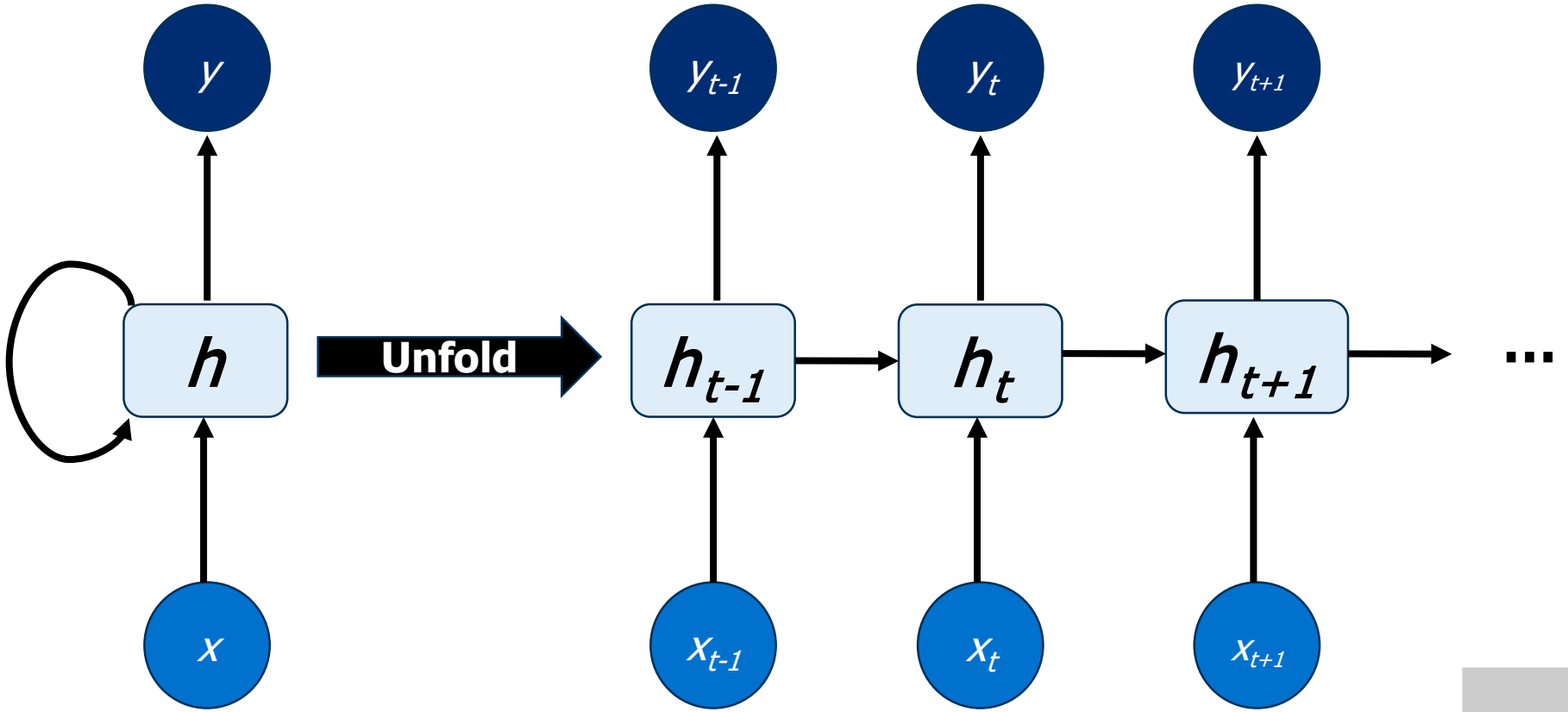
Transformers for LLMs

RNNs and LSTMs to Transformers

Recurrent Neural Network (RNN)

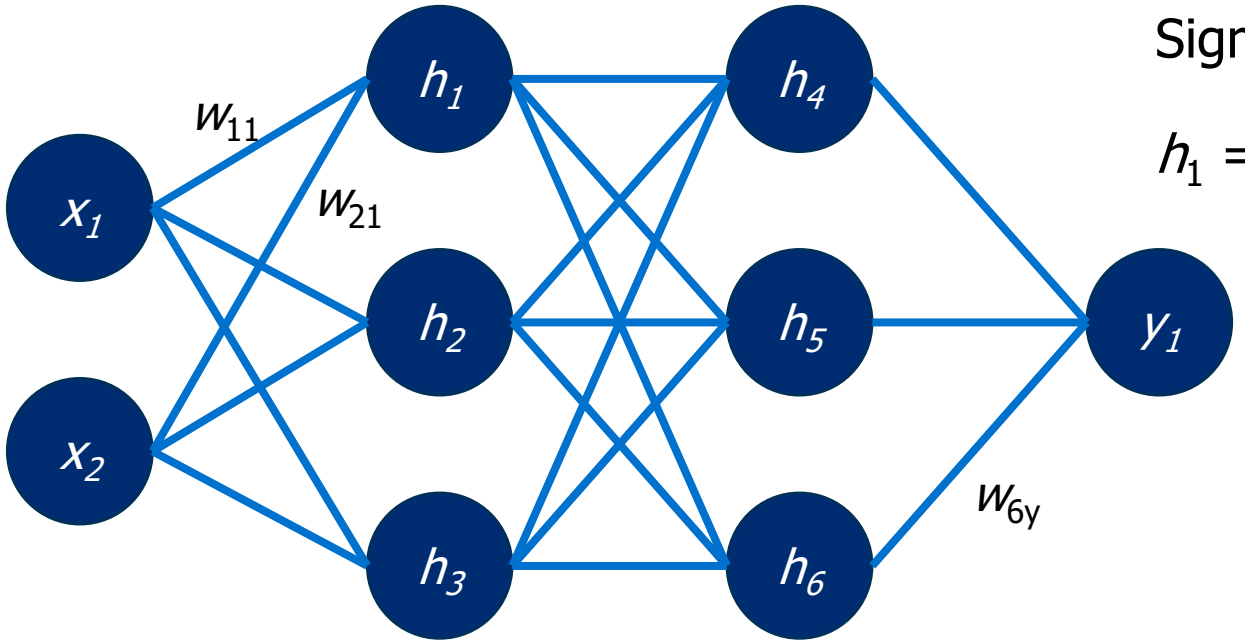


vanishing or exploding gradient

x: input *h:* hidden memory *y:* output

Vanishing/Exploding Gradient Problem

- Assume a neural network with two hidden layers…



Sigmoid function, $f(x) = 1 / (1 + e^{-x})$

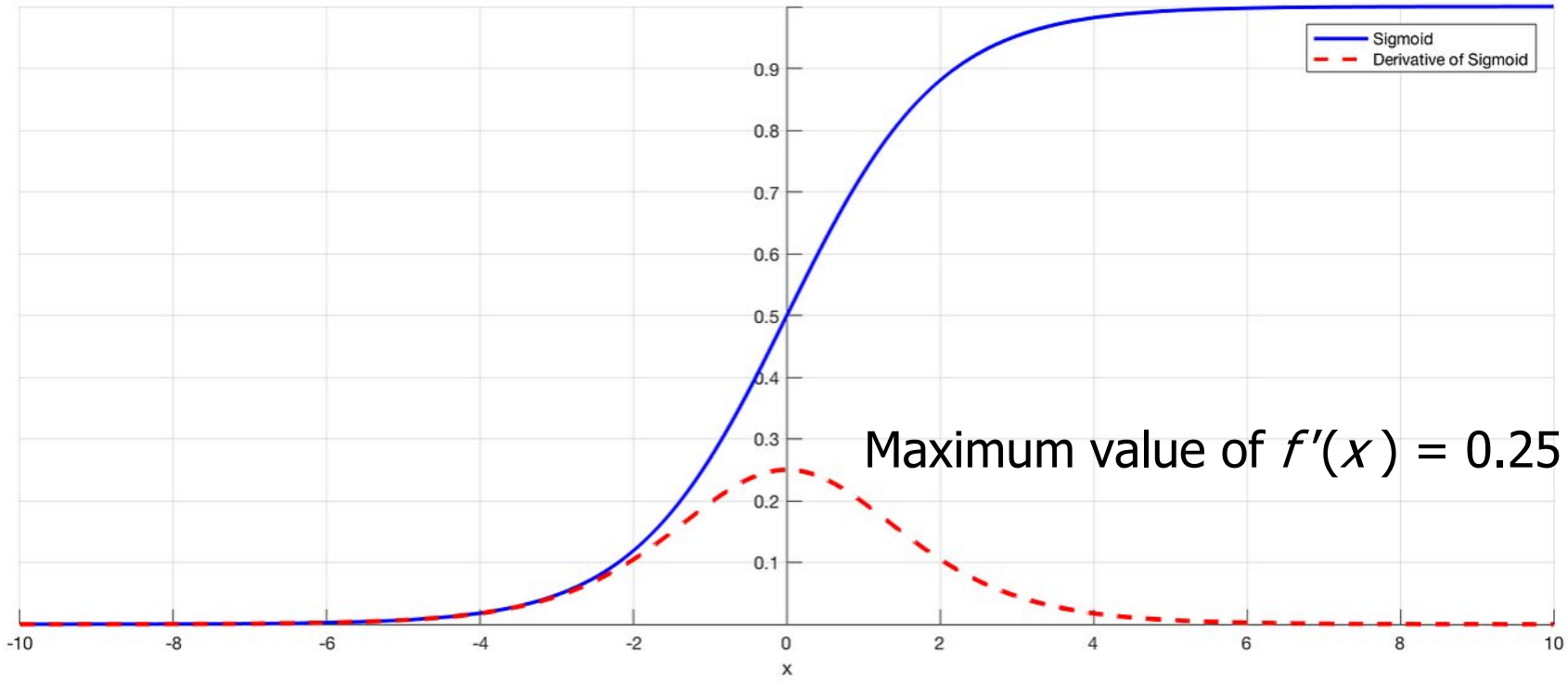
$h_1 = w_{11}*x_1 + w_{21}*x_2$ $\quad\quad$ $f(h_1)$…

Loss function, $L = (y_1 - \text{target})^2$

$$\frac{\partial L}{\partial w_{6y}} = \frac{\partial L}{\partial y_{out}} \frac{\partial y_{out}}{\partial y_{in}} \frac{\partial y_{in}}{\partial w_{6y}}$$

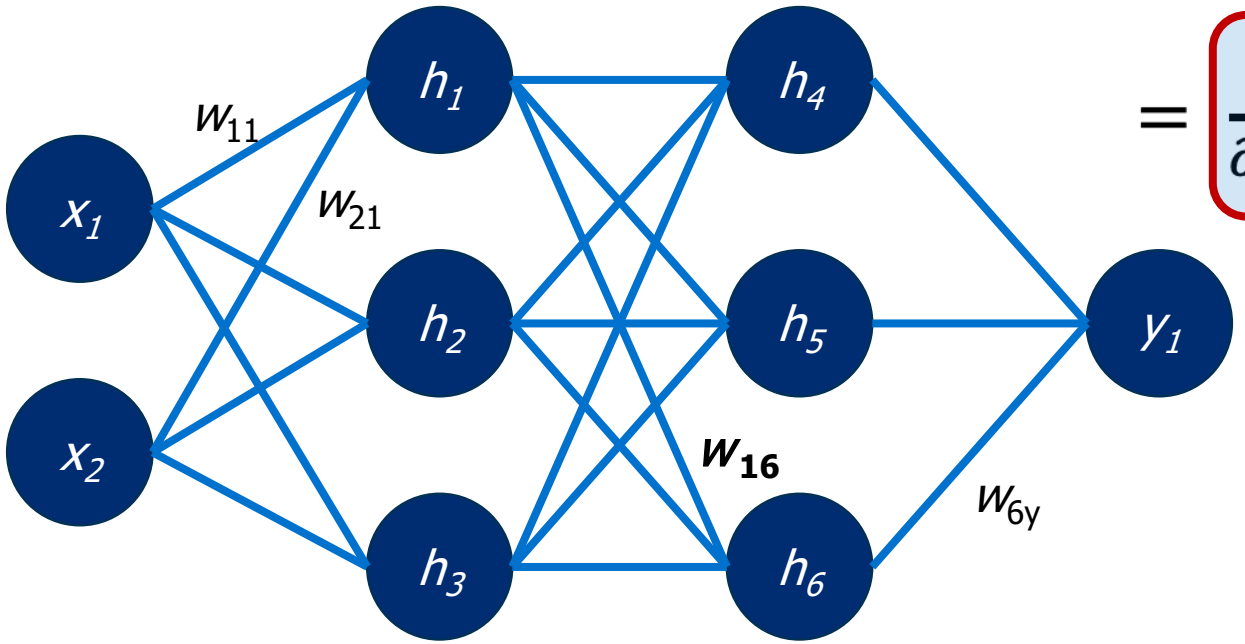
$f(x)$ $\quad$ h_6

Vanishing/Exploding Gradient Problem



Maximum value of $f'(x) = 0.25$

Vanishing/Exploding Gradient Problem



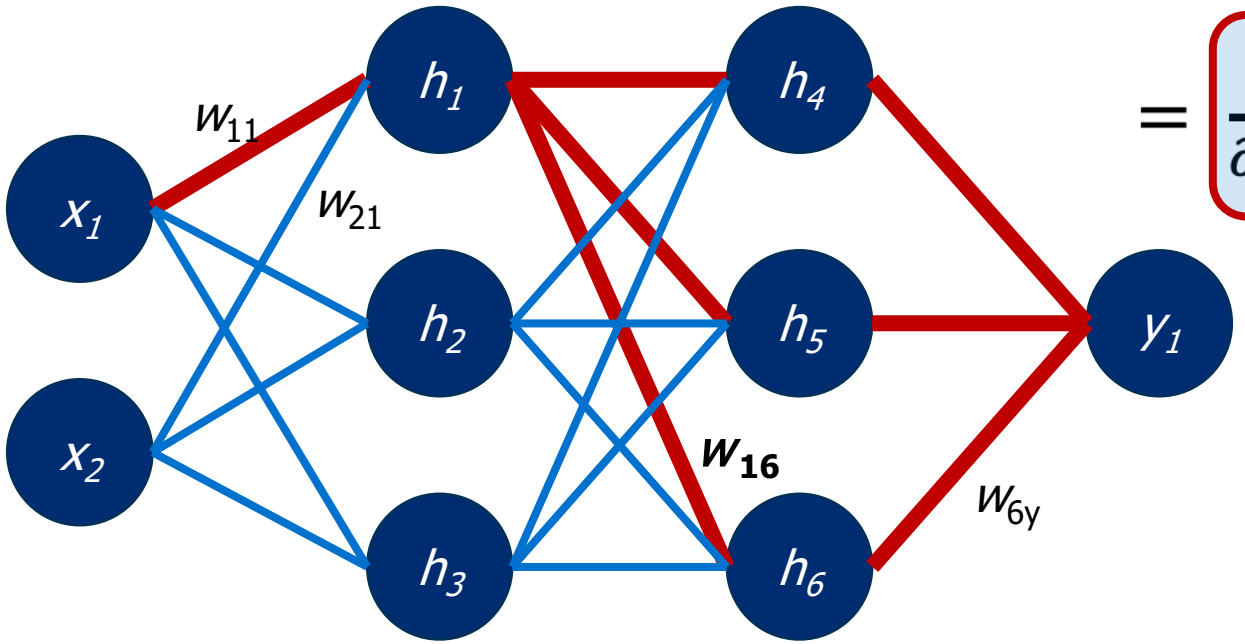
$$\frac{\partial L}{\partial w_{16}} = \frac{\partial L}{\partial h_{6(\text{out})}} \frac{\partial h_{6(\text{out})}}{\partial h_{6(\text{in})}} \frac{\partial h_{6(\text{in})}}{\partial w_{16}}$$

$$= \frac{\partial L}{\partial y_{\text{out}}} \frac{\partial y_{\text{out}}}{\partial y_{in}} \frac{\partial y_{in}}{\partial h_{6(\text{out})}} \frac{\partial h_{6(\text{out})}}{\partial h_{6(\text{in})}} \frac{\partial h_{6(\text{in})}}{\partial w_{16}}$$

w_{6y} $f(x)$

0.25 * 0.25 = 0.0625

Vanishing/Exploding Gradient Problem



$$\frac{\partial L}{\partial w_{16}} = \frac{\partial L}{\partial h_{6(\text{out})}} \frac{\partial h_{6(\text{out})}}{\partial h_{6(\text{in})}} \frac{\partial h_{6(\text{in})}}{\partial w_{16}}$$

$$= \underbrace{\frac{\partial L}{\partial y_{\text{out}}} \frac{\partial y_{\text{out}}}{\partial y_{in}}}_{} \underbrace{\frac{\partial y_{in}}{\partial h_{6(\text{out})}}}_{w_{6y}} \underbrace{\frac{\partial h_{6(\text{out})}}{\partial h_{6(\text{in})}}}_{f(x)} \frac{\partial h_{6(\text{in})}}{\partial w_{16}}$$

0.25 * 0.25 = 0.0625

Impact of w_{11} on loss function is very, very small

Vanishing/Exploding Gradient Problem

The vase is fragile.
Don’t move the vase to the edge of the table.

The vase is fragile.
Don’t move the vase to the edge of the table.

Long Short Term Memory (LSTM)

- Different paths for long and short term memory
- A sigmoid function (range 0 to 1) determines % of LTM remembered
- A tanh activation function (range -1 to 1) determines % STM added
- Create a new short-term memory using the sigmoid and tanh fns.

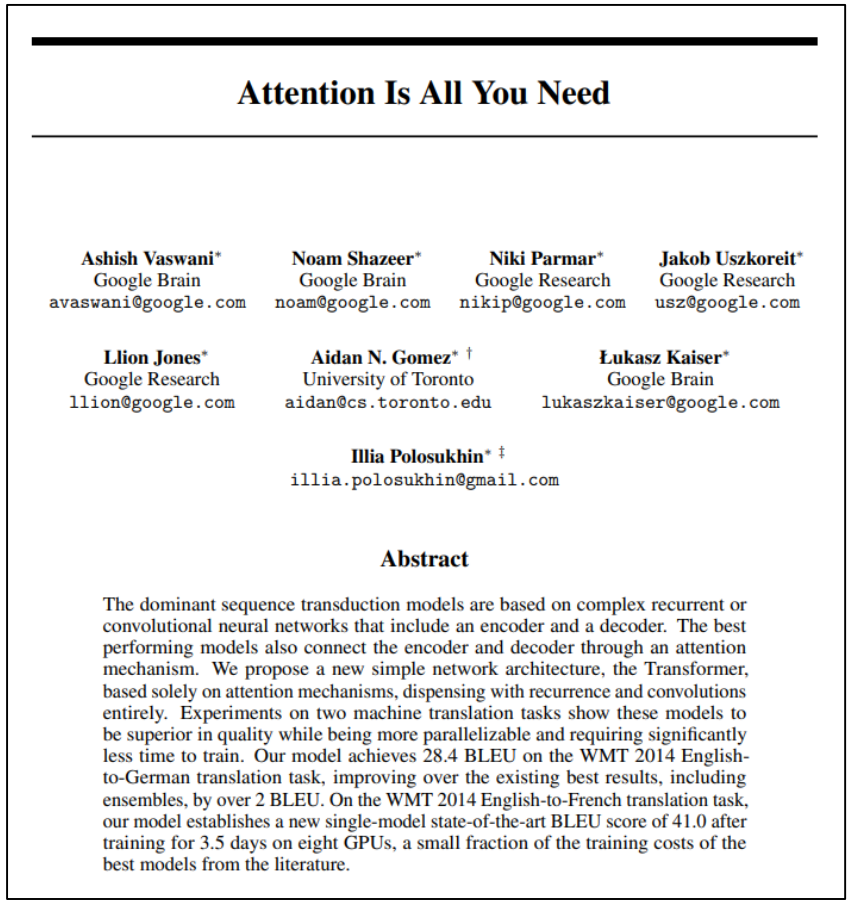
Limitations of Sequential Models

- Sequential dependency
- Vanishing/exploding gradient
- Memory constraints
- Contextual range
- Scalability

Create many new paths from Encoder to Decoder, one per input value, so each step of the Decoder can directly access input values.

Attention is all you need

- Google introduced the **transformer** in 2017
- Transformer models weigh the importance (attention) of different words
- Attention models have advantages:
  - Parallelization
  - Memory efficient
  - Adaptive focus/weighting
  - Scalable to input/output length



Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

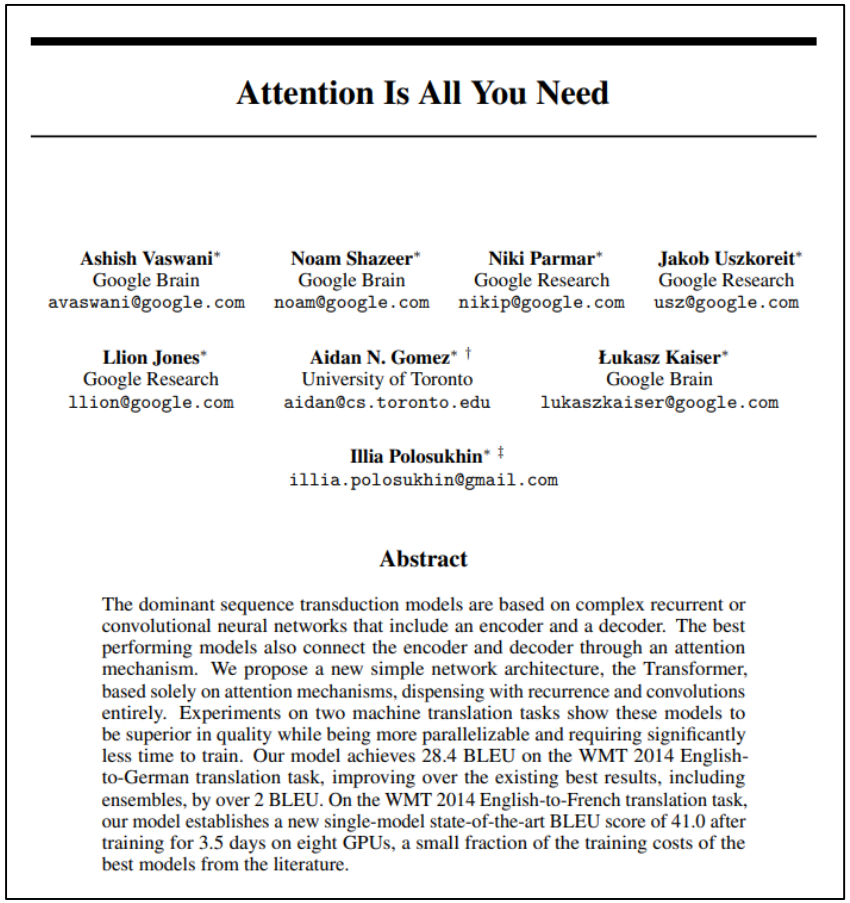
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention Is All You Need. In *Advances in Neural Information Processing Systems* (pp. 5998-6008).

Natural Language Processing

The Transformer Revolution

Attention is all you need

- Google introduced the **transformer** in 2017
- Focus on all tokens simultaneously
- Parallel computation = speed + scale
- Enables context (BERT) and generation (GPT)



Attention Is All You Need

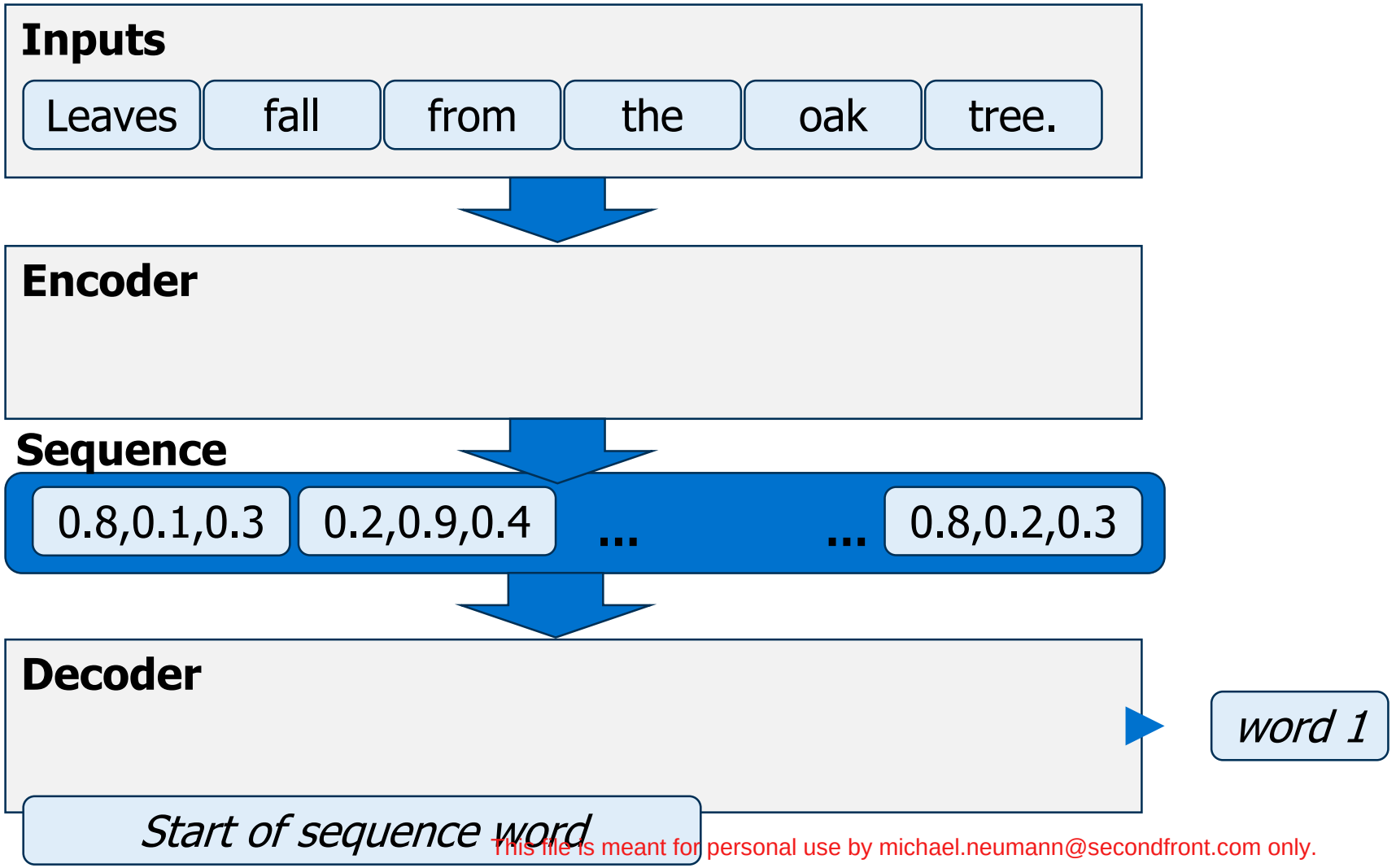
Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

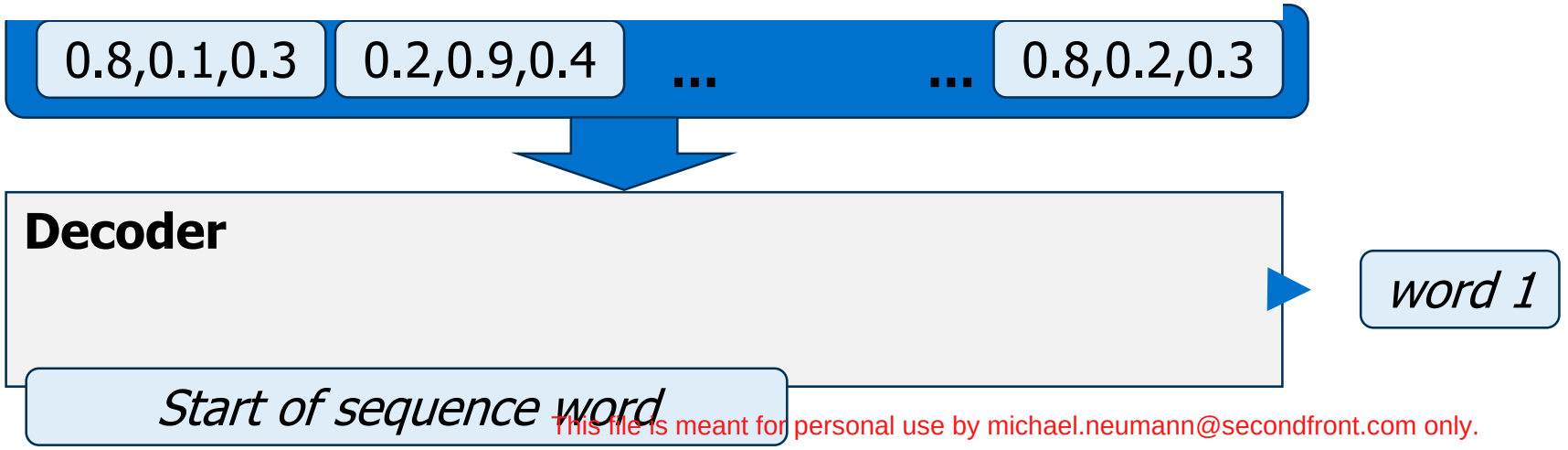
The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention Is All You Need. In *Advances in Neural Information Processing Systems* (pp. 5998-6008).

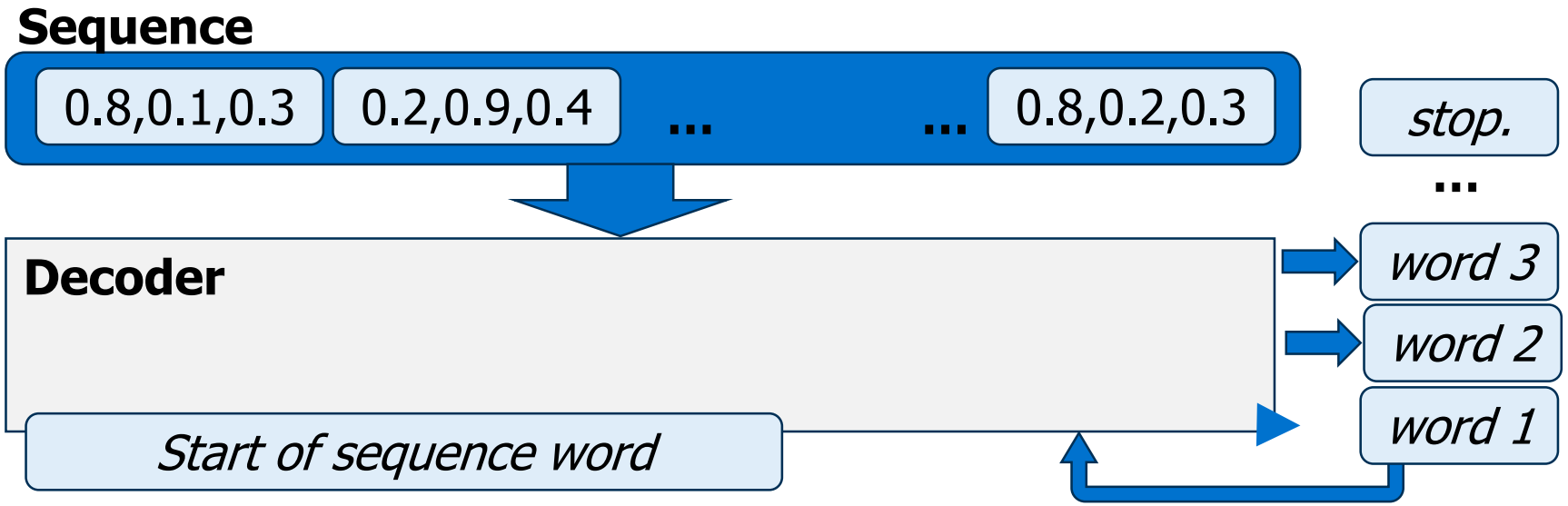
How Does a Transformer Work?



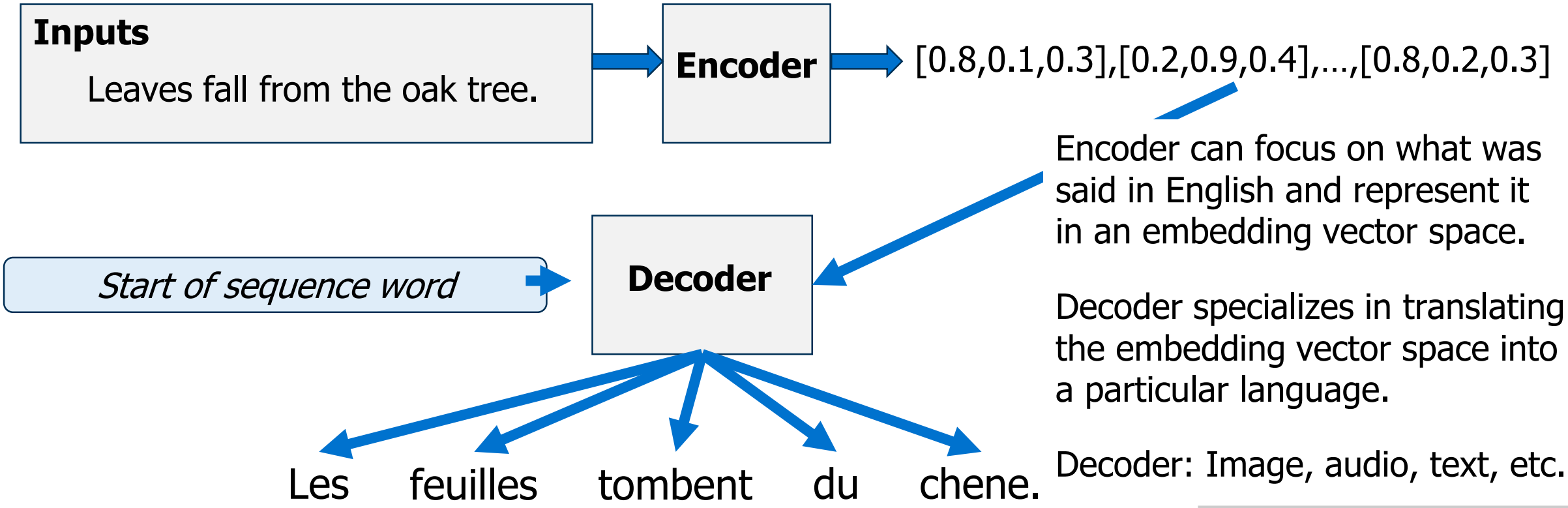
How Does a Transformer Work?



How Does a Transformer Work?



How Does a Transformer Work?



The encoder and decoder often do not share weights.

Key Components of Transformers

- Input embeddings: Converts tokens into dense embedding vectors
- Positional embeddings: Adds information about position of tokens

  e.g. ”John saw the dog” vs “The dog saw John”
- Self-attention: Focus on relevant parts of sentence when processing tokens

Attention is all you need

leaves /lēvz/
1. **noun**. Plural form of **leaf**
2. **verb**. Third-person singular form of **leave**.

Leaves fall from the oak tree.

- **Contextual Understanding**: considers the context of each word by neighboring words
- **Parallel Processing**: can process words all at once instead of sequentially
- **Long Range Dependencies**: No vanishing gradient.
- **Flexibility**: Pre-trained models like BERT, GPT without redesigning architecture

Natural Language Processing

Attention

Attention Models

- No standard rules for how attention should be added to encoder/decoder models.
- How similar are outputs from encoder at each step to outputs from decoder?
- Many ways to calculate similarity.
  - Different attention algorithms use different similarity metrics.

$$cosine\ similarity = \frac{A \cdot B}{\|A\|\|B\|}$$

Leaves fall ...

Attention Models

Leaves fall ...

$$cosine\ similarity = \frac{A \cdot B}{\|A\|\|B\|}$$

Encoder Model

- leaves = [0.8, 0.1, 0.3]
- fall = [0.2, 0.9, 0.4]

Decoder Model

- les = [0.5, 0.2, 0.1]
- feuilles = [0.9, 0.1, 0.2]
- Tombent = [0.3, 0.8, 0.5]

Similarity

- les - leaves = $\frac{0.5 \cdot 0.8 + 0.2 \cdot 0.1 + 0.1 \cdot 0.3}{0.5477 \cdot 0.8602} = 0.95$
- les - fall = $\frac{0.5 \cdot 0.2 + 0.2 \cdot 0.9 + 0.1 \cdot 0.4}{0.5477 \cdot 1.0049} = 0.58$

The word “les” is closer to ”leaves” than to “fall”.

Attention Models

Leaves fall ...

$$cosine\ similarity = \frac{A \cdot B}{\|A\|\|B\|}$$

Encoder Model

- leaves = [0.8, 0.1, 0.3]
- fall = [0.2, 0.9, 0.4]

Decoder Model

- les = [0.5, 0.2, 0.1]
- feuilles = [0.9, 0.1, 0.2]
- Tombent = [0.3, 0.8, 0.5]

part = [0.3, 0.7, 0.6]

Similarity

- feuilles - leaves = $\frac{0.72+0.01+0.06}{0.9274 \cdot 0.8602} = 0.99$
- feuilles - fall = $\frac{0.18+0.09+0.08}{0.9274 \cdot 1.0049} = 0.39$
- tombent - leaves = $\frac{0.24+0.08+0.15}{0.9899 \cdot 0.8602} = 0.52$
- tombent - fall = $\frac{0.06+0.72+0.20}{0.9899 \cdot 1.0049} = 0.89$
- part - leaves = $\frac{0.24+0.07+0.06}{0.9695 \cdot 0.8602} = 0.59$

Softmax Function

Converts inputs to numbers between 0 and 1 that sum to 1.0

$$P_i = \frac{e^{s_i}}{\sum_j e^{s_i}}$$

s_i is cosine similarity functional transformation

sum of all similarity scores

$s_{feuilles} = 0.99 \quad e^{0.99} = 2.6918$

$s_{part} = 0.59 \quad e^{0.59} = 1.7994$

$$\sum_j e^{s_i} = 2.6918 + 1.7994 = 4.4912$$

$$P_{feuilles} = \frac{2.6918}{4.4912} = 0.6$$

$$P_{part} = \frac{1.7994}{4.4912} = 0.4$$

Softmax normalizes scores
emphasizes differences
maintain context sensitivity

Attention Models

Target Word	Leaves	Fall
Les	0.7	0.3
Feuilles	0.5	0.5
Tombent	0.2	0.8

Feuilles attends to leaves and fall equally, while tombent attends heavily to fall.

Self-Attention and Transformers (1)

a spherical object

The	dog	chased	after	the	ball	as	it	rolled	down	the
Cinderella	went	to	the	ball	in	a	fancy	glass	coach	

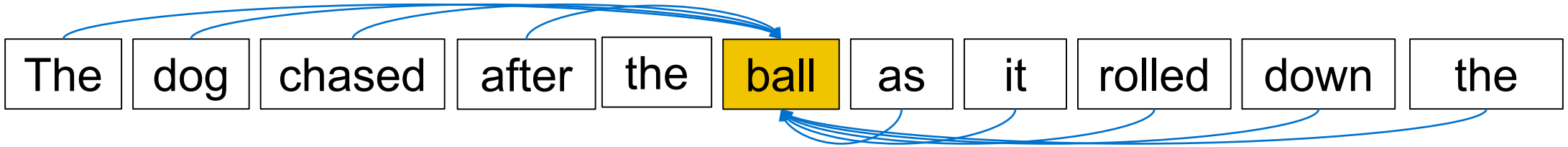
a formal dance party

The	dog	chased	after	the	ball	as	it	rolled	down	the
Cinderella	went	to	the	ball	in	a	fancy	glass	coach	

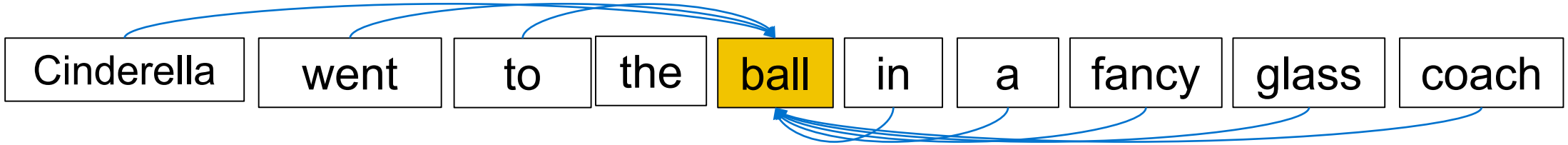
- Different contextual words are *attended to* when encoding different meanings:

Self-Attention and Transformers (2)

- Transformer networks predict tokens within the context of others



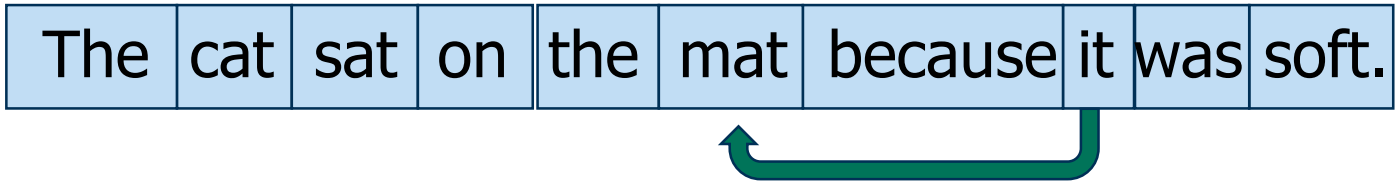
- Doing so enables the network to understand how the same word can be used in different ways, depending on the context



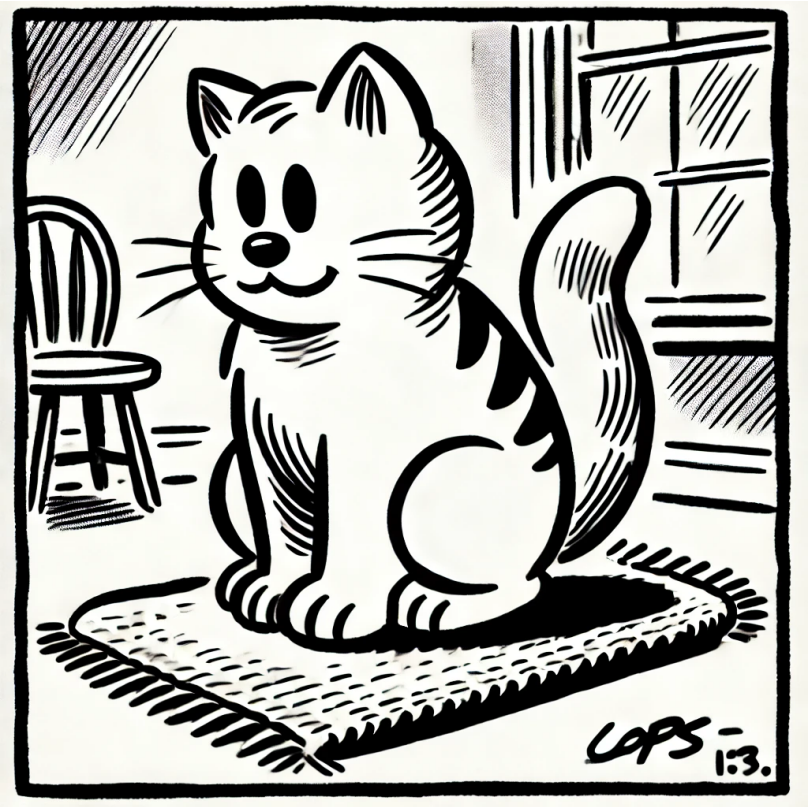
Natural Language Processing

Coding a Simple Attention Model

Attention Example



Attention improves on LSTM by allowing dynamic focus



Components of Attention

1. **Query (QQ):** What we're looking for (e.g., "it" in sentence).
2. **Keys (KK):** Possible matches (e.g., "cat," "mat," etc.).
3. **Values (VV):** The actual information we want to retrieve (e.g., representations of "cat," "mat").
4. **Scores:** A similarity measure between the query and keys, determining how much focus each part of the input receives.

Example: "I love pizza, but hate olives." – love focused on pizza

Python

python Copy code

```python
import numpy as np

# Define query, keys, and values as vectors
query = np.array([1, 0, 1])  # Represents "love"
keys = np.array([[1, 0, 1],  # Represents "pizza"
                 [0, 1, 0],  # Represents "but"
                 [1, 0, -1]])  # Represents "olives"
values = np.array([[5], [0], [-3]])  # Sentiment scores for "pizza," "but," "oli

# Calculate attention scores (dot product of query and keys)
scores = np.dot(keys, query)

# Apply softmax to get weights
def softmax(x):
    exp_x = np.exp(x - np.max(x))  # Subtract max for numerical stability
    return exp_x / np.sum(exp_x)

weights = softmax(scores)

# Weighted sum of values
attention_output = np.dot(weights, values)

print("Attention Scores:", scores)
print("Attention Weights:", weights)
print("Attention Output:", attention_output)
```

Write a simple Python script to demonstrate how attention mechanisms work in large language models using a basic example sentence, 'I love pizza, but hate olives.'
Calculate attention scores using the dot product, apply the softmax function to get attention weights, and compute the final attention output as a weighted sum of the values. Include comments explaining each step.

Output Explanation

1. **Scores:** The dot product measures similarity between the query ("love") and each key.
2. **Weights:** Softmax normalizes scores to probabilities, shows focus allocated to each token.
3. **Output:** The weighted sum retrieves the most relevant value (e.g., the sentiment for "pizza").

To improve stability for high-dimensional vectors, we scale the dot product by dividing by the square root of the key dimension (dkdk):

python Copy code

```python
import numpy as np

# Define query, keys, values, and scaling factor
query = np.array([1, 0, 1])
keys = np.array([[1, 0, 1],
                 [0, 1, 0],
                 [1, 0, -1]])
values = np.array([[5], [0], [-3]])
```

Why Attention Matters

- **Improving Performance:** Enabling models to handle long-range dependencies better.
- **Enhancing Flexibility:** Supporting tasks like translation, summarization, and question answering.
- **Powering Transformers:** Laying the foundation for state-of-the-art architectures like BERT and GPT.

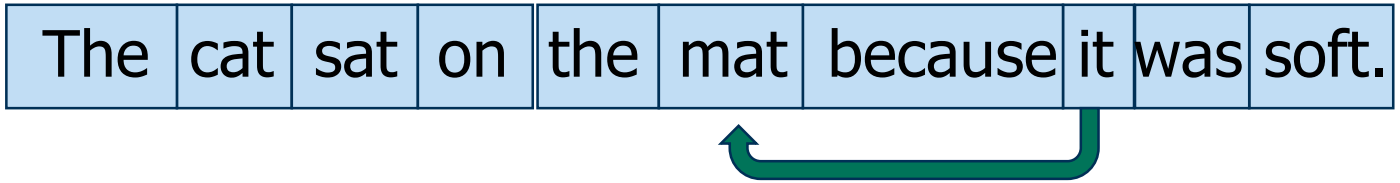
Natural Language Processing

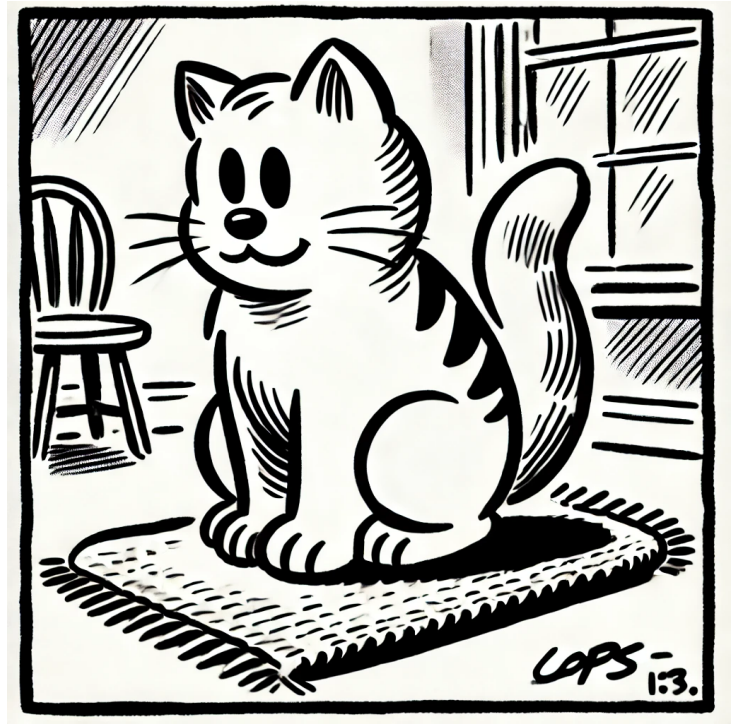
Multi-Head Attention

Self vs Multi-Head Attention

- Self-attention allows the model to determine which parts of an input sequence are most relevant to understanding any given word
- Multi-head attention extends self-attention by allowing the model to focus on different types of relationships simultaneously.

Self Attention Example





Attention improves on LSTM by allowing dynamic focus

How Self-Attention Works

1. Generate Query, Key, and Value Vectors:

For each word in the sequence, create three vectors:

Query (QQ): What the word is looking for.

Key (KK): How the word describes itself.

Value (VV): The information carried by the word.

2. Compute Attention Scores:

Use the dot product of the **Query** and **Key** vectors to calculate how much attention one word should pay to another.

3. Apply Softmax and Weighted Sum:

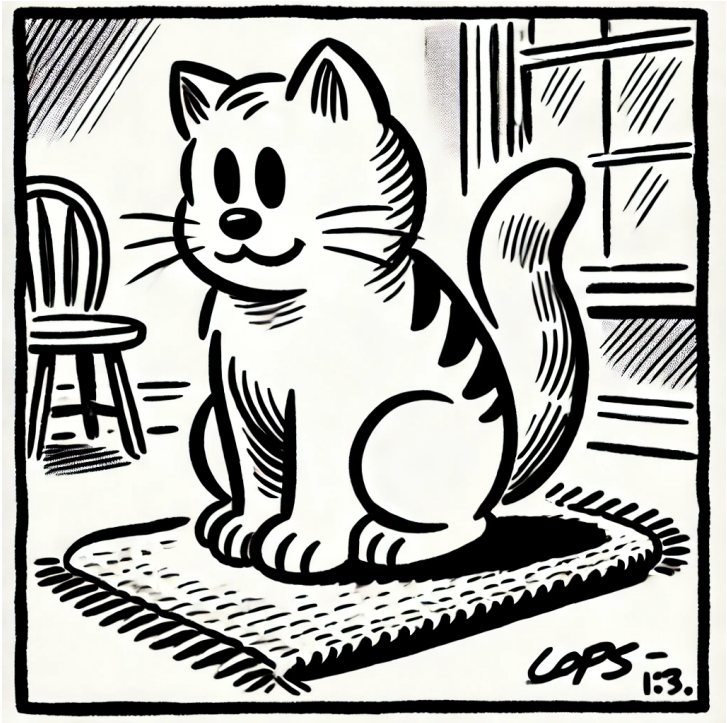
Normalize the scores using softmax.

Combine the **Value** vectors and weights, producing the final output.

Self Attention Example

```python
import numpy as np

# Define sentence embeddings (each row represents a word)
sentence_embeddings = np.array([
    [1, 0, 1],  # "The"
    [0, 1, 0],  # "cat"
    [1, 1, 1],  # "sat"
    [0, 0, 1],  # "on"
    [1, 0, 0]   # "mat"
])

# Initialize weight matrices for Q, K, V
W_q = np.random.rand(3, 3)
W_k = np.random.rand(3, 3)
W_v = np.random.rand(3, 3)

# Compute Q, K, V vectors
Q = sentence_embeddings @ W_q  # Queries
K = sentence_embeddings @ W_k  # Keys
V = sentence_embeddings @ W_v  # Values

# Compute attention scores (scaled dot product)
scores = Q @ K.T  # Shape: (num_words, num_words)

# Scale scores by the square root of key dimension
d_k = K.shape[1]
scores /= np.sqrt(d_k)
```

Copy code



Interpret Output

Attention Weights Matrix

- The **rows** correspond to ea. word in sentence (5 words).
- The **columns** are attention each word pays other words.
- The values in each row sum to **1** (softmax normalization).

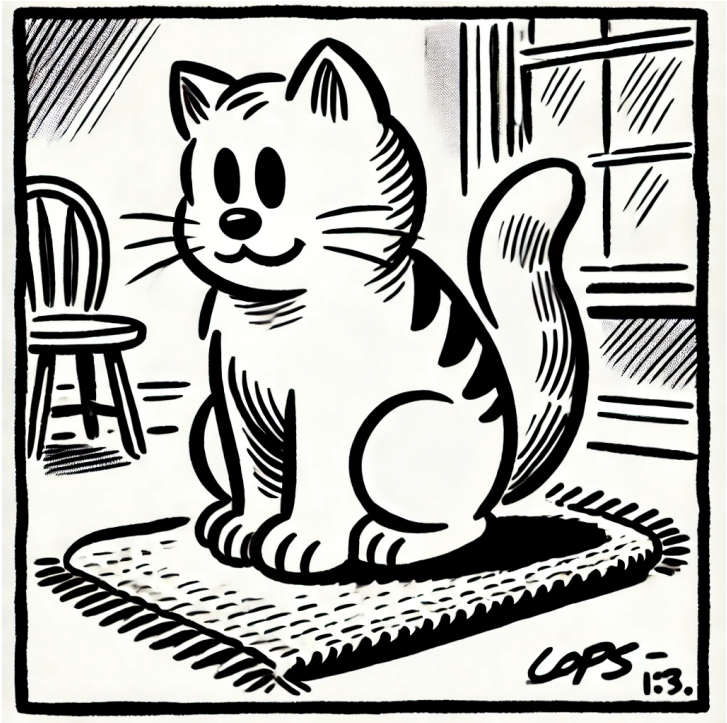
Row 3 (index 2, third word "sat"):

[0.16487252 0.08927606 0.6205296 0.06660528 0.05871654]

This means **"sat"** pays:**62.05% attention to itself**
(dominates self-attention).

- **16.49% to "The".**
- **8.92% to "cat".**
- **6.66% to "on".**
- **5.87% to "mat".**

Suggests **"sat" is the most self-reliant** in context



Interpret Output

Attention Output Matrix
Transformed embeddings after attention applied.

Row 2 (word "cat"):
[0.74618576 0.38450039 1.08029289]

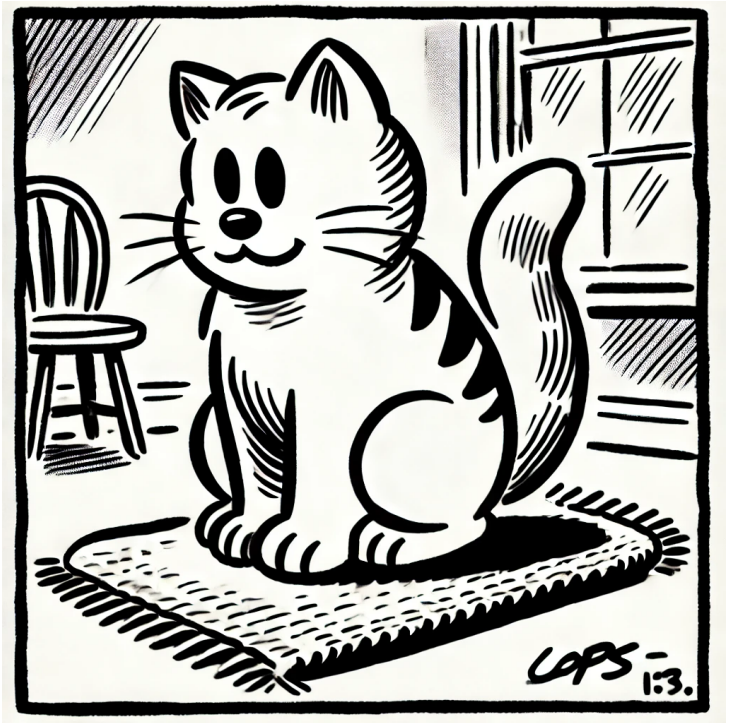
Row 3 (word "sat"):
[0.99057756 0.50673902 1.39992304]

Sat is the **most significant row** because it has **higher values** across dimensions.

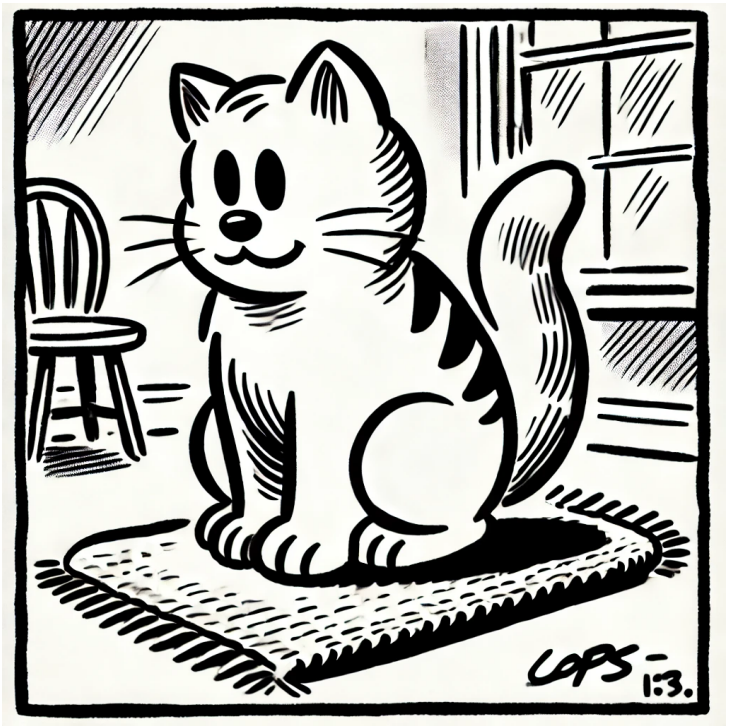
It may act as a **contextually central word** in this phrase.

"Cat" had a **more even attention distribution**

Its embedding is more **balanced and contextually adjusted**



Multi-Headed Attention



Multi-Head Setup

- Two attention heads are created.
- Each head processes a different projection of the embeddings.
- Each head has its own **Q, K,** and **V** transformation matrices.

Per-Head Attention Calculation

- Each head computes attention scores as in self-attention

Concatenation

- Attention outputs for both heads are concatenated.

Captures Different Relationships:

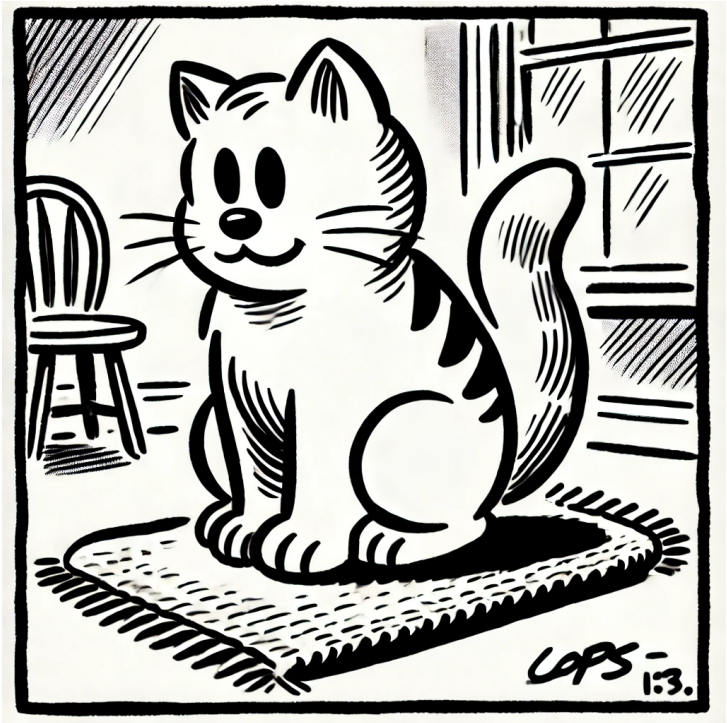
Each head focuses on **different aspects** of the sentence:

- Head 1 might focus on **subject-verb** relationships (e.g., "cat" → "sat").
- Head 2 might focus on **object-preposition** relationships (e.g., "mat" → "on").

Improves: Expressiveness, Contextual Understanding, and AI/ML Performance

Multi-Headed Attention

```
# Multi-head attention will have two heads for this example
num_heads = 2
head_dim = sentence_embeddings.shape[1] // num_heads  # Dimension per head

# Function to create weight matrices for each head
# We'll create separate W_q, W_k, W_v for each head
np.random.seed(42)  # Seed for reproducibility
def generate_weights(num_heads, head_dim):
    return [
        (np.random.rand(head_dim, head_dim),  # W_q for this head
         np.random.rand(head_dim, head_dim),  # W_k for this head
         np.random.rand(head_dim, head_dim))  # W_v for this head
        for _ in range(num_heads)
    ]

```

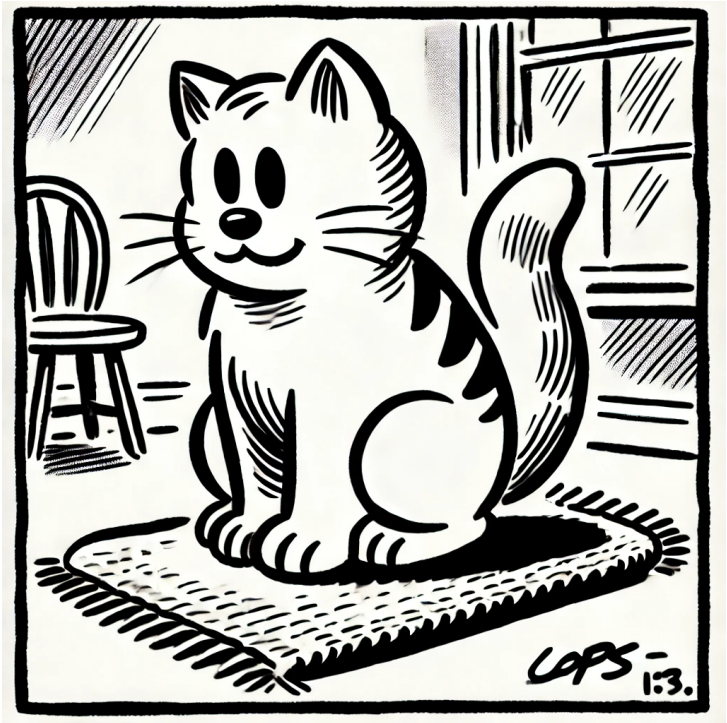


Attention in models like BERT and GPT **initially uses random weights** for the **query** (W_q), **key** (W_k), and **value** (W_v) **matrices**, but these weights are **learned and updated** during training through **backpropagation**.

- Avoiding Symmetry: Each head learns unique relationships in the data.
- Learning Complex Patterns: Discover patterns through training.
- Standard Practice in Neural Networks: Avoids vanishing/exploding gradient

Interpret Head Output

```
[[0.19687765 0.20256664 0.20184662 0.19757994 0.20112916]
 [0.1727621  0.2232503  0.21620917 0.17838832 0.20939011]
 [0.17568776 0.22065491 0.21445791 0.18076445 0.20843496]
 [0.193778   0.20513864 0.20368291 0.19516293 0.20223752]
 [0.17863977 0.18314844 0.21269322 0.21806136 0.20745723]]
```



Row 1 ("The"): Attention spread evenly, not focused on specific word
This makes sense for a determiner.

Row 2 ("cat"): Higher attention (0.22) is given to itself and "sat",
suggests "cat" is contextually linked to "sat".

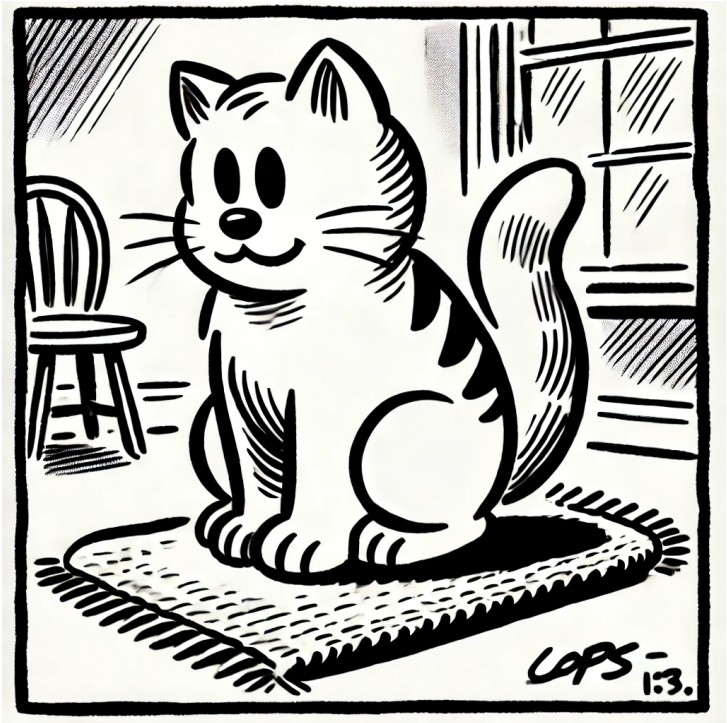
Row 5 ("mat"): High attention on "sat" (0.21) and "on" (0.21),
"mat" connects strongly to the action and location.

The second head is more uniform and less selective – global sentence structure

```
[[0.19921608 0.20052285 0.20071023 0.19940224 0.20014861]
 [0.19376598 0.20417066 0.20570196 0.19521925 0.20114215]
 [0.19299284 0.20468993 0.20641784 0.19462201 0.20127738]
 [0.19843347 0.20104528 0.20142119 0.1988045  0.20029556]
 [0.19531638 0.20313068 0.20427224 0.19641403 0.20086666]]
```

Interpret Combined Output

```
[[0.39804471 0.08126514]
 [0.41973964 0.08229866]
 [0.41707689 0.08244593]
 [0.40080164 0.08141305]
 [0.41439898 0.08200382]]
```



- The output combines information from both heads.
- **Head 1** provides **focused contextual information**
- **Head 2** offers a **general overview** of the sentence structure.
- **"cat" (0.4197, 0.0823)** and **"sat" (0.4171, 0.0824)** remain dominant, indicating that they are key elements in this sentence.

Summary

- Attention provides **context-aware representations** for each word, improving the model's ability to handle complex sentence structures.
- This mechanism helps models understand **long-range dependencies** and **resolve ambiguity** in language
- Multi-headed attention **splits attention into multiple heads**, each learning **different relationships** in the sentence.
- They enable modern language models to achieve **human-like comprehension and generation capabilities**.

Natural Language Processing

Encoder-Only Models: BERT

Encoder-Only Models

- Focus exclusively on understanding and processing input text
- Designed for language understanding tasks such as:
  - Text classification (e.g., sentiment analysis, spam detection)
  - Named entity recognition (NER)
  - Question answering (extractive QA)
  - Text similarity detection

BERT (Bidirectional Encoder Representations from Transformers)

- Bidirectional Contextual Understanding
  - Example: **"He went to the bank to deposit money”**
- Masked Language Modeling (MLM) during training
  - Example Input: **"The cat sat on the [MASK]."**
- Next Sentence Prediction (NSP)

BERT (Bidirectional Encoder Representations from Transformers)

- Architecture of BERT
  - BERT-Base: 12 layers, 768 hidden units, 12 attention heads.
  - BERT-Large: 24layers, 1024 hidden units, 16 attention heads.
- Each encoder layer applies **multi-headed** self-attention and **feed-forward networks** to transform the input.

Fine-Tuning BERT

- Once BERT is pretrained, it can be fine-tuned on specific tasks with minimal modifications:
- Text Classification: Add a classification head on top and train with labeled data.
- Question Answering: Add an output layer to predict the start and end of the answer span.
- NER: Add token classification head to label tokens.

Why is BERT Important?

- State-of-the-Art Performance: BERT set new benchmarks on multiple NLP tasks.
- Contextual Embeddings: Unlike static embeddings (like Word2Vec), BERT's embeddings are contextual and dynamic, changing based on sentence context.
- Transfer Learning: BERT is pretrained on large corpora (e.g., Wikipedia), allowing fine-tuning on smaller datasets for specific tasks.

Summary

- Encoder-only models designed for language understanding tasks.
- BERT's bidirectional attention and masked language modeling help it capture deep context and relationships within text.
- BERT is widely used for text classification, NER, question answering, and other NLP tasks.
- Fine-tuning BERT for a specific task is straightforward and highly effective.

Natural Language Processing

Decoder-Only Models: GPT

Decoder-Only Model

- Transformer model that focuses on text generation and language modeling.
- Excel at generating coherent and contextually appropriate text based on a given input
- Example Tasks for Decoder-Only Models:
  - Text generation: Generate stories, descriptions.
  - Auto-complete: Predict the next word.
  - Dialogue generation: Power chatbots.
  - Summarization: Generate concise summaries.

How GPT Works

- GPT (Generative Pretrained Transformer) is the most well-known decoder-only model, developed by Open AI.
- Consists only of decoder blocks, trained to predict the next token given a sequence of preceding tokens – **causal language model**
- Unidirectional attention (only look at previous tokens).
- Generating text (e.g., text completion, summarization)

GPT Architecture

- Each decoder layer contains
  - Self-Attention Layer (Causal Attention): focuses **only on past tokens**.
  - Feed-Forward Network (FFN): one-direction, no loops for efficiency, scale, performance
  - Layer Normalization and Residual Connections: improves training stability.
- Pre-trained on massive data set including Wikipedia, books, etc.
- Scalability and Performance
  - GPT models are available in various sizes: from GPT-2 (1.5 billion parameters) to GPT-3 (175 billion parameters) and GPT-4.
  - Larger models demonstrate better fluency and understanding.

Coding a GPT

```python
from transformers import GPT2Tokenizer, GPT2LMHeadModel

# Load pre-trained GPT tokenizer and model
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model = GPT2LMHeadModel.from_pretrained("gpt2")

# Example prompt
prompt = "Once upon a time in a faraway land,"

# Tokenize the input prompt
inputs = tokenizer.encode(prompt, return_tensors="pt")

# Generate text
output = model.generate(inputs, max_length=50, num_return_sequences=1, temperature=0.7)

# Decode the generated text
generated_text = tokenizer.decode(output[0], skip_special_tokens=True)
print("Generated Text:", generated_text)
```

Comparing GPT to BERT

Feature	Encoder-Only Models (BERT)	Decoder-Only Models (GPT)
Primary Task	Understanding text	Generating text
Attention Type	Bidirectional	Unidirectional (causal)
Example Tasks	Classification, QA	Text generation, summarization
Architecture	Stacked encoder layers	Stacked decoder layers
Pretraining Task	Masked language modeling	Causal language modeling
Strengths	Contextual understanding	Fluent text generation

What Makes GPT Powerful

- **Causal Attention:** Generates coherent text only using prior tokens.
- **Pretraining on Large Datasets:** Enables broad knowledge and generalization capabilities.
- **Zero-Shot and Few-Shot Learning:** Perform tasks w/ no examples.
- **Multitasking:** Capable of writing, translating, summarizing, coding, and more—all with the same architecture.

Natural Language Processing

Encoder-Decoder Models: T5

Encoder-Decoder Models

- An **encoder-decoder** model is a Transformer-based architecture designed for **sequence-to-sequence** (Seq2Seq) tasks.
  - **Encoder:** Processes the input and converts it into a rich, context-aware representation.
  - **Decoder:** Takes this representation and generates the output sequence one token at a time, using both the encoded input and previously generated tokens as context.

- Ideal for transformation of one sequence into another, such as:
  - **Machine Translation**: Translating text from one language to another
  - **Summarization**: Generating a summary for a given document.
  - **Question Answering**: Generating answers based on context.
  - **Text Simplification**: Rewriting complex text in simple language.

T5 (Text-to-Text Transfer Transformer)

- Unlike traditional models that focus on specific tasks (e.g., translation), T5 reframes every NLP task as a text-to-text problem.
- Pretraining with Span Corruption (Fill-in-the-Blank):
  Similar to BERT's masked language modeling, T5 is pretrained by masking random spans of text and asking the model to fill in the blanks. This task is called "span corruption."
- Example:
  - Input: "The [MASK] is blue and [MASK] over the sea."
  - Target: "sky stretches"
- Scalability: multiple sizes (from **T5-Small** to **T5-XXL**)
- Few-shot and Zero-shot learning: minimal fine-tuning.

T5 (Text-to-Text Transfer Transformer)

```python
from transformers import T5Tokenizer, T5ForConditionalGeneration

# Load pre-trained T5 tokenizer and model
tokenizer = T5Tokenizer.from_pretrained("t5-small")
model = T5ForConditionalGeneration.from_pretrained("t5-small")

# Input text for translation
input_text = "translate English to French: The weather is sunny."

# Tokenize the input text
inputs = tokenizer(input_text, return_tensors="pt")

# Generate translation
output = model.generate(inputs.input_ids, max_length=50)

# Decode the generated translation
translation = tokenizer.decode(output[0], skip_special_tokens=True)
print("Translation:", translation)
```

Summary

- Encoder-decoder models are ideal for sequence-to-sequence tasks (e.g., translation, summarization).
- T5 reframes every task as text-to-text, making it highly flexible and powerful for a wide range of NLP tasks.
- T5's span corruption pretraining and scalable architecture allow it to outperform traditional models in many tasks.