Causal Masking

This post specifically goes into Casual Masking, and why we use it. This is mostly for my understanding of the topic and the Workshop

During attention, when we compute the attention scores like this:

$$S = \frac{QK^T}{\sqrt{d}}$$

where S_{ij} is how much token i wants to look at token j.

A casual matrix M is such where:

$$M_{ij} = \begin{cases} 0 & j \leq i \\ -\infty & j > i \end{cases}$$

The mask is lower-triangular, where every position above the diagonal is $-\infty$ or in practice, `-1e9`.
Then you compute a softmax with this matrix added to the score matrix S:

$$\text{softmax}(S + M)$$

Numerically, what happens is that the the dot product of extremely large negative numbers, collapses to near-zero, which essentially, removes all future positions from the distribution, which means the model can no longer look *back.*

Why do we need to do this at all?

This is use-case dependent. In the original paper by Vaswani et al., they don't actually use casual masking on the encoder part of the transformer. The decoder part however is still **autoregressive**, and does use casual masking.

Semantically, a decoder-only autoregressive model, such as GPT or Claude or Gemini needs casual masking. The task in language modeling is to learn:

$$p\left(x_i | x_1, \ldots, x_{i-1}\right) \qquad (1)$$

This is the only condition. All language, all syntax, all long-range structure and planning comes from being forced to predict under uncertainty about the future.

If you allow the model to attend to tokens in reverse, you are no longer solving the same problem. The model is no longer learning the distribution instead, it's learning this:

$$p\left(x_i | x_1,\ \ldots,\ x_{i-1},\ x_{i+1},\ \ldots\right) \qquad (2)$$

Which is a far easier problem, and a mostly degenerate one.

During training, you already have the full sequence. If future tokens are visible, the model can:

- Learn near-identity maps (keys line up with future values)
- Encode “the answer” directly in attention
- Stop modeling uncertainty altogether

In the extreme, the optimal solution becomes:

> “Look at the next token and copy it.”

That achieves near-zero training loss without learning anything generative.

This isn’t hypothetical. If you remove the causal mask and train a decoder-only model, it *will* converge faster and then completely fail at generation.

At inference, the future does not exist yet. So you’re asking the model to operate under (1) but it was trained under the assumption of (2) and it collapses completely.

Example, if you teach a child to do math, and allow them to look at the back for answers, and see them get every answer correct, but during test time, when there is no answer key, they don't pass. The child hasn't learned to do math, the child has learnt to look at the answer key.

The Implementation

```
>>> import torch
>>> import torch.nn.functional as F
>>> import math
>>> seq, embd = 5, 4
>>> x = torch.randn((seq, embd))
>>> q, k, v = x, x, x # this is replaced by learned reprs. in practice
>>> # triangle_upper, since we want to mask the upper triangle.
>>> # diagonal=1 => how my diagonals to skip. 0 = none, 1 = 1 diagonal, ....
>>> mask = torch.triu(torch.ones((seq, seq)), diagonal=1).bool() ; mask
tensor([[False,  True,  True,  True,  True],
        [False, False,  True,  True,  True],
        [False, False, False,  True,  True],
        [False, False, False, False,  True],
        [False, False, False, False, False]])
>>> scores = (q @ k.T) / math.sqrt(embd) ; scores
tensor([[ 1.1037e+00,  1.3700e+00,  3.4402e-03, -7.2684e-02,  1.3372e-01],
        [ 1.3700e+00,  3.8073e+00,  9.3326e-01, -6.9241e-01, -1.9216e-01],
        [ 3.4402e-03,  9.3326e-01,  2.7168e+00, -1.8498e+00, -7.4956e-01],
```

```
        [-7.2684e-02, -6.9241e-01, -1.8498e+00,  1.2658e+00,  4.8337e-01],
        [ 1.3372e-01, -1.9216e-01, -7.4956e-01,  4.8337e-01,  4.3930e-01]])
>>> scores = scores.masked_fill(mask, -1e9) # `float('-inf')` works but it's
more numerically stable to just use `-1e9`
>>> scores
tensor([[ 1.1037e+00, -1.0000e+09, -1.0000e+09, -1.0000e+09, -1.0000e+09],
        [ 1.3700e+00,  3.8073e+00, -1.0000e+09, -1.0000e+09, -1.0000e+09],
        [ 3.4402e-03,  9.3326e-01,  2.7168e+00, -1.0000e+09, -1.0000e+09],
        [-7.2684e-02, -6.9241e-01, -1.8498e+00,  1.2658e+00, -1.0000e+09],
        [ 1.3372e-01, -1.9216e-01, -7.4956e-01,  4.8337e-01,  4.3930e-01]])
>>> attn = F.softmax(scores, dim=1)
>>> attn # notice how everything above diagonal is 0.
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000], # token 0 only sees itself
        [0.0804, 0.9196, 0.0000, 0.0000, 0.0000], # token 1 sees itself and
0
        [0.0537, 0.1361, 0.8101, 0.0000, 0.0000], # token 2 sees 0, 1, 2
        [0.1811, 0.0975, 0.0306, 0.6907, 0.0000], # ...
        [0.2036, 0.1470, 0.0842, 0.2888, 0.2764]])
>>> # see how token 1 has full attention to itself...
>>> attn @ v
tensor([[ 0.4920,  0.2533, -1.3494,  0.2833],
        [-0.6409, -0.7308, -2.4200, -0.2402],
        [ 0.6196, -1.5246, -0.6432, -1.0218],
        [-0.4017,  0.7434, -0.3175,  0.5210],
        [-0.1386,  0.1804, -0.5905,  0.3895]])
```