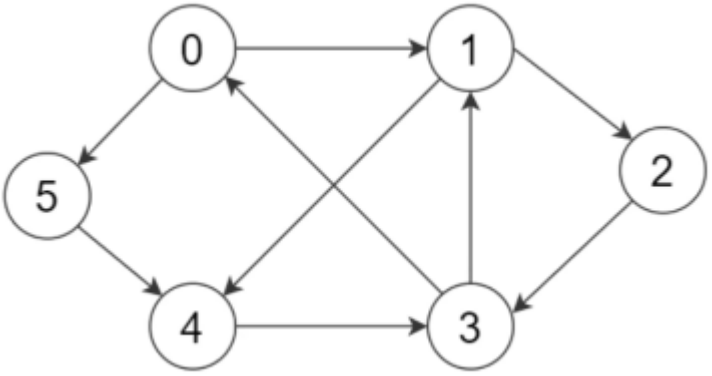
Self Attention Notes

Logic behind self attention

- Attention is a communication mechanism, where we have a number of nodes in a directed graph. Each node has some vector information and it gets to aggregate information via a weighted sum from all of the nodes that point towards it (This is data dependent)



- However our graph may look different from this as we have 8 nodes (our chosen block size) and there are always 8 tokens in a window. The graph goes as follows:
    - 1st node is only pointed to by itself
    - 2nd node pointed to by 1st and itself .....

    - 8th node pointed to by all the previous nodes and itself
- Attention mechanism can be applied to any collection of nodes in a directed graph
- Also there is no notion of space. Attention only works a certain set of vectors.
    - by default these nodes have no idea where they are positioned in space
    - This is why we need to encode them positionally. This is used to anchor them to their specific position.
    - This is different from convolution, coz there is a strict layout to be followed in space. Also the convolution filter acts in space. Whereas if we wanna have a notion of space in self attention, we need to add it ourselves.
- Each example across the batch dimension is of course processed completely independently and never 'talk' to each other.

** To create transformers for different tasks like sentiment analysis we would have to create the transformer as two halves: the encoder and decoder. The encoder would be the same as standard self attention block just without the upper triangular masking (allowing for all nodes to communicate with each other). And the decoder block would be the standard self attention block.

- "self-attention" means that the values and keys both come from the same source as queries. In "cross attention", the queries get produced from a different location than from the keys and values (from the encoder module).
- "Scaled" attention additionally divides wei by 1/sqrt(head_size). This makes it so that when the input Q, K are unit variance, wei would also be unit variance and the softmax on wei would stay diffused and not saturate too much.
- Formula of self attention:

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

Here Q =Query, K=Key, V=Value, dk = head size.

Self Attention Simplified Code and Review

What is Self-Attention?

Self-attention allows the model to focus on different parts of the input sequence when making decisions, such as predicting the next word in a sentence. Instead of just looking at the immediate previous word, it can focus on relevant words from the entire sequence.

Think of it like this: when you read a sentence, sometimes you need to "attend" to earlier words to understand the meaning. Self-attention mimics this process mathematically.

Step-by-Step Breakdown of the Code:

1. **Input (`x`)**:
    - You start with a batch of sequences (`x`), where:
        - `B` is the batch size (4 sequences processed in parallel).
        - `T` is the sequence length (8 tokens or words in each sequence).
        - `C` is the number of channels (32-dimensional embeddings for each word/token).
    - So `x` has shape `(B, T, C)`, which means 4 sequences, each 8 tokens long, and each token is represented by a 32-dimensional vector.

```
x = torch.randn(B,T,C)  # Randomly initialized input ten
sor
```

2. **Keys, Queries, and Values**:
    - Self-attention is all about comparing tokens in the sequence to each other. You do this with **keys**, **queries**, and **values**.
        - **Key**: Represents what information each token contains.
        - **Query**: Represents what each token is looking for.
        - **Value**: Contains the actual information each token wants to pass on.
    - You create linear transformations for the keys, queries, and values from the input `x`.

```
key = nn.Linear(C, head_size, bias=False)
query = nn.Linear(C, head_size, bias=False)
value = nn.Linear(C, head_size, bias=False)
```

- Each transformation maps the 32-dimensional input (`C=32`) down to a smaller `head_size=16`. This makes the attention mechanism lighter and faster.

```
k = key(x)   # (B, T, 16) - the 'key' for each token
q = query(x) # (B, T, 16) - the 'query' for each token
```

3. **Attention Scores (`wei`)**:
   - **Attention scores** determine how much each token should focus on every other token in the sequence. These scores are calculated by taking the dot product between queries and keys.
   - The formula for attention scores is:

     $$\text{Attention Score} = q \times k^\top$$

     - `q` is the query for a token, and `k^\\top` is the transposed key for all tokens.
     - This gives a matrix of shape `(B, T, T)`, where each token attends to every other token in the sequence.

```
wei = q @ k.transpose(-2, -1)  # (B, T, T) - attention scores for all tokens
```

   - **Example**: Imagine you're trying to predict the next word in a sentence. For each word (query), you're calculating how much attention it should pay to every other word (key) in the sentence.

4. **Masking Future Tokens**:
   - In this example, the model should only attend to the **past and present tokens**, not future ones. This is crucial when you're training on sequences like text because you don't want to cheat by looking at future words.

- The `torch.tril(torch.ones(T, T))` creates a triangular matrix, where values above the diagonal are 0 (blocked), meaning future tokens won't be considered.

```
tril = torch.tril(torch.ones(T, T))  # Lower triangular
matrix to block future tokens
wei = wei.masked_fill(tril == 0, float('-inf'))  # Mask
future tokens with -infinity
```

- **Example**: Imagine you're reading a sentence one word at a time. If you're at word 3, you can only focus on words 1, 2, and 3. You shouldn't be able to "peek" at words 4, 5, etc. This triangular matrix ensures that you only pay attention to the current and previous words.

5. **Softmax on Attention Scores**:
   - Once the attention scores are calculated, you apply **softmax** to convert these raw scores into probabilities, so that they sum to 1.
   - The higher the attention score, the more focus a token will have on another token. The softmax ensures that you give more weight to relevant tokens.

```
wei = F.softmax(wei, dim=-1)
```

   - **Example**: If you're at word 3, you might want to pay 80% attention to word 2 and only 20% attention to word 1. Softmax helps you figure out these probabilities.

6. **Applying the Values**:
   - The final step is to take the **value** (`v`) for each token and multiply it by the attention weights (`wei`). This effectively selects which tokens' values will contribute most to the final output.
   - The attention-weighted values are then combined to produce the final output (`out`), which is the weighted sum of all tokens' values.

```
v = value(x)  # (B, T, 16) - the 'value' for each token
out = wei @ v  # (B, T, 16) - weighted sum of values
```

- **Example**: Imagine you're at word 3. Based on the attention scores, you'll take 80% of the information from word 2 and 20% from word 1 to form a new understanding of word 3. The final `out` tensor reflects this combined information.

Summary of the Code Logic:

1. **Input (`x`)**: A batch of sequences, where each token is represented by a 32-dimensional vector.
2. **Keys, Queries, and Values**: These are computed from the input `x` using linear transformations.
3. **Attention Scores**: The query for each token is compared to the keys of all tokens to compute how much attention each token should pay to every other token.
4. **Masking**: Future tokens are masked to prevent the model from looking ahead.
5. **Softmax**: Attention scores are converted to probabilities to weight the importance of different tokens.
6. **Weighted Sum of Values**: The attention probabilities are used to combine the values, producing a new output for each token that takes into account the most relevant tokens from the sequence.

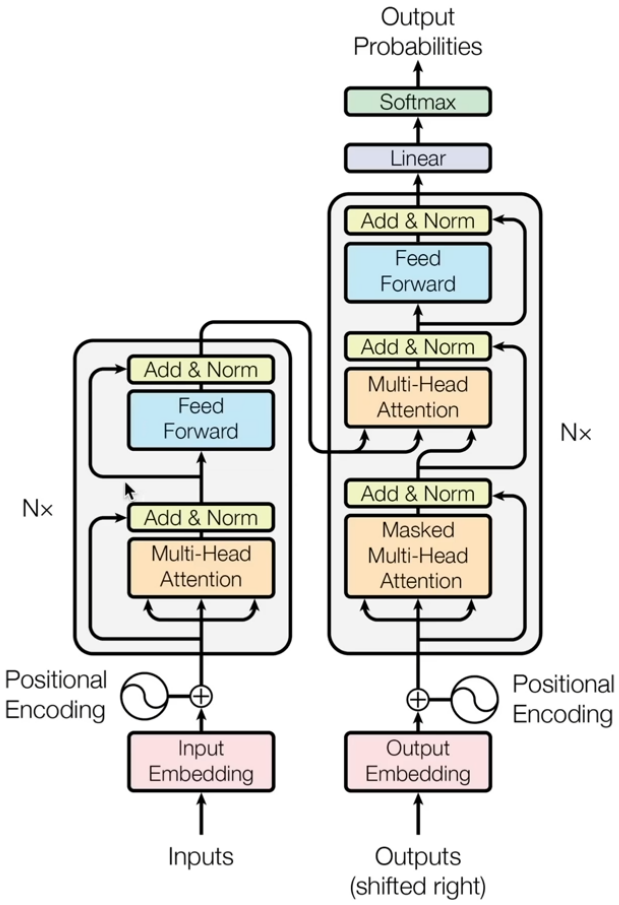
Intuitive Example:

Think of this like being in a classroom discussion. Each student (token) is listening (query) to other students (keys) and deciding who to pay attention to. Some students might have more important information (values). Based on how relevant each student's comment is, you'll focus more on some students than others. Masking ensures that you're not hearing comments from students who haven't spoken yet (future tokens). Finally, you combine all the relevant information and form your new understanding of the discussion (output).

This is exactly what self-attention does, but mathematically!

Multi-headed Attention:

- Unlike the previous one, here we use multiple heads in parallel
- Diagram for Multi-headed Attention



- In the model currently implemented on Colab, we have build an only decoder model. That means, it doesn't have the first block in the above image, and has only the second(right) block.
- The reason for the decoder is that we are just re-generating text and it's not conditioned on anything.
- Also, it's because we are using a triangular mask in our transformer, so it has this auto-regressive property where we can just go and sample from it.
    - The reason why the original paper has both encoder and decoder is because it's actually a machine translation model. It gets inputs in a different language and condition on it. The generation starts with a special new token, (Eg: <start> and <end>).
    - The generation is conditioned on the basis of additional info.
    - The Info reads the French Part, and creates tokens.
    - They then put a transformer on it, without a triangular mask. This allows each token to communicate exactly how they want to. this allows them to encode the contents of the French Sentence.

    - Now, to connect the encoder and the decoder block, cross attention is used. In the above figure, the second Multi-Headed Attention block get’s it’s keys and values from the encoder.
- Here, since we don’t have any conditions on the generation, we are just trying to re-create the text on an input file, so we use only the decoder model.