Demystifying Transformers: A Deep Dive

1kg · Follow
7 min read · Jan 6, 2024

74

Transformers have become one of the most influential innovations in deep learning in recent years, revolutionising natural language processing and computer vision. But how exactly do they work under the hood? In this comprehensive deep dive, we'll go beyond the basics and unpack the key concepts underpinning transformers and transformer-based models.

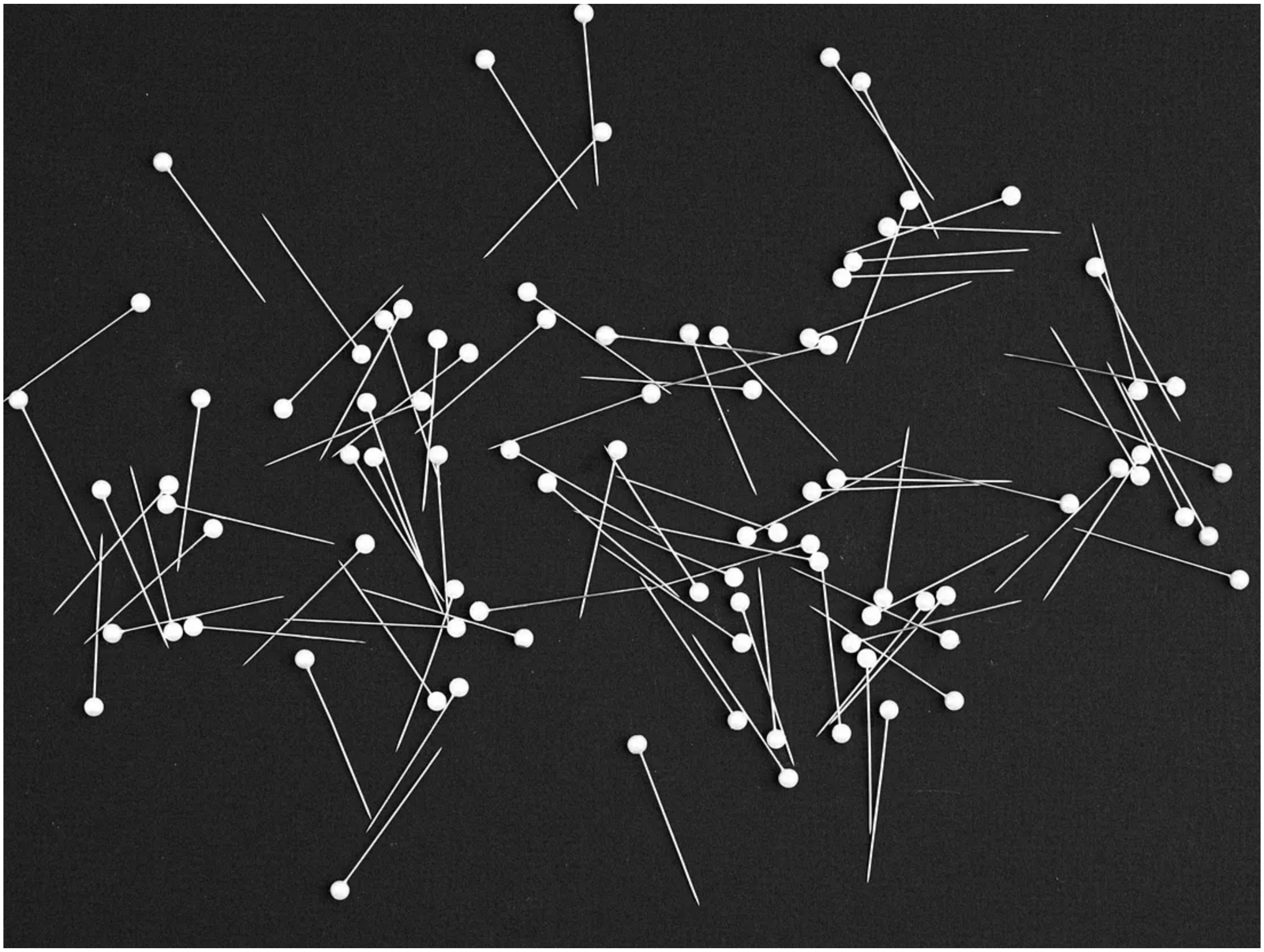


Photo by Kier in Sight Archives on Unsplash

In this deep dive for the AI-curious, we'll unravel the mystery behind transformers through an intuitive explainer. You'll learn:

- How transformers brought about a revolution in deep learning
- The key innovations that set transformers apart from previous models
- How attention allows transformers to "focus" on relevant information
- The core components that enable transformers to handle long sequences

- What dot-product attention is and how it works
- How transformers can be implemented in just over 100 lines of code

Let’s get started!

Transformers 101

First, a quick refresher on what exactly transformers are. Transformers are a novel neural network architecture particularly well-suited for processing sequential data. They were first introduced in 2017 in the paper “Attention is All You Need” and consist of an encoder and a decoder.

The encoder reads in the input sequence and generates an internal representation of it. This condensed vector encapsulates information about the entire sequence.

The decoder then uses the encoder’s output to generate the target sequence, one element at a time. This is done autoregressively — each output is fed back into the model to predict the next element.

This encoder-decoder structure makes transformers shine for tasks like translation, where the goal is to map between sequences. They can learn to encode the input sequence into a dense vector capturing its semantics, and decode that vector into the target language.

But what makes transformers so different from what came before?

The Limitations of RNNs

Before transformers, deep learning models relied heavily on structures called recurrent neural networks (RNNs) for processing language and other sequence data.

RNNs work by passing information sequentially through a chain of operations. Each step in the sequence handles one element, maintains an internal “state”, and passes it on to process the next element.

But RNNs have a critical flaw: long-range dependencies. They struggle to connect information that is far apart in a sequence, since the internal state can only retain limited information.

Imagine trying to read a book this way — you could only keep track of the last few sentences at a time before forgetting the beginning. Making coherent sense of what you’re reading becomes difficult!

This makes RNNs ineffective for long sequences, constrained by their fixed processing order and bottlenecks.

So what breakthrough was needed to move past these limitations?

The Attention Revolution: Looking Where It Matters

The key innovation enabling transformers is a mechanism called attention.

Recurrent neural networks like LSTMs were previously the dominant approach for sequential data. They process inputs one at a time, maintaining an internal state that gets updated sequentially.

Attention throws this out the window — rather than forcing sequential processing, attention allows models to flexibly incorporate contextual information from the entire input. This resolves RNNs' struggle with long-range dependencies.

With attention, each position in a sequence is able to directly "attend" to every other position, selecting the most relevant pieces of context to focus on. This adds a huge boost to how much contextual information can flow through the model.

There are two types of attention in transformers:

- Self-attention in the encoder lets each position in the input attend to every other position. This allows important contextual information to flow between distant parts of the sequence.
- Encoder-decoder attention in the decoder allows each output position to attend over all positions in the input sequence. This lets the model focus on the relevant input context as it generates each token.

Attention works by calculating compatibility scores between each pair of positions (query and key). The scores determine how much each value vector is weighted in the output — allowing it to amplify or ignore different parts of the input.

But how does attention work under the hood?

Dot-Product Attention: A Neural Spotlight

The standard attention used by transformers is scaled dot-product attention.

You can imagine attention as a spotlight operators can control. For a given position, attention highlights the parts of a sequence that are most relevant and dims the rest.

Dot-product attention accomplishes this by calculating compatibility scores between each pair of positions (the query position and each key position).

These scores determine how much each value vector from the sequence gets weighted in the output. This acts like a spotlight intensity control — amplifying certain parts and downplaying others.

The compatibility scores are based on multiplying query and key vector representations together. Positions with highly compatible query and key vectors will have large scores, directing the spotlight towards them.

This allows each position to retrieve precisely the relevant context it needs from the entire sequence, providing much more flexibility than RNNs.

Here’s how it works:

1. The input embeddings are projected into queries, keys, and values using learned weight matrices.
2. The dot product of the query with each key is calculated. This determines the compatibility between each query-key pair.
3. The scores are divided by sqrt(dim) to normalize them before the softmax.
4. A softmax converts the scores into probabilities representing the attention weights.
5. The attention weights are multiplied by the values to produce the weighted average forming the output.

This allows the model to retrieval relevant information from the entire sequence for each position. The matrices can be seen as lookups into an associative memory.

Multi-Headed Attention: Parallel Spotlights

Dot-product attention forms the foundation, but transformers use an extended version: multi-headed attention.

This projects the input into multiple “heads” and runs separate dot-product attentions in parallel.

You can imagine this as controlling several spotlights at once, each retrieving different types of information. This allows the model to jointly attend based on different representations.

The independent heads are then concatenated together, merging their extracted context. This multi-headed attention allows the model to combine different attention perspectives.

Thinking of attention as spotlights that can be controlled dynamically helps build an intuition for why it’s so powerful in transformers.

Feedforward and Positional Encodings

In addition to attention, transformers incorporate other key elements:

- Feedforward layers after attention allow further processing and transformation of representations. They expand the dimensionality and then project back down.

- Residual connections let gradients flow smoothly during training. The output is added to the input before flowing to the next layer.

- Layer normalization keeps values stable as they pass through stacked layers.

- Positional encodings provide order information to the otherwise permutation invariant self-attention. Fixed sinusoidal encodings are commonly used.

Without these supporting components, attention alone would not be enough. Together they enable transformers to be trained to handle extremely long, high-dimensional sequences.

Transformers In 100 Lines of Code

While modern transformer models have grown massively complex, the concepts can be implemented in a remarkably compact code:

```
import torch
import torch.nn as nn

class Transformer(nn.Module):
    def __init__(self, embedding_size, num_heads):
        super().__init__()

        # Input embedding
        self.token_embedding = nn.Embedding(vocab_size, embedding_size)
        self.position_embedding = nn.Embedding(max_len, embedding_size)

        # Encoder
        self.attention = MultiHeadedAttention(num_heads, embedding_size)
        self.feedforward = FeedForward(embedding_size)

        # Decoder
        self.output = nn.Linear(embedding_size, vocab_size)
    def forward(self, x):

        # Embed input tokens and positions
        x = self.token_embedding(x)
        pos = torch.arange(x.shape[1], device=x.device)
        x += self.position_embedding(pos)

        # Apply encoder
        x = self.attention(x)
        x = self.feedforward(x)

        # Generate output prediction
        x = self.output(x)
        return x

class MultiHeadedAttention(nn.Module):

    def __init__(self, num_heads, embedding_size):
        super().__init__()

        self.project_q = nn.Linear(embedding_size, embedding_size)
        self.project_k = nn.Linear(embedding_size, embedding_size)
        self.project_v = nn.Linear(embedding_size, embedding_size)

        self.attention = ScaledDotProductAttention()
        self.normalize = nn.LayerNorm(embedding_size)

    def forward(self, x):

        # Project input into multiple heads
        q, k, v = self.project(x)

        # Apply attention
        out = self.attention(q, k, v)

        # Concat and normalize
        out = self.normalize(out)
        return out

class ScaledDotProductAttention(nn.Module):

    def forward(self, q, k, v):

        # Calculate scaled scores
        scores = torch.matmul(q, k.transpose(-2, -1)) / np.sqrt(k.shape[-1])

        # Apply softmax
        scores = F.softmax(scores, dim=-1)
        # Multiply with value vectors
        out = torch.matmul(scores, v)
        return out

class FeedForward(nn.Module):

    def __init__(self, embedding_size):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(embedding_size, embedding_size * 4),
            nn.ReLU(),
            nn.Linear(embedding_size * 4, embedding_size)
        )

    def forward(self, x):
        return self.net(x)
```

While modern variants have hundreds of millions of parameters, this minimal implementation captures the transformer's essence. Much complexity is added to improve performance, but the core concepts remain the same.

Transformers In Practice

The transformer architecture provides an extendable framework for sequence modeling. This has fueled breakthrough advances across language, vision, speech, and beyond:

- BERT — Bidirectional pretraining resulted in huge gains across NLP tasks.

- GPT-3 — Massive scaling demonstrated the potential of transformer language models.

- DALL-E — Transformers enabled generating realistic images from text captions.

And this is just the beginning. With greater scale and innovations in pretraining, transformers are poised to power further revolutionary progress in AI.

The Road Ahead

In this deep dive, we've demystified the transformers behind the recent wave of AI advancement. Starting from RNN limitations, we explored how the flexibility of attention resolves sequence modeling constraints.

We dove into dot-product attention, residual connections, and multi-headed designs. And saw how transformers can be implemented compactly.

Transformers introduce a paradigm shift — eschewing fixed processing for dynamic models that focus on what matters. This innovation catalyzed a revolution across applications involving language, vision, speech, and beyond.

But the journey is far from over. With rapid progress in model scale and pretraining techniques, transformers remain a wellspring of innovation. Who knows what creative applications could blossom next?

The future looks bright as transformers continue lighting the way towards more powerful and capable AI.

Transformers Artificial Intelligence Generative Ai Gpt Data Science

Written by 1kg

543 Followers

Follow

More from 1kg

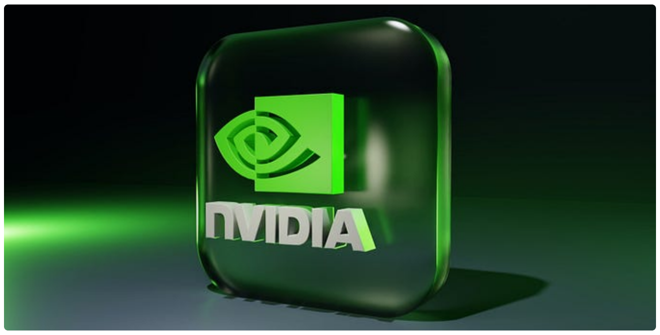
1kg

Heist of Salim Kara: A Story Beyond Belief

Have you ever wondered how someone could steal over $2 million in coins right under the...

3 min read · Jan 3, 2024

50



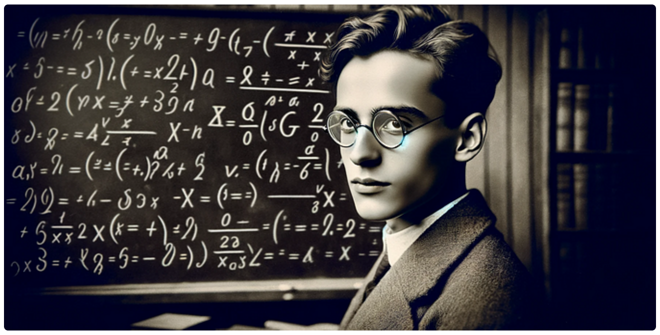
1kg

Nvidia's CUDA Monopoly

A Deep Dive Analysis

23 min read · Aug 8, 2023

38 2



1kg

Gödel's Incompleteness Theorems Explained for Everyone

In the world of mathematics, few discoveries have sparked as much intrigue and wonder ...

3 min read · Nov 24, 2023

476 3

1kg

CUDA vs ROCm: The Ongoing Battle for GPU Computing...

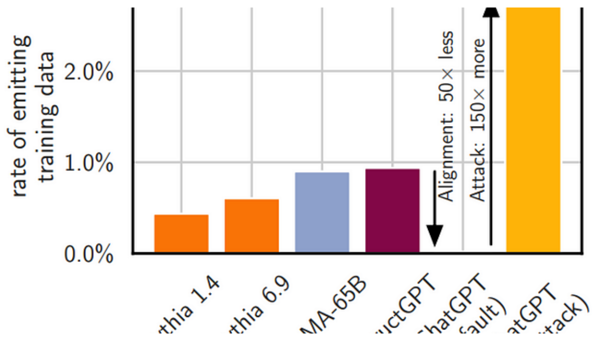
GPU computing has become indispensable to modern artificial intelligence. The vast parall...

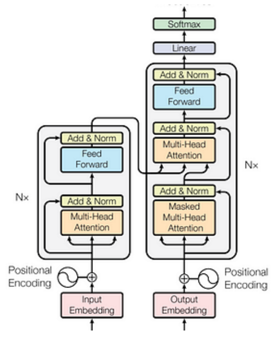
3 min read · Jan 20, 2024

52

See all from 1kg

Recommended from Medium





Devansh in DataDrivenInvestor

Google extracted ChatGPT's Training Data using a silly trick.

Scalable Extraction of Training Data from (Production) Language Models

13 min read · Jan 8, 2024

2.2K 15

Hrithick Sen

The Math Behind the Machine: A Deep Dive into the Transformer...

The transformer architecture was introduced in the paper "Attention is All You Need" by...

12 min read · Jan 16, 2024

188 5

Lists

Predictive Modeling w/ Python

20 stories · 932 saves

AI Regulation

6 stories · 326 saves

Natural Language Processing

ChatGPT prompts

Open in app ↗

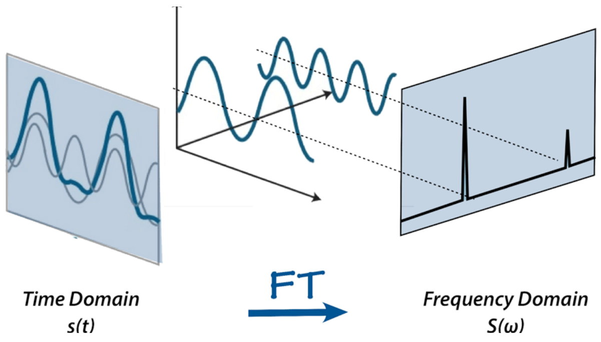
Search Write L

Abdulkader Helwan

Liquid Neural Networks

Neural networks are powerful machine learning models that can learn from data an...

✦ · 6 min read · Sep 3, 2023



Everton Gomede, PhD in The Modern Scientist

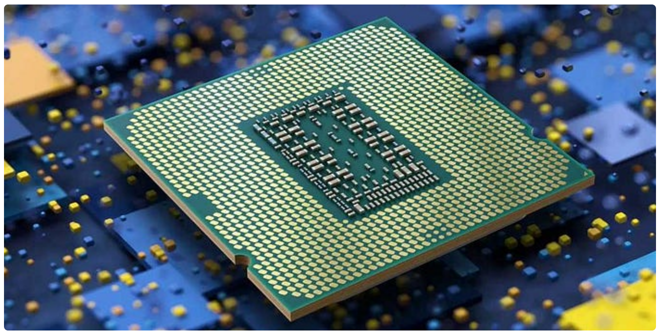
The Fourier Transform and its Application in Machine Learning

Introduction

5 min read · Nov 3, 2023

115 3

1.7K 11



Ayush Thakur in Stackademic

CPU vs TPU vs GPU vs DPU: What the Hell 🤔

Hey there tech enthusiasts! Ever wondered about those acronyms flying around in the...

4 min read · Dec 30, 2023

148 4

James Presbitero Jr. in Practice in Public

These Words Make it Obvious That Your Text is Written By AI

These 7 words are painfully obvious. They make me cringe. They will make your reader...

5 min read · Jan 1, 2024

40K 1055

See more recommendations