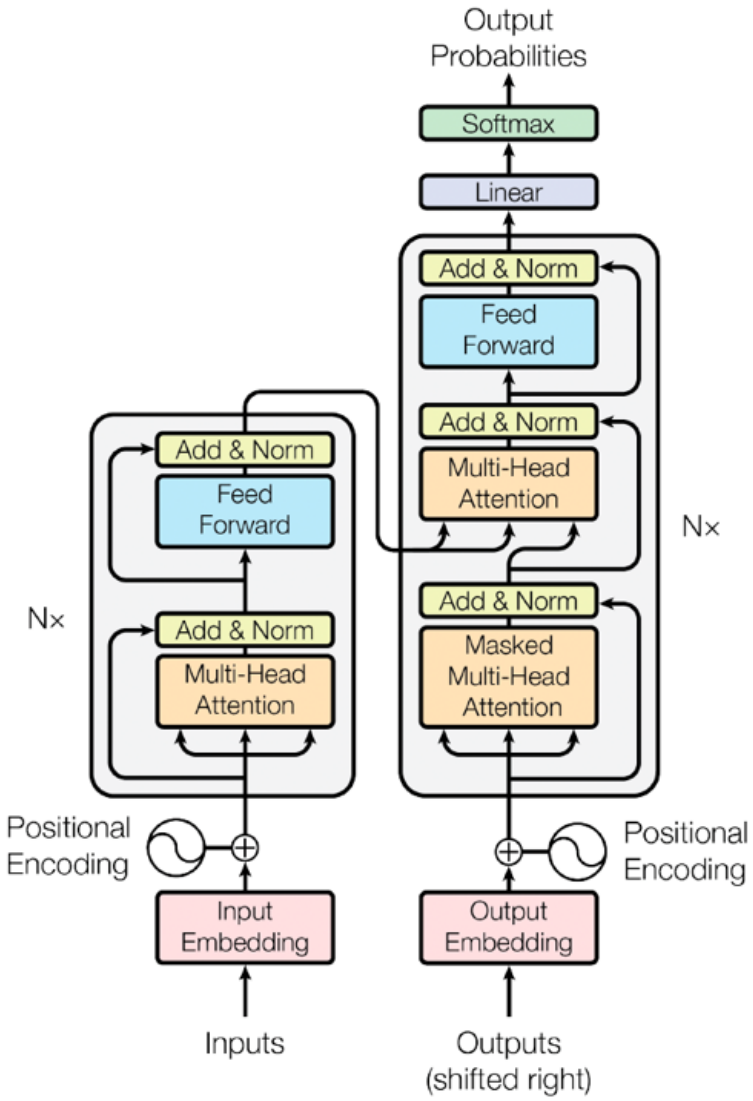
Graph Theory in Transformer Models

Cole Wendrowski

25 April 2024



Transformer model diagram [1].

Table of Contents

Introduction

Large language models are one of the most exciting and influential technologies of our day. Their rapid evolution is transforming industries. According to the World Economic Forum, this field may disrupt 40% of existing jobs [2]. Popular large language models, such as ChatGPT and LLaMa rely on a technology known as transformer models [3]. Transformers are regarded as one of the most influential advances in natural language processing [3]. Transformers were invented for text processing and now underpin many natural language processing systems today.

This whitepaper will explore the mechanics of transformer models and the use of graph theory in their construction and analysis.

Traditional Neural Networks vs. Transformers

Encoder / Decoder Architecture

Early applications of neural networks for natural language processing relied on a process known as an "encoder-decoder" model [1]. This process broke down into two stages.

1. Encoding
    a. An encoder processes an input data sequence and converts it into a mathematical model that captures the essence of the data. This includes the structure of and relationship between the data in the sequence. This process steps token by token through the data sequence.
2. Decoding
    a. A decoder then converts this mathematical model into the structure of the desired output.

A common use of an encoder-decoder model is language translation. First, an input in language A is fed into the model. The encoder converts this input into a mathematical model that captures the semantics of the input. A decoder then reconstructs this mathematical model in target language B. This translated output has the same underlying semantic meaning as the input from the original language.

This process works well for some natural language processing applications. However, the encoder-decoder model is limited in several ways. First, the encoding process requires the linear processing of tokens in the data sequence. This is not an issue for small data sizes. However, with large datasets, this linear processing can result in slower computation.

Additionally, because each token is examined sequentially, the encoder may be unable to capture relationships between distant tokens [1].

These limitations necessitated the creation of a new technology that could efficiently and correctly extract the meaning from input data.

Enter the transformer.

Transformers

To improve processing time and accuracy, transformer models use a mechanism called "self-attention" [1]. This allows the transformer to capture relationships between distant tokens. Additionally, the self-attention mechanism allows the transformer to process the input sequence in parallel. This dramatically reduces processing time. This improvement allows the transformer model to process larger datasets.

The pioneering 2017 research paper "Attention is All You Need" cited the transformer model's improved processing time and language translation ability when compared with its contemporary NLP peers [4].

How Does a Transformer Work?

Self-Attention

As introduced above, a key component of transformer models is their use of "self-attention". The self-attention mechanism allows the transformer to discern the importance of and relationship between each token. This enables the semantic understanding of tokens in the context of the rest of the input. Consider the two sentences "Bob sits on a chair" and "Bob is the philanthropy chair". The word "chair" carries different meanings in each sentence. This meaning is determined by the context supplied by the other words in the sentence. Self-attention enables the transformer to glean this context by grouping related tokens [1].

Graph theory is imperative for the application of self-attention. Tokens are represented as nodes. The relationships between tokens, known as weights, are represented as edges. This creates a connected weighted graph. There are two main types of attention models used in transformers, as shown in Figure 1 below [5].

1. **Full Attention** parses the relationships between every token in the input [5]. This creates a *complete* weighted graph. As the data set grows, this computation becomes more intensive. With each token added to the input, another node is added to the complete graph. Therefore, the degree of every node in the graph will increase in tandem as additional edges (attention weights) are added between this new token and the existing tokens. This enables the transformer to discern relationships between every token in the input.
2. **Local Attention** restricts attention to the neighbors of nodes. This is critical for larger datasets where inputs can include billions of nodes and complete graphs at this scale might be practically infeasible due to size [5].

Figure 1. Full attention relates every word in the input while local attention only relates neighbors [5].

These attention graphs can then be visualized as an attention matrix. In graph theory, this is known as a weighted adjacency matrix. Figure 2 below shows an equivalent connected weighted graph and weighted adjacency matrix.

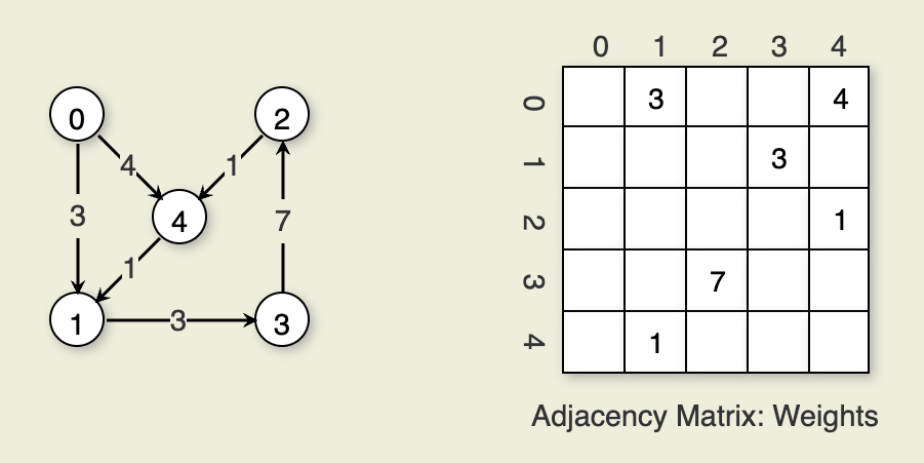
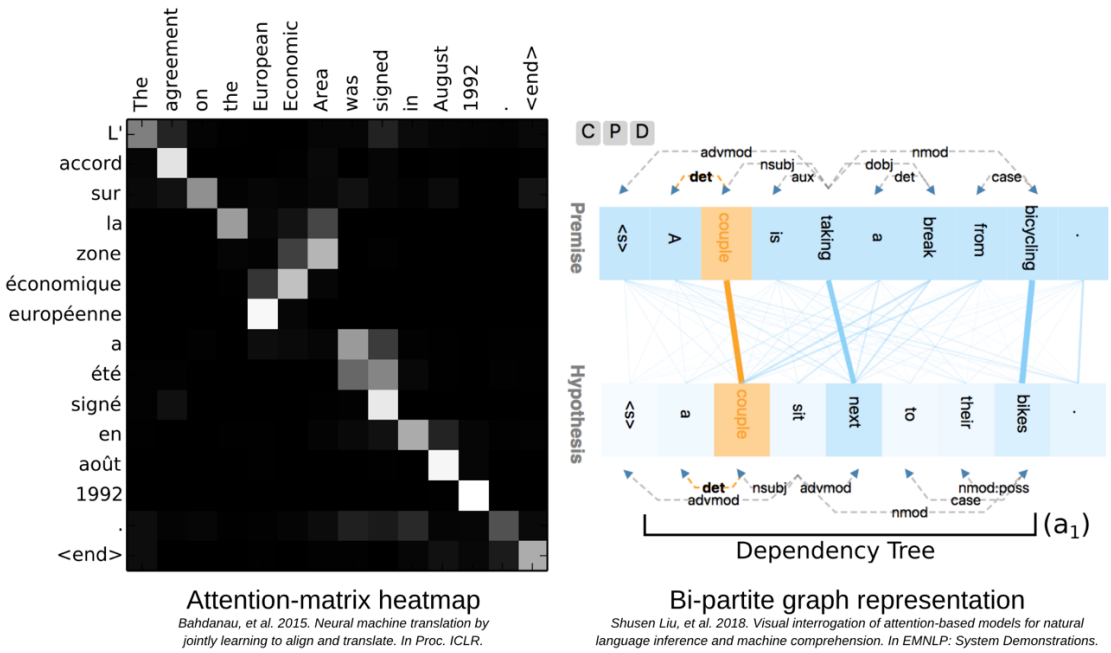


Figure 2. A connected weighted graph and a weighted adjacency matrix [6].

In transformer model analysis, the adjacency matrices can be visualized through the construction of an attention-matrix heatmap. Figure 3 below shows a heatmap of the weighting process when translating between French and English. The dark squares indicate minimal relation while the light squares demonstrate increased relation. The heatmap demonstrates that the model includes context from surrounding words rather than just mapping one-to-one [3]. Figure 3 also displays a bipartite graph. The transformer determines a semantic connection between the two sentences. This is demonstrated through edges connecting related words in each vertex set, the “premise” and the “hypothesis” [3].



Attention-matrix heatmap

Bahdanau, et al. 2015. Neural machine translation by jointly learning to align and translate. In Proc. ICLR.

Bi-partite graph representation

Shusen Liu, et al. 2018. Visual interrogation of attention-based models for natural language inference and machine comprehension. In EMNLP: System Demonstrations.

Figure 3. Visualizing self-attention weightings through a heatmap and a bi-partite representation [3].

Input Embeddings

First, the input data is translated into a series of tokens. For example, a sentence of text would be broken down such that each word is a token [1].

These new tokens are then transformed into vectors. The relationships in space between the vectorized tokens capture semantic meaning and relationships. These vectors can be quantified on many axes such as meaning, parts of speech, and other intrinsic characteristics [1].

Positional Encoding

Because the transformer processes input in a parallel fashion, rather than linear, the model does not initially encode the ordered relationship between tokens. To capture the positional relationships from the input data, the transformer adds position data to the embeddings of the tokens [1].

Transformer Block

The transformer block has two components [1]:

1. Multi-Head Self-Attention Mechanism
    a. This component implements the self-attention functionality described above.
2. Position-Wise Feed Forward Neural Network
    a. This neural network includes two functions:
        i. **Layer normalization** bounds the output values for each layer within a common range so that the output from each layer remains proportional [1]. This improves the training process.
        ii. **Linear transformation functions** allow for fine-tuning of the model to improve performance on specific tasks.

Linear and Softmax Blocks

This is the final stage of the transformer model. This stage converts the internal mathematical computations into the output. The linear block converts vectorized tokens back and generates a score for each called a "logit". These logits are then converted into a series of probabilities via a softmax function [1]. These probabilities represent the confidence in a token being the best output.

Demonstration

Code – Generating Attention Weights

Below is a Python project utilizing PyTorch and HuggingFace's BERT model to compute and visualize attention weights for input text. This program was developed with the assistance of ChatGPT.

```
import torch
from transformers import BertModel, BertTokenizer
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.ticker import FixedLocator
import networkx as nx

# generate attention weights for input text using BERT model
```

```
def get_attention_map(text, model_name='bert-base-uncased'):
    tokenizer = BertTokenizer.from_pretrained(model_name)
    model = BertModel.from_pretrained(model_name, output_attentions=True)
    inputs = tokenizer(text, return_tensors='pt')
    outputs = model(**inputs)
    attentions = outputs.attentions  # attention tensors for each layer
    return attentions, tokenizer

# display attention weights as heatmap
def plot_attention_map(attentions, tokenizer, text):
    # select first layer and first head attention map
    attention = attentions[0][0, 0].detach().numpy()

    # tokenize input text
    tokens = tokenizer.tokenize(text)
    tokens = ['[CLS]'] + tokens + ['[SEP]']

    # create heatmap
    fig, ax = plt.subplots(figsize=(10, 8))
    cax = ax.matshow(attention, cmap='bone')
    fig.colorbar(cax)

    # conigure graph
    ax.set_xticks(np.arange(len(tokens)))
    ax.set_yticks(np.arange(len(tokens)))
    ax.xaxis.set_major_locator(FixedLocator(np.arange(len(tokens))))
    ax.yaxis.set_major_locator(FixedLocator(np.arange(len(tokens))))
    ax.set_xticklabels(tokens, rotation=90, fontsize=10)
    ax.set_yticklabels(tokens, fontsize=10)

    # display the graph
    plt.show()

# display attention weights as connected graph
def plot_graph_from_attention(attentions, tokenizer, text):
    # select first layer and first head attention map
    attention = attentions[0][0, 0].detach().numpy()

    # tokenize input text
    tokens = tokenizer.tokenize(text)
    tokens = ['[CLS]'] + tokens + ['[SEP]']

    # create connected graph from the attention matrix
    G = nx.DiGraph()

    # add nodes
    for i, token in enumerate(tokens):
        G.add_node(i, label=token)
```

```
    # add edges with the attention weights
    for i in range(len(tokens)):
        for j in range(len(tokens)):
            if attention[i, j] > 0.01: # ensure that edges are significant
                G.add_edge(i, j, weight=attention[i, j])

    # display the graph
    pos = nx.spring_layout(G)  # positions for nodes
    nx.draw(G, pos, with_labels=True, labels=nx.get_node_attributes(G, 'label'))
    edge_labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
    plt.show()

#input text to analyze relation between tokens
text = "bob is the philanthropy chair"
attentions, tokenizer = get_attention_map(text)

#display attention matrix as a heat map
#this shows the weights between tokens
plot_attention_map(attentions, tokenizer, text)

#plot directed graph showing weights (edges) between tokens (vertices)
plot_graph_from_attention(attentions, tokenizer, text)
```

Graphs – Visualizing Self-Attention

This program generates the following graphs:

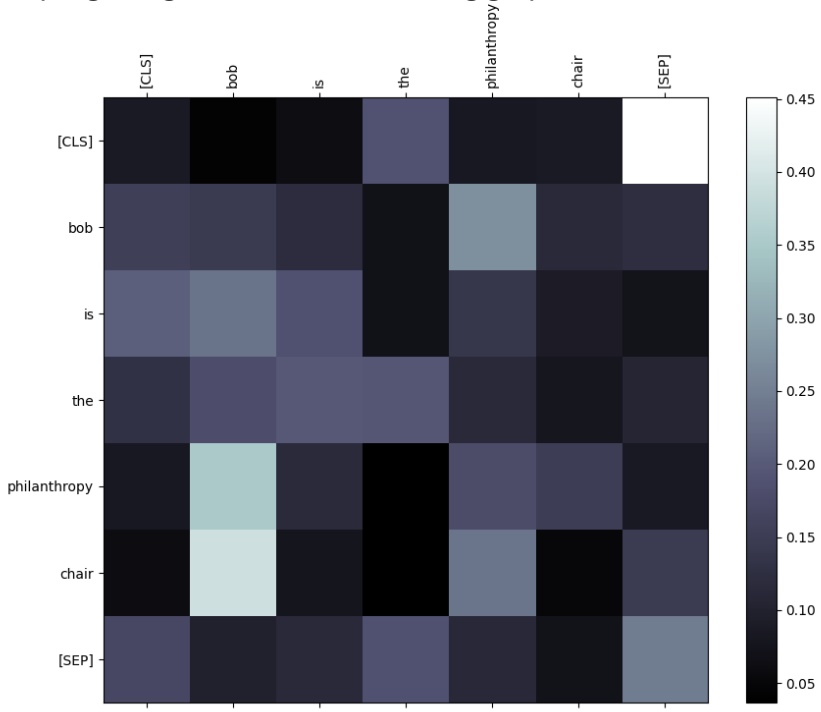


Figure 4. An attention-matrix heatmap for input “bob is the philanthropy chair”.

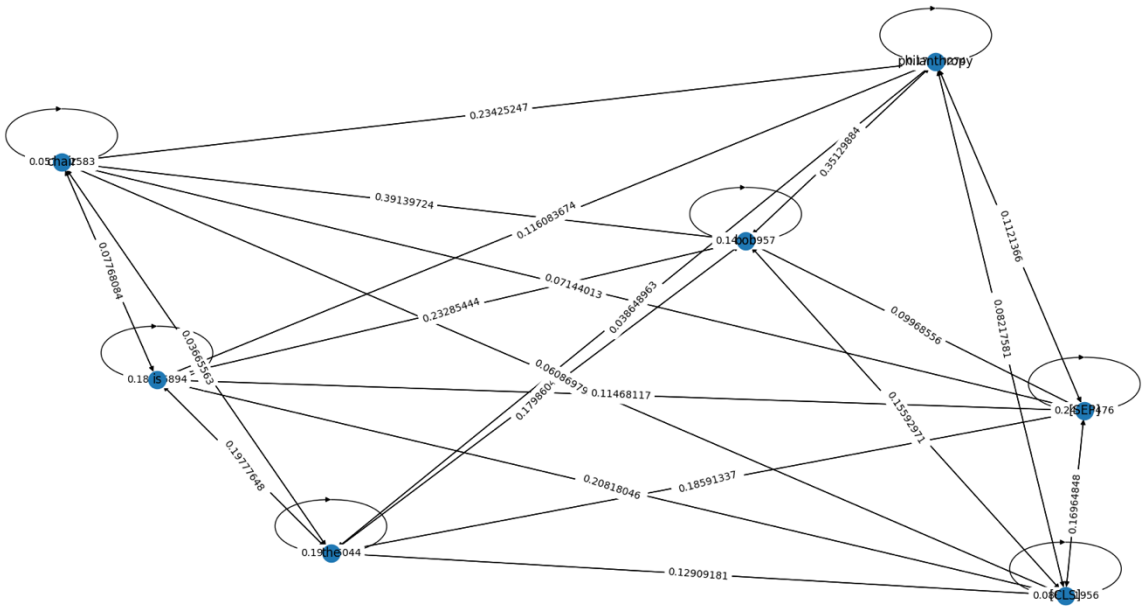


Figure 5. A complete graph of attention weights for input “bob is the philanthropy chair”.

Figure 4 displays the attention weightings between tokens in the input. Lighter colors indicate a greater relation while darker colors represent a minimal relation. “bob” and “philanthropy” are strongly weighted, as are “bob” and “chair”. Figure 5 displays the attention map as a weighted complete graph. Each vertex represents a token (input word) while edges represent the attention weight between each token generated by the transformer model.

Conclusion

Large language models rely on a technology known as transformer models. Transformer models efficiently and accurately process input data via “self-attention”. Self-attention allows the transformer to process data in parallel, speeding up computations and improving recognition of interactions between distant tokens. The transformer performs full attention or local attention depending on the size of the dataset. This creates a complete or connected graph respectively such that tokens are represented as nodes and weights are represented as edges. The attention weights can be visualized through a weighted adjacency matrix known as an attention-matrix heatmap. Transformer models consist of several stages – input embedding; positional encoding; and transformer, linear, and softmax blocks. These stages vectorize the tokens, embed location data, and convert the mathematical representations of the input to probabilistic output.

References

[1] "What Are Transformers In Artificial Intelligence?," AWS, [Online]. Available: https://aws.amazon.com/what-is/transformers-in-artificial-intelligence/. [Accessed 24 April 2024].

[2] D. Elliott, "AI: 3 important considerations for leaders as large language models transform business," World Economic Forum, 8 January 2024. [Online]. Available: https://www.weforum.org/agenda/2024/01/large-language-models-future-jobs/.

[3] "Explainable AI: Visualizing Attention in Transformers," Comet, 16 July 2023. [Online]. Available: https://www.comet.com/site/blog/explainable-ai-for-transformers/. [Accessed 24 April 2024].

[4] "Attention Is All You Need," Cornell University, 12 June 2017. [Online]. Available: https://arxiv.org/abs/1706.03762. [Accessed 24 April 2024].

[5] V. P. Dwivedi, "Graph Transformer: A Generalization of Transformers to Graphs," Towards Data Science | Medium, 4 March 2021. [Online]. Available: https://towardsdatascience.com/graph-transformer-generalization-of-transformers-to-graphs-ead2448cff8b.

[6] "19.1. Graphs Chapter Introduction," OpenDSA Data Structures and Algorithms Modules Collection, 1 October 2023. [Online]. Available: https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/GraphIntro.html.