Self-Attention Mechanism

Uzair Ahmad

Self-attention, also known as intra-attention or internal attention, is a mechanism used in deep learning models, particularly in the context of natural language processing (NLP) and sequence modeling tasks like machine translation, text summarization, and sentiment analysis.

The concept of attention in neural networks is inspired by how humans focus on relevant parts of information when processing inputs. Self-attention extends this idea to allow the model to focus on different parts of the input sequence when processing each element in the sequence.

Here's how self-attention works:

1. **Input Embeddings**: Initially, each input token or word is transformed into three vectors - Query (Q), Key (K), and Value (V). These transformations are typically linear projections (learned matrices).
2. **Calculating Attention Scores**: For each word in the sequence, attention scores are computed with respect to every other word in the sequence. These scores represent the relevance/importance of other words for the current word. The attention score between a word at position i and a word at position j is calculated as the dot product of their respective Query and Key vectors:
   $\text{Attention}(Q_i, K_j) = \frac{Q_i \cdot K_j}{\sqrt{d}}$
   Here, d represents the dimension of the vectors.

   The attention scores are passed through a softmax function to obtain normalized weights, ensuring that the weights sum up to 1. These weights determine how much focus each word should place on other words. The softmax operation ensures that the attention is spread among all words but weighted towards the more relevant ones.
   $\text{Attention\_Weights}(Q_i, K_j) = \frac{\exp(\text{Attention}(Q_i, K_j))}{\sum_{j'} \exp(\text{Attention}(Q_i, K_{j'}))}$
3. **Weighted Sum of Values**: Finally, the weighted sum of the Value vectors is computed using the obtained attention weights. This weighted sum represents the output of the self-attention mechanism for the current word in the sequence.
   $\text{SelfAttention}(Q_i, K, V) = \sum_j \text{Attention\_Weights}(Q_i, K_j) \times V_j$
4. **Multi-Head Attention**: In practice, self-attention is often performed multiple times in parallel, each with its own set of learned parameters. This is called multi-head attention, and it allows the model to focus on different aspects of the input sequence simultaneously.

Step by step example

Let's walk through each step of the self-attention mechanism using an example batch of 6 sentences, where each sentence is of length 8 and each word is represented by a vector of size 32.

Given:

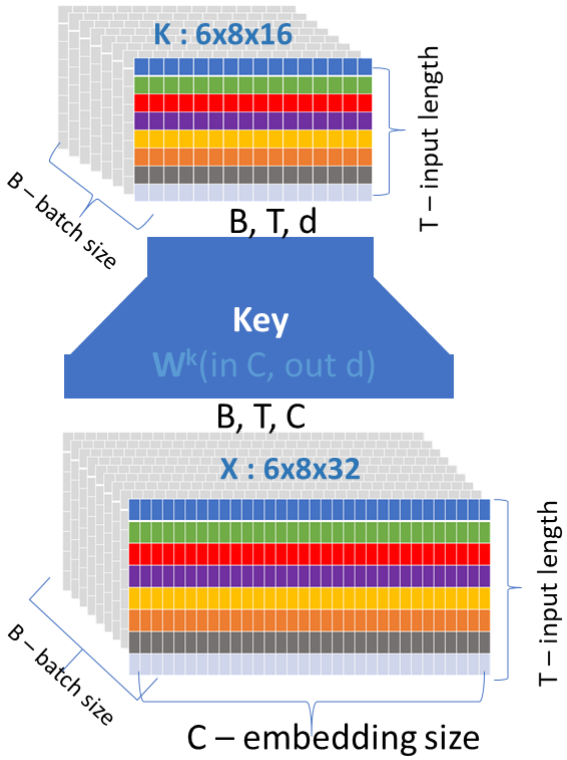
- Batch size $B = 6$
- Embedding size $C = 32$
- Input length $T = 8$
- Head size $d = 16$

Transforming Input into Key Vectors

The intuition behind this transformation is to project the input embeddings into a lower-dimensional space while preserving the relevant information. The Key vectors capture the "keys" or identifiers that help the model understand the relationships between different words in the sentence.

We start with the input embeddings of shape $(B, T, C) = (6, 8, 32)$. To transform these into Key vectors, we perform a linear transformation using a learned weight matrix W^K of shape $(C, d) = (32, 16)$ to reduce the dimensionality to d.

For each sentence in the batch, we perform the following computation:
$\text{Key (K)} = \text{Input embeddings} \times W^K$

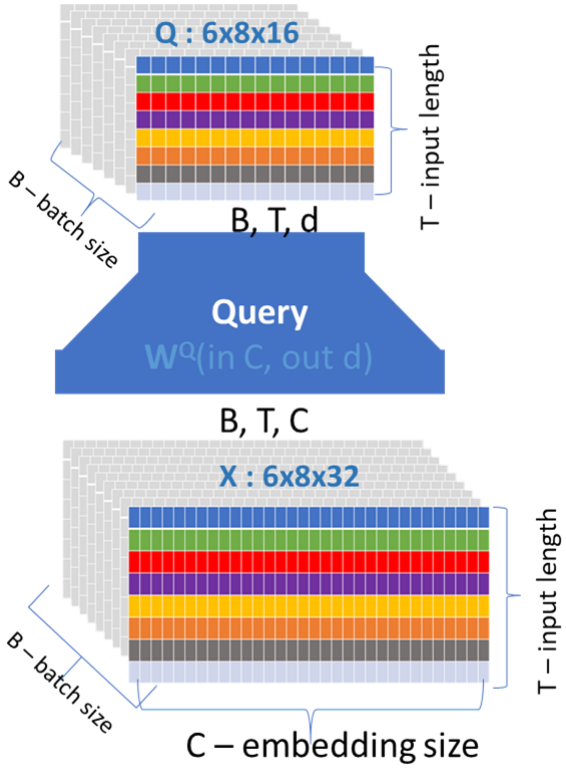


This operation results in Key vectors of shape $(B, T, d) = (6, 8, 16)$.

Transforming Input into Query Vectors:

Similarly, we transform the input embeddings into Query vectors. We perform another linear transformation using a learned weight matrix W^Q of shape $(C, d) = (32, 16)$ to maintain consistency in dimensionality.

$ \text{Query (Q)} = \text{Input embeddings} \times W^Q $



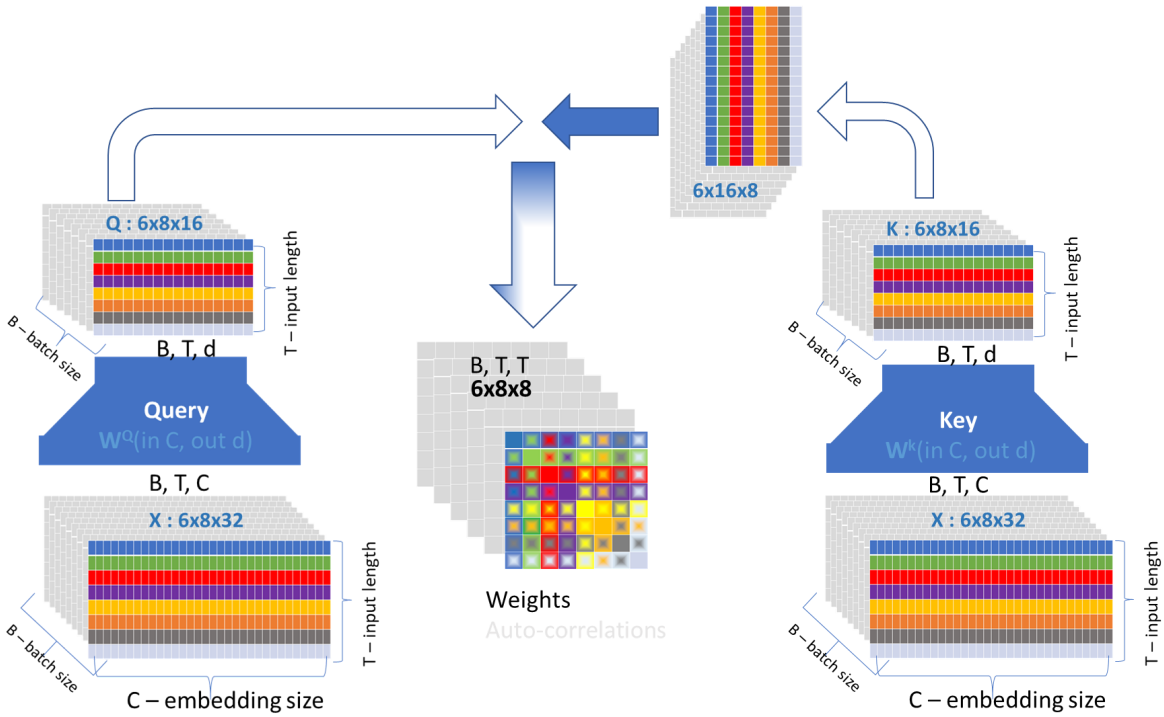
This operation results in Query vectors of shape $(B, T, d) = (6, 8, 16)$.

The Query vectors represent the "queries" or questions asked by the model to understand the relationships between different words in the sentence.

Calculation of Attention Scores:

To calculate the attention scores, we compute the dot product between the Key vectors and Query vectors. Since both Key and Query vectors have a shape of $(B, T, d) = (6, 8, 16)$, the dot product operation will result in a tensor of shape $(B, T, T) = (6, 8, 8)$.

$$\text{Attention Scores} = \text{Query} \times \text{Key}^T$$



These attention scores quantify the similarity or relevance between each word in the sentence and every other word. Higher scores indicate stronger relationships or dependencies between words.

Softmax and Attention Scores:

The attention scores obtained in the previous step are passed through the softmax function along the last dimension (dimension T) to obtain attention weights. This ensures that the weights sum up to 1 for each word in the sentence.

$$\text{Attention Weights} = \text{Softmax}(\text{Attention Scores})$$

The softmax function is applied element-wise to the normalized attention scores along the last dimension (dimension T). For each word in the sequence, the softmax function computes a probability distribution over all words in the sequence. This is achieved by exponentiating each value in the attention scores and dividing by the sum of all exponentiated values.

Mathematically, for a given sequence s of length T, the softmax function is defined as:

$$\text{Softmax}(s_i) = \frac{e^{s_i}}{\sum_{j=1}^{T} e^{s_j}}$$

Where s_i represents the normalized attention score for the i^{th} word in the sequence.

This operation introduces non-linearity and ensures that the attention weights are a valid probability distribution.

Normalizing the attention scores

Before applying Softmax, it is common to normalize the attention scores by dividing by the square root of the head-size dimensionality d of the Key and Query vectors.

$$\text{Attention}(Q_i, K_j) = \frac{Q_i \cdot K_j}{\sqrt{d}}$$

This normalization helps stabilize the training process and prevents the softmax function from saturating, especially when the dimensionality of the vectors is large.

Why this normalization step is necessary:

1. **Stability in Training:**
   When the dimensionality d of the Key and Query vectors is large, the dot products of these vectors can result in very large or very small values. This can lead to numerical instability during training, especially when applying the softmax function, which involves exponentiation of these dot products. Dividing the dot products by the square root of d helps scale down these values, making the computation more numerically stable.
2. **Effective Learning Rate:**
   Normalizing the attention scores by the square root of d effectively adjusts the learning rate for the attention mechanism. It ensures that the magnitude of the attention scores remains consistent across different values of d, allowing the model to learn meaningful attention distributions regardless of the dimensionality of the Key and Query vectors.
3. **Balancing Signal Magnitudes:**
   By normalizing the attention scores, we prevent any single dimension of the Key and Query vectors from dominating the attention calculation. This ensures that the attention mechanism considers all dimensions of the vectors equally, leading to a more balanced representation of the relationships between words in the sequence. A quick check of this point is shown in the following code where applying softmax on a logits vector (the attention

scores) can result in a well diffused probability distribution (line 3). At the same time for an arbitrarily scaled values of the logits vector will result in a pointy (almost like on-hot-encoded) probability distribution (line 4).

```
1  import torch
2  logits = torch.tensor([0.1,0.4,-0.9, 0.02, 0.35, -0.62])
3  print(torch.softmax(logits, dim=0).numpy().round(2)) # diffused
4  print(torch.softmax(logits*100, dim=0).numpy().round(2)) # One-hot-encoding
```

```
[0.18 0.25 0.07 0.17 0.24 0.09]
[0.   0.99 0.   0.   0.01 0.  ]
```

Therefore, a normalization factor $\sqrt{d}$ is typically applied after computing the dot products between the Key and Query vectors.

After applying the softmax function on the normalized attention scores, we obtain the attention weights, which represent the importance or relevance of each word in the sequence relative to all other words in the sequence.

The resulting attention weights represent how much attention or focus each word should place on other words in the sequence. Higher attention weights indicate higher importance or relevance of the corresponding words.

For instance, if the attention weight for a particular word is close to 1, it means that the model assigns high importance to that word when processing the current word in the sequence. Conversely, if the attention weight is close to 0, it means that the model assigns low importance to that word.

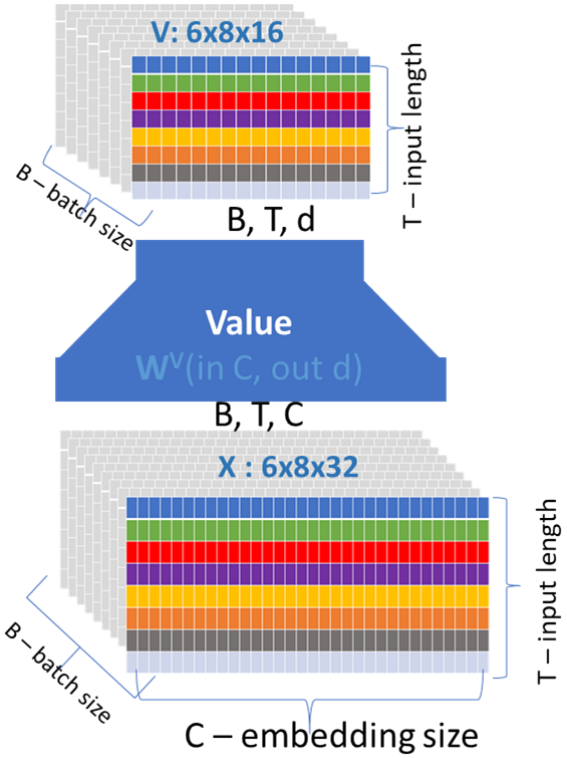
Transforming input into Value vectors

The Value vectors serve as the content-based representation of each word in the sequence, capturing the semantic meaning and significance that guide the attention mechanism in understanding the relationships and dependencies within the sequence. To compute the Value vectors, we perform a linear transformation on the input embeddings using a learned weight matrix W^V of shape $(C, d) = (32, 16)$. This transformation reduces the dimensionality of the input embeddings to match the head size d.

For each sentence in the batch, the input embeddings of shape $(B, T, C) = (6, 8, 32)$ are multiplied by the weight matrix W^V to obtain the Value vectors:

$$\text{Value (V)} = \text{Input embeddings} \times W^V$$

The resulting Value vectors have a shape of $(B, T, d) = (6, 8, 16)$, where d is the head size.



The Value vectors represent the information or content embedded in each word of the input sequence. Each Value vector encodes the semantic meaning or significance of the corresponding word.

Weighted Sum of Value Vectors:

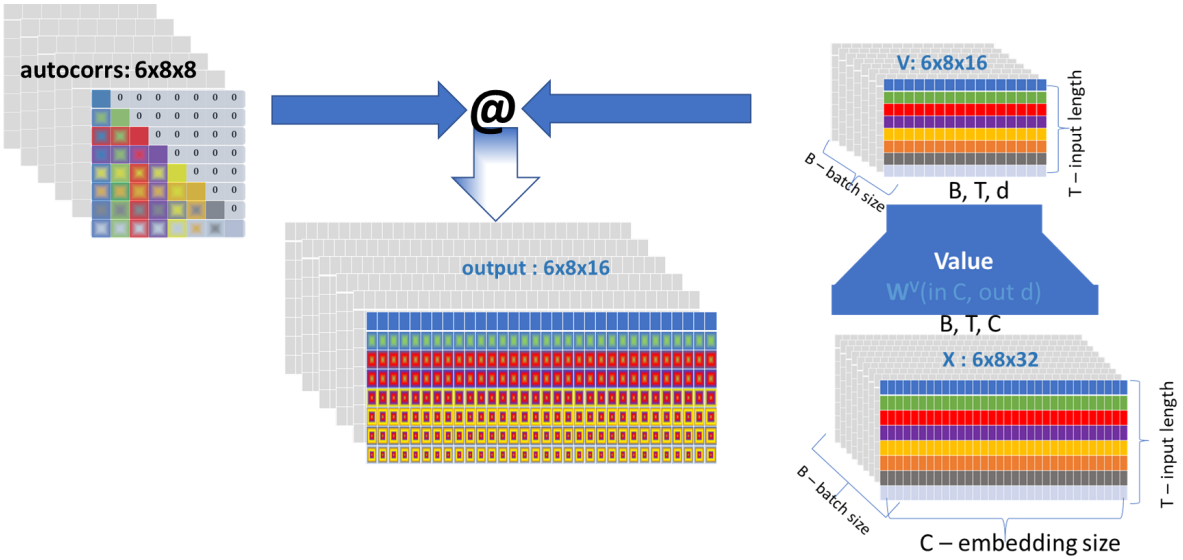
During the weighted sum computation in the self-attention mechanism, the attention weights determine how much importance each Value vector should receive. Higher attention weights indicate that the corresponding word is more relevant to the current word being processed, leading to a stronger influence on the final output.

Once we have obtained the attention weights, we use them to compute a weighted sum of the Value vectors. This weighted sum represents the final output of the self-attention mechanism for the current word in the sequence.

Mathematically, the weighted sum of Value vectors V is computed as follows:

$$\text{Output} = \sum_{j=1}^{T} \text{Attention\_Weights}_j \times V_j$$

Where $\text{Attention\_Weights}_j$ represents the attention weight assigned to the j^{th} word in the sequence, and V_j represents the Value vector corresponding to the j^{th} word.



Masking attention scores

In the Transformer architecture, particularly in the decoder part for tasks like sequence generation (e.g., language translation), it's crucial to prevent the decoder from attending to future tokens during the self-attention mechanism. This is because the decoder should generate tokens autoregressively, meaning it generates one token at a time, and it should only have access to the information from the tokens it has already generated.

To ensure this, a technique called "masking" is employed during the calculation of the weighted sum in the self-attention mechanism of the decoder. This masking involves modifying the attention scores before applying the softmax function, so that future tokens are effectively ignored during the attention calculation.

There are different types of masking that can be applied in the decoder:

1. **Look-Ahead Masking:**
   Look-ahead masking is used to prevent the decoder from attending to future tokens in the sequence. It involves masking out (setting to a very large negative value) the attention scores corresponding to future positions in the sequence before applying the softmax function. This ensures that the softmax operation effectively assigns zero probability to future tokens, preventing the decoder from attending to them.

   For example, if the model is generating the third token in the sequence, the attention scores corresponding to the fourth, fifth, and subsequent tokens are masked out.

2. **Padding Masking:**
   Padding masking is used to ignore padding tokens in the input sequences. In batched sequences, padding tokens are added to ensure that all sequences have the same length. By masking out the attention scores corresponding to padding tokens, the model effectively ignores them during the attention calculation, preventing them from influencing the output.

The reason for masking the attention scores in the decoder is to enforce causality and ensure that the model generates tokens based only on the information available up to the current position in the sequence. By preventing the decoder from attending to future tokens, the model learns to generate sequences autoregressively, one token at a time, in a left-to-right manner, similar to how humans generate sequences of text.

Next

After the self-attention mechanism produces the weighted sum of attention weights and Value vectors, the resulting output undergoes further processing before being passed to subsequent layers in the neural network.

1. **Processing the Weighted Sum:**
   The weighted sum of attention weights and Value vectors represents the output of the self-attention mechanism for each word in the sequence. This output captures the refined representation of the current word, considering its interactions with all other words in the sequence, weighted by their importance.
2. **Additional Processing:**
   Depending on the architecture of the neural network, additional processing steps may be applied to the output of the self-attention mechanism. These steps can include:
   - **Residual Connections:** The output of the self-attention mechanism may be combined with the original input embeddings using residual connections. This helps prevent the vanishing gradient problem and allows the network to learn more effectively.
   - **Normalization:** Normalization techniques such as Layer Normalization or Batch Normalization may be applied to the output to stabilize training and improve the convergence of the neural network.
   - **Non-linear Activation:** Activation functions such as ReLU (Rectified Linear Unit) or GELU (Gaussian Error Linear Unit) may be applied to introduce non-linearity into the network and enable it to learn complex patterns in the data.
3. **Feedforward Neural Network:**
   In many architectures, including Transformer-based models, the output of the self-attention mechanism undergoes further processing through a feedforward neural network (FFNN) layer. This FFNN layer typically consists of one or more fully connected layers followed by non-linear activation functions. The purpose of the FFNN layer is to perform additional transformations and feature extraction from the output of the self-attention mechanism.
4. **Output Representation:**
   Finally, the output of the feedforward neural network layer is used to generate predictions or representations relevant to the specific task at hand. For example, in a machine translation task, the output may be used to generate the translated sentence, while in a sentiment analysis task, the output may represent the sentiment of the input text.

Summary

The key benefit of self-attention is its ability to capture dependencies between different elements in the input sequence, allowing the model to weigh the importance of each element dynamically based on its context within the sequence. This makes self-attention particularly effective for capturing long-range dependencies in sequences, which is crucial for many NLP tasks. Attention mechanism answers three question related to each input word in a batch of words/sentence.

Who am I ?

What I am looking for ?

What value will it produce if I find what I am looking for ?

- Key

The Key vectors capture the "keys" or identifiers that help the model understand the relationships between different words in the sentence. You can think of a key as the answer to **"who am I?"**

- Query

  The Query vectors represent the "queries" or questions asked by the model to understand the relationships between different words in the sentence. You can think of a key as the answer to **"who I am looking for ?"**

- Attention scores:

  The attention scores represent the relevance/importance of each token to every other token in the sequence. Higher scores indicate stronger relationships or dependencies between tokens.

  - Normalization

    The normalization step is crucial for the stability and effectiveness of the self-attention mechanism, particularly in scenarios where the dimensionality of the vectors is high. It helps prevent issues such as vanishing or exploding gradients, making the training process more robust and efficient.

  - Softmax

    Applying softmax on normalized attention scores helps convert them into attention weights, which determine how much attention each word should pay to other words in the sequence during the weighted sum computation. These attention weights guide the model in capturing meaningful relationships and dependencies within the sequence.

- Value:

  The Value transformation capture the richness of information carried by each word in the sequence. They represent the content that the model should focus on when computing the self-attention mechanism. For example, in a machine translation task, the Value vectors might encode the semantic meaning of each word in the source language sentence, which is crucial for generating accurate translations. You can think of a key as the answer to **What value will it produce if I find what I am looking for ?**

- Weighted sum of Attention scores and Value:

  Applying softmax on normalized attention scores helps convert them into attention weights, which determine how much attention each word should pay to other words in the sequence during the weighted sum computation. These attention weights guide the model in capturing meaningful relationships and dependencies within the sequence.

  - Masking

    Masking the attention scores in the decoder is essential for training Transformer models for sequence generation tasks, ensuring that the model generates coherent and meaningful sequences based on the input information available up to the current position in the sequence.