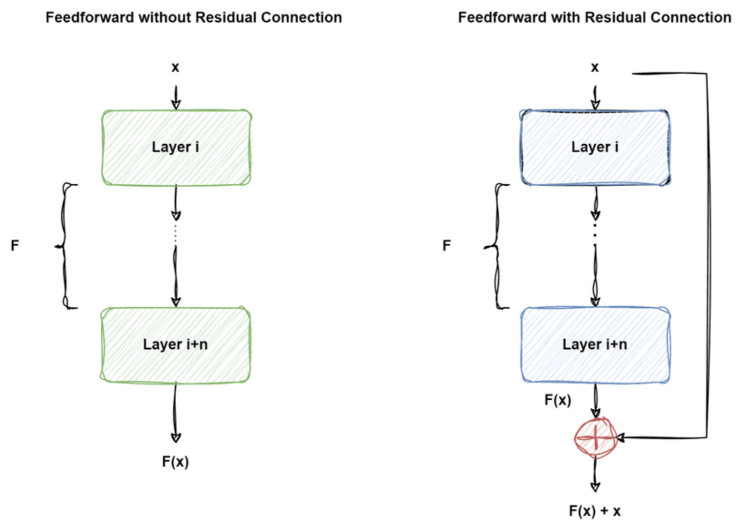
Typically used after Attention to reinterpret the information locally. More specifically, it applies position-wise nonlinear transformations (ReLU/GeLU) so that model can learn from the contextual information generated by Attention blocks. Both together can be seen as the "Thinking" step of transformer.

- Attention gathers info from others
- Feed-forward interprets and transforms that info locally

Residual Connections:

Stabilizes training in deeper neural networks by allowing the gradients to flow directly through the network. It acts as "shortcut" between original representation and the new representation. It motivates the model to learn refinements instead of completely new representations.

- Helps prevent vanishing gradients.
- Allows deeper model training
- Ensures the model doesn't forget original representation while learning new transformations



Feedforward without Residual Connection
x
Layer i
F
Layer i+n
F(x)
Feedforward with Residual Connection
x
Layer i
F
Layer i+n
F(x)
F(x) + x

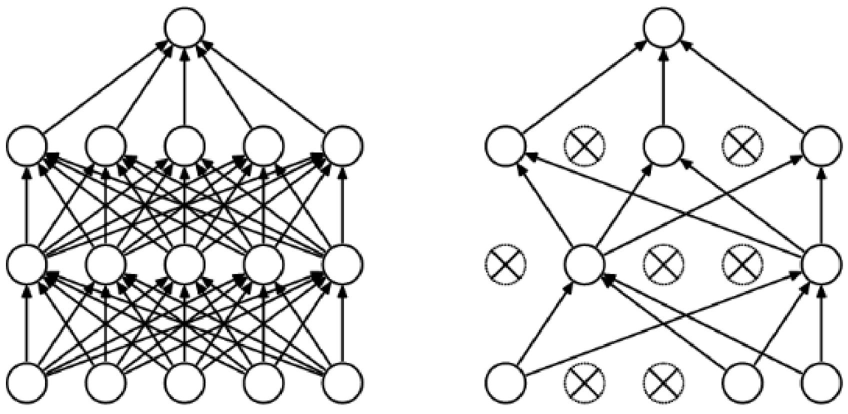
Layer Normalization:

Works similar to BatchNorm, but instead of normalizing across the batches, we normalize across the layers. That is changing the normalization dim from rows to columns. This way, we don't have to worry about running mean/std anymore. And it works same in both training and inference time.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Dropout:

Regularization technique that improves generalization by randomly turning off some percentage of neurons in the network. It prevents overfitting in the network, allowing us to train deeper NN.


(a) Standard Neural Net (b) After applying dropout.