This writeup takes the plain transformer module and builds a GPT with it: ie, we use transformers to create a text-prediction model. Transformers act as the core of the "understands the text sequence" component, and the machinery around it harnesses that capability to coherently predict text.

As a preview, the model we end up with feeds data through the following modules in order:

$$\text{embeddings} + \text{pos. encoding} \to \text{transformer with } N \text{ blocks} \to \text{LN} \to \text{unembed} \to \text{softmax}$$

Also, the LayerNorm on the transformer's output is just because we don't want the essentially random / irrelevant residual stream magnitude to introduce noise to the prediction — pre-LN norms upon every read, not every output, so we LN the transformer's output when we read it.

# 1 Tokenization

Text is converted into a list of integer token IDs. Our vocab is size $V$, each token id is an integer in $[0, V-1]$. IDs alone have no relative semantic meaning to each other.

## 1.1 Embedding module

An embedding module, in general, takes in an index tensor $X$ (indices in range $[0, i_{max}-1]$) of shape shape($X$), and outputs $X'$ of shape shape($X$) + $[d_{out}]$, where $i_{max}$ and $d_{out}$ are hyperparameters of the embedding module. Its goal is essentially to map arbitrary integer indices (entries of $X$) to semantically meaningful $\mathbb{R}^{d_{out}}$ vectors.

We store parameters:

- $W$: shape $[i_{max}, d_{out}]$. Row $i$ of $W$ is the vector we intend index $i$ to map to.

Then, for the forward pass:

- Create index tensor $I$ of shape shape($X$) + $[d_{out}]$. Then, for every index tuple $j$ that indexes $X$, $\boxed{I[j] = (X[j_{-k}], j[k])}$ if $k$ is the axis of $d_{out}$ features. $X[j_{-k}]$ is the ID which collects the correct row from $W$, and $j[k]$ collects the row of features as you move along the feature axis in $I$.
- Then, $\boxed{X' = W.\text{gather}(I)}$.

Suppose $p(j)$ is the element of $j$ corresponding to the sequence position axis.

- Semantic embedding sets $X[j] = t_{p(j)}$ where $t_{p(j)}$ is the ID of the token at position $p(j)$. We set $i_{max}$ to be the vocab size.
- Positional encoding sets $X[j] = p(j)$, tying the token's info to its position. We set $i_{max}$ to be the sequence length.
- Note that we want to **add** these two vectors together for any token ID's true embedding in $\mathbb{R}^{d_{out}}$; these two embedding modules must match $d_{out}$ values.

Conceptual notes:

- The embedding module's purpose is just to make info flow directly from token IDs to their own vectors, letting the network treat tokens as categorical indices instead of numbers with magnitude. It's intentionally simple and permissive to let the model decide where they'll go from the base we give it.
- Remember, we're giving the module tools that hard-code patterns we believe are important (like magnitude invariance) and letting the module figure out how to best use its tools, rather than making the module make its own tools / approximate tools we're already confident are important. Generally, we more tightly define info flow structure regarding info we're confident we understand, and more loosely define info flow structure regarding info we're less confident we understand. The tighter the structure, the easier it is to optimize, but the looser the structure, the more expressive it can be.

## 1.2 Unembedding module + output softmax

We'll store parameters:

- $W_U$: shape $[d, V]$, maps $\mathbb{R}^d$ vectors back to logits over vocab. It's the last "understanding" step before softmax, so each column of $W_U$ is directly shaped so dot product against the transformer's output $\mathbb{R}^d$ vector gives confidence on the corresponding entry in vocab, but the column data itself is not necessarily interpretable relative to the vocab since the actual mechanics of how they produce logits deals with already-likely-uninterpretable data — the transformer's output.

If the output of the transformer is $X$ of shape $[T, d]$, then the output of the unembedding module is $XW_U$ of shape $[T, V]$ — each row contains a "logit distribution" over the vocab. This is interpretable because it's tied closely enough to the training objective, only one softmax away; explained in the next training objective section.

The unembedding module can be implemented with a linear module — the same one used in building dense neural networks.

# 2 Training objective

Since the transformer already creates an output for each $\mathbb{R}^d$ vector in the input sequence (it's $[T, d] \to [T, d]$ and does the same independent "kind" of work for each $\mathbb{R}^d$ vector in the sequence), this speeds up training — make the transformer predict the output token for every prefix of the input sequence.

However, one fix to make this work: since each token attends to all tokens in the sequence, not only the tokens that precede it, each token already has access to the token that comes after it. This almost guarantees the model will cheat and use the existing next tokens in the input sequence directly as the "predicted" tokens — info that obviously doesn't exist when a sequence is being generated.

- So we add **causal masking** to the attention function: $\boxed{\text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V}$ where $M$ is of shape $[T, T]$ and has $M_{ij} = -\infty$ for all $j > i$ and $M_{ij} = 0$ otherwise, which zeros the influence of future tokens $j > i$ on a prediction on a prefix $x_0, ..., x_i$.

We also choose cross-entropy as loss, in order to make the model predict tokens given a prefix at the frequencies they appear in the training set. $P(x_{i+1} \mid x_0, ..., x_i)$ is the prob we predict $x_{i+1}$ should be the next token, given the prefix $x_0, ..., x_i$; lean on law of large numbers to implicitly compute the "source distribution" we're fitting against despite not counting prefix-token frequency pairs in the data beforehand.

$$\boxed{L(\theta) = -\frac{1}{T}\sum_{i=0}^{T-1} \log P_\theta(x_{i+1} \mid x_0, ..., x_i)}$$

Make sure sequences grabbed from training data are length $T + 1$ if sequences of length $T$ are fed into the model; you want $x_0, ..., x_T$ where the model operates on $x_0, ..., x_{T-1}$ and the training objective is $x_1, ..., x_T$.

# 3 Training run

Let $M$ be the number of examples we want. We'll sample $M$ examples, each of which is a sequence of length $T + 1$; first $T$ is input, last $T$ is output. Prepare input and output $X, Y$ of shapes $[M, T]$ and $[M, T, V]$ respectively, where each entry in $X$ is a token ID, and each last-axis vector (len $V$) in $Y$ is a one-hot encoding of a token ID.

- We do this because the model should learn embeddings for $X$, that's how useful embeddings are learned, so we pass in batched sequences of token IDs; and the model outputs predictions as distributions over vocab with that shape, so we make $Y$ match it.

# 4 Inference

If you feed in a sequence $x_0, ..., x_{T-1}$, the predicted $x_T$ is sampled according to the $T$th output softmax distribution.