LARGE LANGUAGE MODELS

Dr. Bishnu Sarker, Cameron Pykiet, and Jaylin Dyson

Meharry SACS

South Dakota State University

Meharry Medical College School of Applied Computational Sciences 2021

U.S. National Science Foundation

WHAT WE PLAN ON COVERING

- How popular LLMs work internally
- Way to improve LLM model outcomes
- Creating quick interactive web apps using Streamlit
- Practical application of Streamlit apps.
  - Sentiment analysis of text application
  - Retrieval-Augmented Generation application with Ollama
  - Hugging face protein function prediction application

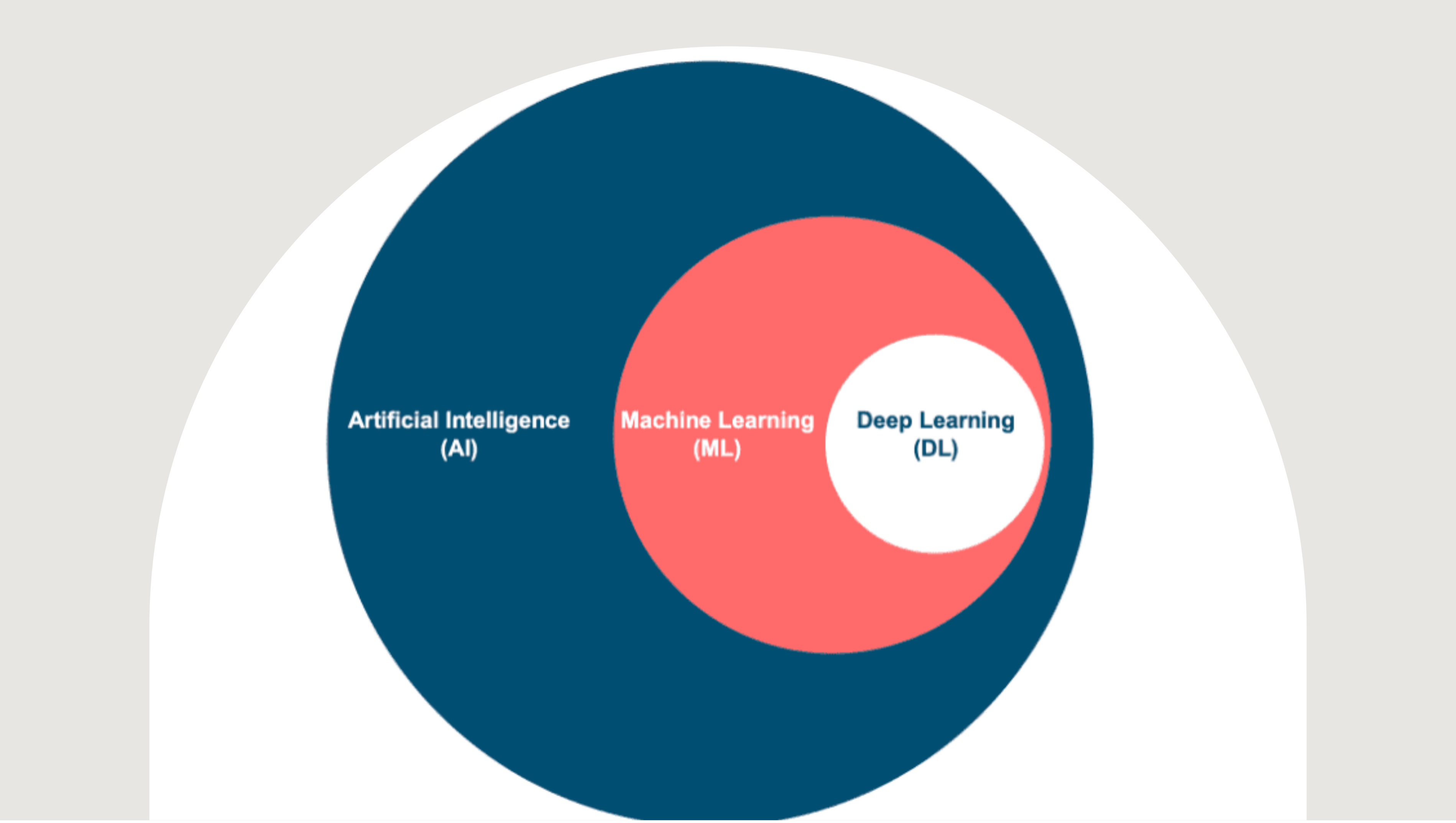
WHAT ARE LLMS?

Everyone has heard of them by now, but formally, Large Language models are nothing more than a large scaled predictive model that utilizes efficient coding architectures to generate human-like text. They work through learning the patterns of language through vast datasets of text and using transformers to understand the context and relationships between words.

Their revolutionary success comes from attention mechanisms in a transformer that allows focus on relevant parts of the data that leads to understanding and generating accurate text.

They have shown promise in NLP, content generation, and research.

These models are very powerful, so it is important to know how are built so that users are better equipped for challenges and/or issues when they arise.


Artificial Intelligence
(AI)
Machine Learning
(ML)
Deep Learning
(DL)

OVERVIEW OF GPT ARCHITECTURE

Decoder: GPT is based on the transformer model and uses only the decoder portion.

Causal Attention Masking: Ensures the model only considers preceding tokens to generate text step-by-step.

Scalability: Scales with more layers, larger embeddings, and larger datasets.

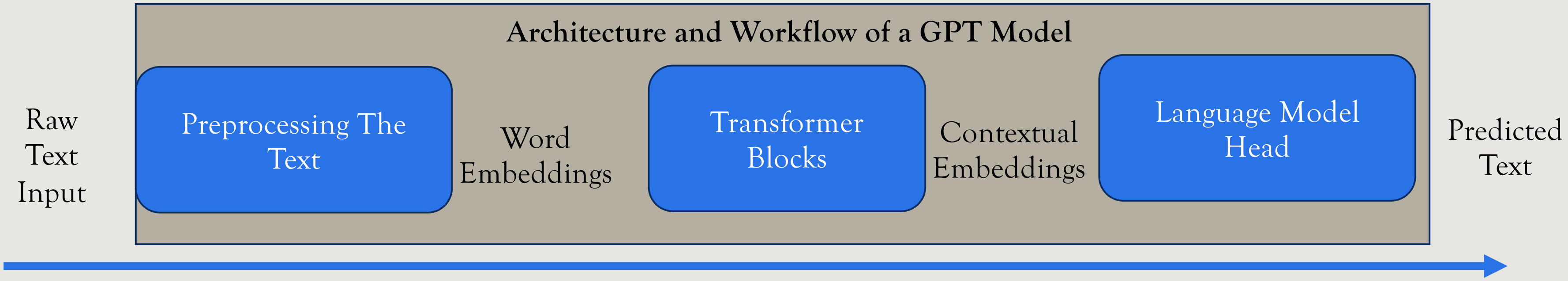
Unidirectional Context: Processes text left-to-right, focusing on sequential token generation.

Self-Attention: Tokens understand relationships by attending to each other.

Feedforward Layers: Add complexity and non-linearity to improve predictions.

This architecture has evolved through versions (GPT-1, GPT-2, GPT-3, GPT-4), with improvements in size, training data, and fine-tuning techniques for better performance and versatility

GPT Model



Architecture and Workflow of a GPT Model

Raw Text Input

Preprocessing The Text

Word Embeddings

Transformer Blocks

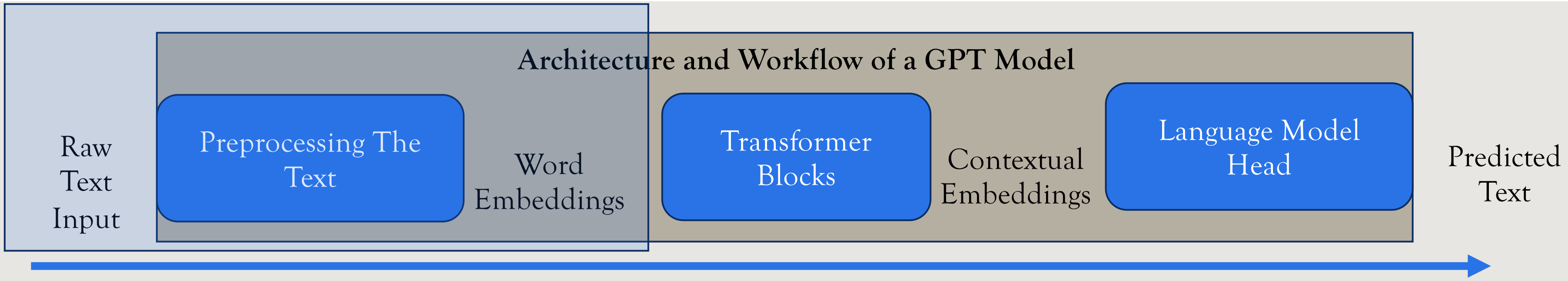
Contextual Embeddings

Language Model Head

Predicted Text

PROCESSING TEXT OVERVIEW:

- Because LLMs use text data, there are several natural language processing steps prior to vectorizations
  1. Cleaning Text
     - Remove noise, special characters, whitespace, lowercasing (consistency)
  2. Normalization
     - Handing contractions, spelling variations, stemming/lemmatization
  3. Stop Words
     - Weighing or removing common words (the, is, and)
  4. Tokenization (VOCABULARY)
     - Splitting words into subwords with techniques such as BPE (byte pair encoding)
     - Initial step between text representation and numerical representation



Architecture and Workflow of a GPT Model

Raw Text Input

Preprocessing The Text

Word Embeddings

Transformer Blocks

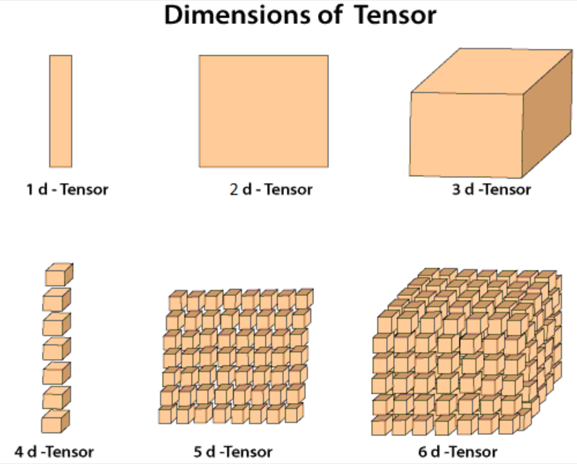
Contextual Embeddings

Language Model Head

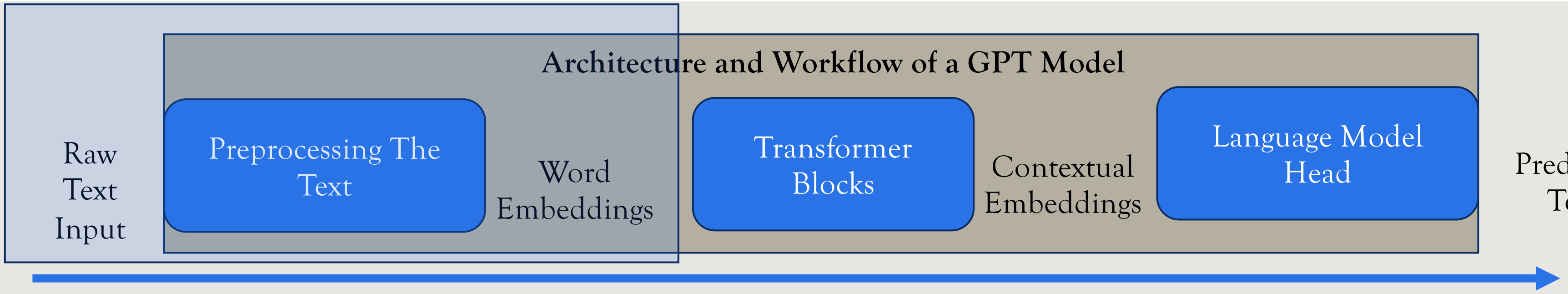
Predicted Text

PROCESSING TEXT OVERVIEW:

VECTORIZATION



- Following tokenization, tokens get vectorized into tensors to allow the model to process the tokens.
- These representations are also known as embeddings
- These embeddings are stored in tensors (multidimensional arrays)
- Embedding dimension determines the size of each token vector.
  - This effects overall parameter count
- Obviously, larger dimensions capture more info, but it comes at a larger computational cost



Architecture and Workflow of a GPT Model

Raw Text Input

Preprocessing The Text

Word Embeddings

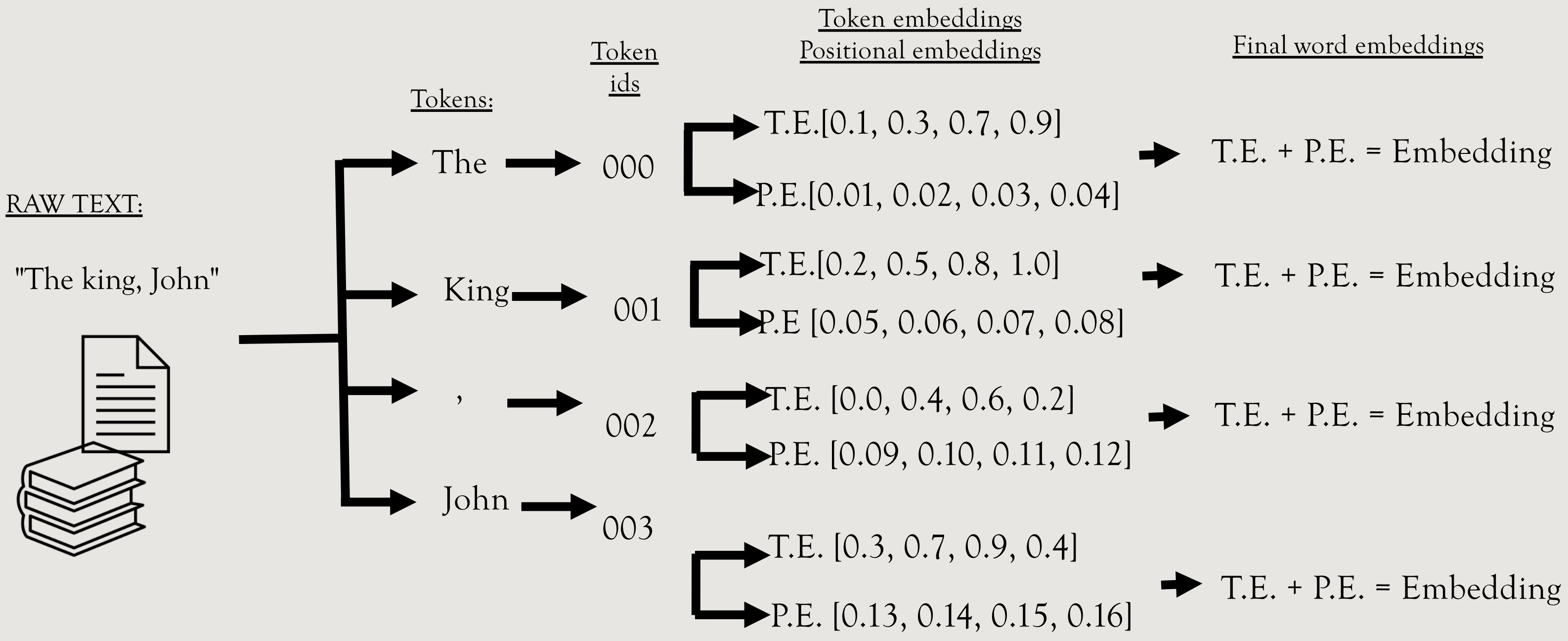
Transformer Blocks

Contextual Embeddings

Language Model Head

Predicted Text

PROCESSING TEXT OVERVIEW:



RAW TEXT:

"The king, John"

Tokens:

Token ids

Token embeddings

Positional embeddings

Final word embeddings

The → 000

T.E.[0.1, 0.3, 0.7, 0.9]

P.E.[0.01, 0.02, 0.03, 0.04]

→ T.E. + P.E. = Embedding

King → 001

T.E.[0.2, 0.5, 0.8, 1.0]

P.E [0.05, 0.06, 0.07, 0.08]

→ T.E. + P.E. = Embedding

, → 002

T.E. [0.0, 0.4, 0.6, 0.2]

P.E. [0.09, 0.10, 0.11, 0.12]

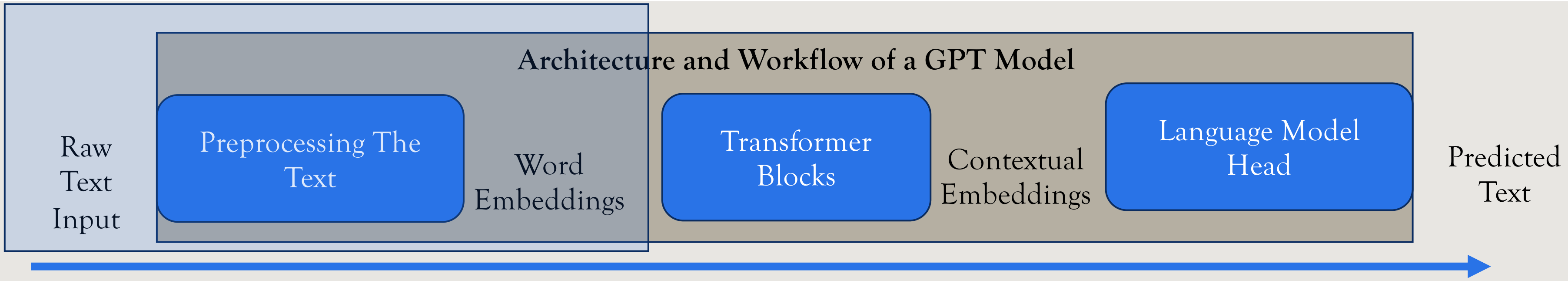
→ T.E. + P.E. = Embedding

John → 003

T.E. [0.3, 0.7, 0.9, 0.4]

P.E. [0.13, 0.14, 0.15, 0.16]

→ T.E. + P.E. = Embedding



Architecture and Workflow of a GPT Model

Raw Text Input

Preprocessing The Text

Word Embeddings

Transformer Blocks

Contextual Embeddings

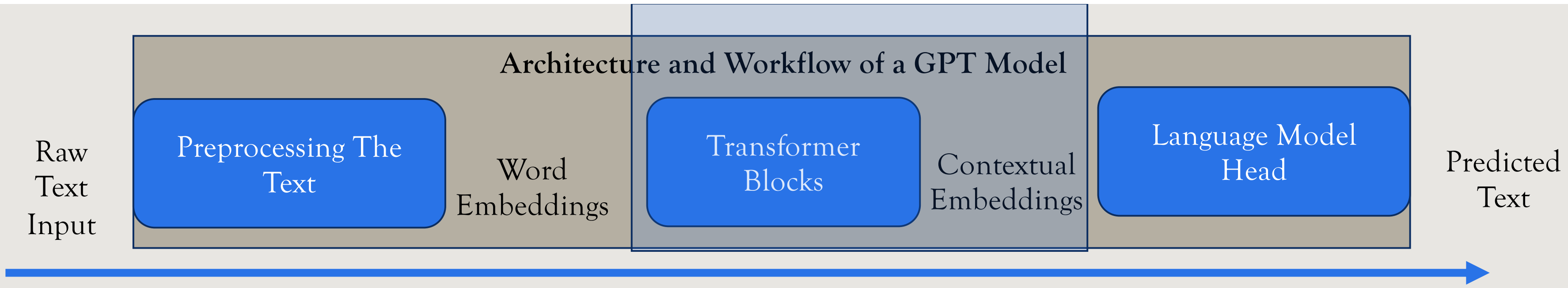
Language Model Head

Predicted Text

PROCESSING TEXT OVERVIEW:

- There are different ways and methods to achieve tokenization and embedding.
  - **Chat GPT uses Byte Pair Encoding (BPE)**

Category	Methods	Tools/Examples
Word-level	Splits into full words	NLTK, SpaCy, Gensim
Subwords	Splits into subwords	BPE (OpenAI GPT), WordPiece (BERT), SentencePiece (T5)
Character-Level	Splits into individual chars	Custom Python, Byte-Level (GPT-2/3)
Sentence-Level	Splits into sentences	NLTK, SpaCy, Hugging Face Transformers
Hybrid	Combines multiple levels	SentencePiece, Custom Pipelines
Specialized	Domain-specific tokens	Galactica (Science), Codex (Code)



Architecture and Workflow of a GPT Model

Raw Text Input

Preprocessing The Text

Word Embeddings

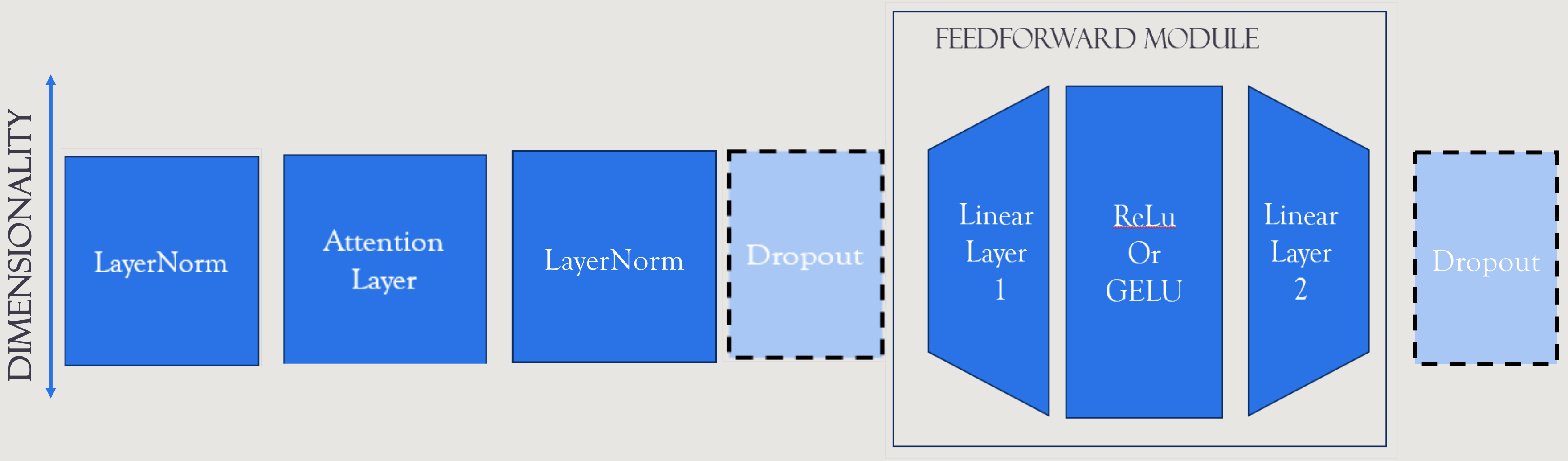
Transformer Blocks

Contextual Embeddings

Language Model Head

Predicted Text

TRANSFORMER BLOCK OVERVIEW:



DIMENSIONALITY

LayerNorm

Attention Layer

LayerNorm

Dropout

FEEDFORWARD MODULE

Linear Layer 1

ReLu Or GELU

Linear Layer 2

Dropout

LAYERNORM

Why is Layer Normalization Important?

- **Helps the network learn faster**: When the numbers are more consistent, the network doesn’t get confused by very big or very small values. It can focus on learning patterns in the data instead of struggling with the math.
- **Improves stability:** Big or small numbers can make the training process unstable (like burning or undercooking cookies). Layer normalization keeps things under control.
- **Works with any batch size**: Unlike some other methods (like batch normalization), this doesn’t depend on how many examples you feed into the network at once. So it’s great when you’re working with small batches of data or even single inputs.

LAYER NORM

Layer normalization takes the outputs (or activations) of a layer and:

1. *Finds the average (mean)*
2. *Calculates how far the numbers are spread out (variance)*
3. *Adjust the numbers to fit into a standard range (mean = 0, variance = 1). This makes data uniform.*

Layer normalization includes two extra ingredients:

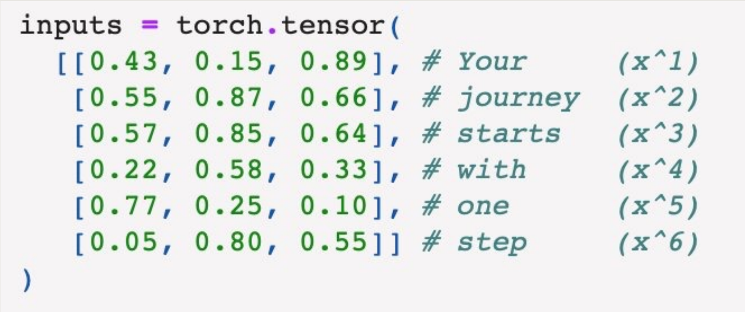
- ***Scale (γ)**: This allows the network to adjust how "big" the numbers should be after normalization.*
- ***Shift (β)**: This lets the network tweak where the numbers are centered.*

WHY ATTENTION MECHANISMS

- Pre-LLM architectures were not good at formulating context which is essential for things such as translating due to grammatical differences.
- Recurrent Neural Networks (RNNs) were a popular choice as they took input text, fed it into an encoder which processes sequentially. Using hidden layers, it tries to capture meaning of the input. This was bad as it couldn't access prior hidden states.
- RNNs were effective for short text, but lost meaning with longer text

ATTENTION MECHANICS

- Self-attention mechanisms allowed for each position in an input sequence to consider the relevancy of all other positions in the same sequence. This is one of the key components of the transformer architecture.
- The output for attention mechanisms is a context vector for each word which are constructed through a series of mathematical processes with the embeddings.

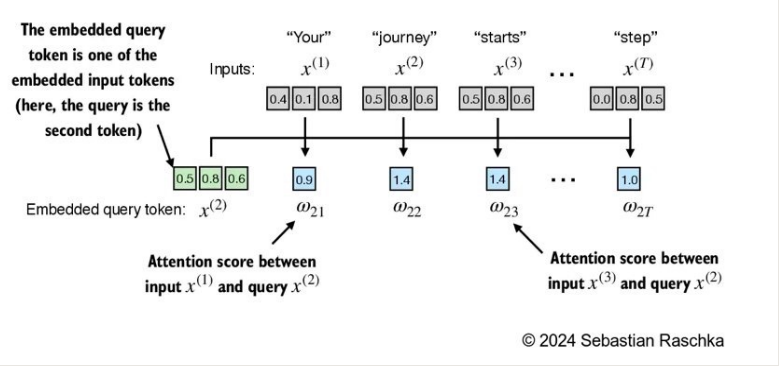


```
inputs = torch.tensor(
  [[0.43, 0.15, 0.89], # Your     (x^1)
   [0.55, 0.87, 0.66], # journey  (x^2)
   [0.57, 0.85, 0.64], # starts   (x^3)
   [0.22, 0.58, 0.33], # with     (x^4)
   [0.77, 0.25, 0.10], # one      (x^5)
   [0.05, 0.80, 0.55]] # step     (x^6)
)
```

ATTENTION MECHANISMS

Attention Score

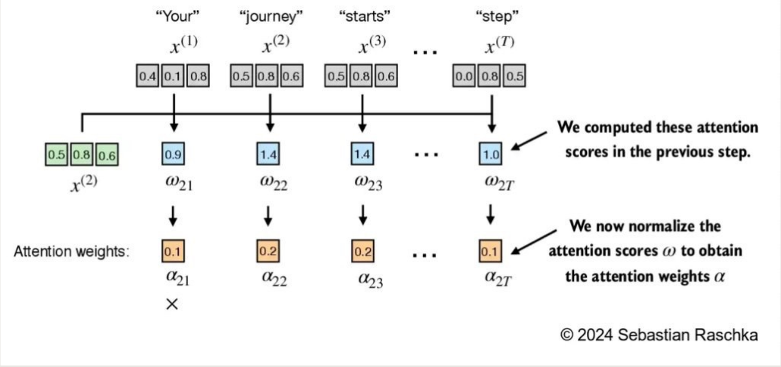
- Attention scores are calculated by performing the 'dot product' with the embeddings of each token with every other token in the sequence.


The embedded query token is one of the embedded input tokens (here, the query is the second token)
"Your"
"journey"
"starts"
"step"
Inputs:
$x^{(1)}$
$x^{(2)}$
$x^{(3)}$
$x^{(T)}$
0.4 0.1 0.8
0.5 0.8 0.6
0.5 0.8 0.6
0.0 0.8 0.5
0.5 0.8 0.6
0.9
1.4
1.4
1.0
Embedded query token: $x^{(2)}$
ω_{21}
ω_{22}
ω_{23}
ω_{2T}
Attention score between input $x^{(1)}$ and query $x^{(2)}$
Attention score between input $x^{(3)}$ and query $x^{(2)}$

© 2024 Sebastian Raschka

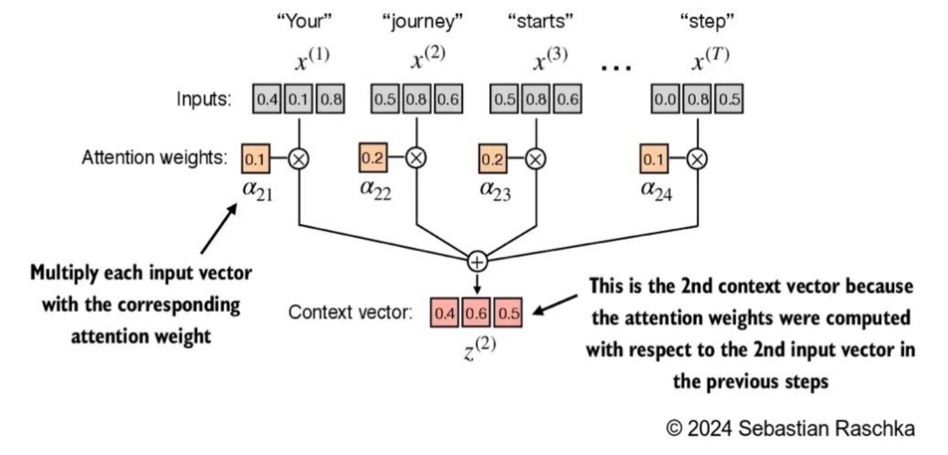
Attention Weights

- Weights calculated by normalizing the scores.
- Score / sum of all scores



© 2024 Sebastian Raschka

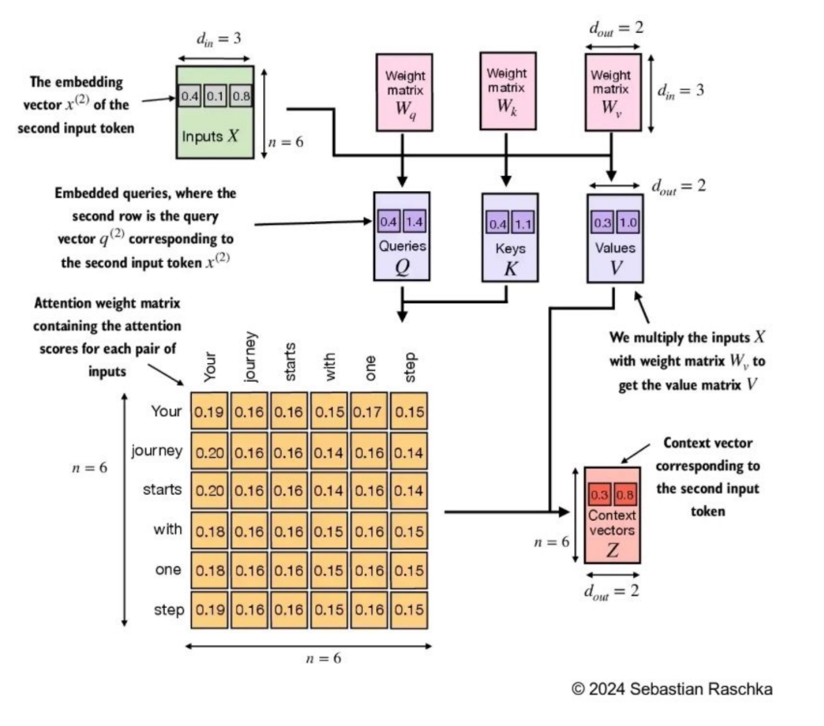
ATTENTION MECHANISMS



- The attention weights are used to calculate the context vector by scaling the input token embeddings by their respective weights and then summing them to result in a context vector.

ATTENTION MECHANISMS

- Trainable weights are implemented to help further the understanding of words within context.
- These weights include Query weights, Key weights, and Value Weights which change through training.
- There are several different methods for calculating weights that allow the models to be more stable.



The embedding vector $x^{(2)}$ of the second input token

Embedded queries, where the second row is the query vector $q^{(2)}$ corresponding to the second input token $x^{(2)}$

Attention weight matrix containing the attention scores for each pair of inputs

We multiply the inputs X with weight matrix W_v to get the value matrix V

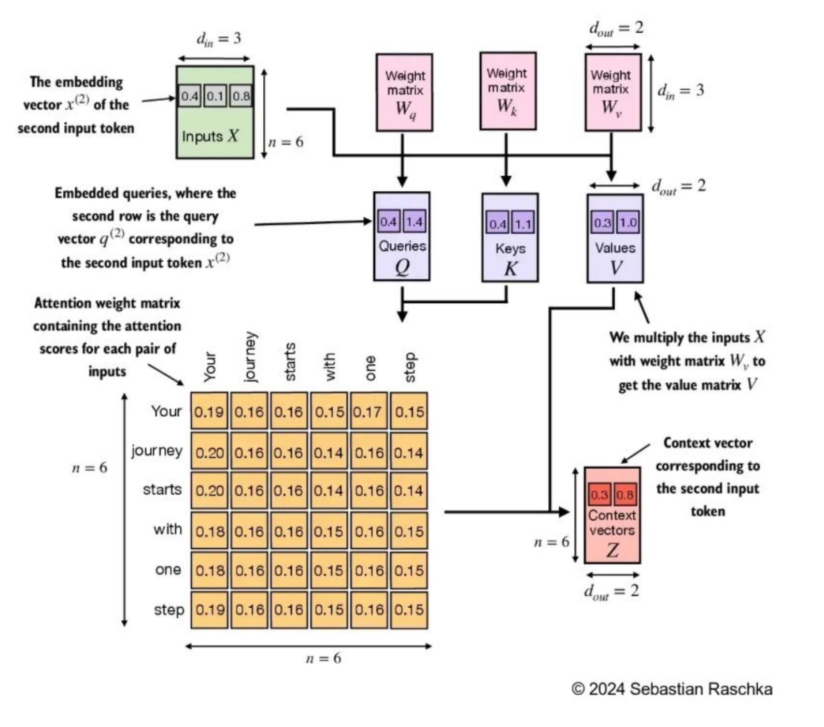
Context vector corresponding to the second input token

	Your	journey	starts	with	one	step
Your	0.19	0.16	0.16	0.15	0.17	0.15
journey	0.20	0.16	0.16	0.14	0.16	0.14
starts	0.20	0.16	0.16	0.14	0.16	0.14
with	0.18	0.16	0.16	0.15	0.16	0.15
one	0.18	0.16	0.16	0.15	0.16	0.15
step	0.19	0.16	0.16	0.15	0.16	0.15

© 2024 Sebastian Raschka

QUERY, KEY, AND VALUE WEIGHTS

- Query weights - 'What am I looking for?'
  - Represents the current tokens's 'question'
- Key weights - 'How relevant am I?'
  - *Represents an address for each token to compare to query*
- Value weights - 'What information do I carry?'
  - *Contains the actual content of a token that is passed to the next layer based on attention scores*



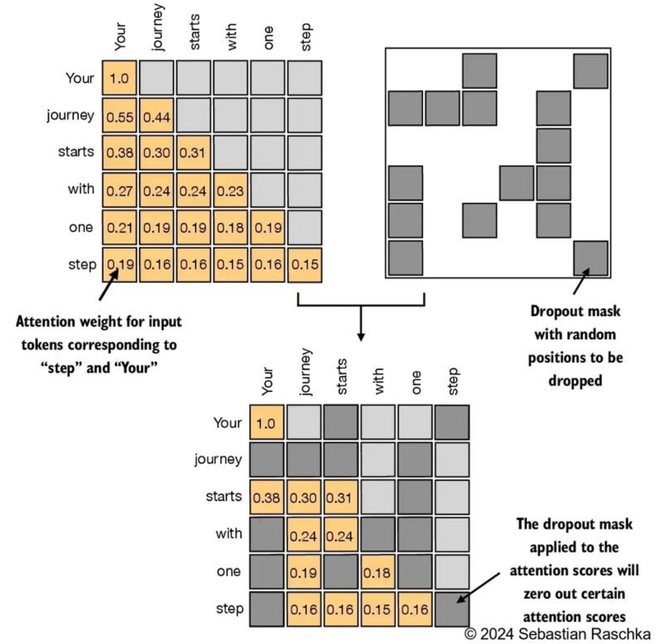
$d_{in} = 3$
The embedding vector $x^{(2)}$ of the second input token
0.4 0.1 0.8
Inputs X
$n = 6$
Weight matrix W_q
Weight matrix W_k
Weight matrix W_v
$d_{out} = 2$
$d_{in} = 3$
Embedded queries, where the second row is the query vector $q^{(2)}$ corresponding to the second input token $x^{(2)}$
0.4 1.4
Queries Q
0.4 1.1
Keys K
$d_{out} = 2$
0.3 1.0
Values V
Attention weight matrix containing the attention scores for each pair of inputs
We multiply the inputs X with weight matrix W_v to get the value matrix V
$n = 6$
Context vector corresponding to the second input token
0.3 0.8
Context vectors Z
$n = 6$
$d_{out} = 2$
$n = 6$

	Your	journey	starts	with	one	step
Your	0.19	0.16	0.16	0.15	0.17	0.15
journey	0.20	0.16	0.16	0.14	0.16	0.14
starts	0.20	0.16	0.16	0.14	0.16	0.14
with	0.18	0.16	0.16	0.15	0.16	0.15
one	0.18	0.16	0.16	0.15	0.16	0.15
step	0.19	0.16	0.16	0.15	0.16	0.15

© 2024 Sebastian Raschka

ATTENTION MECHANISMS

- Previously, all words in a sequence were being taken into account. However, in actual language we only pay attention to what comes before. Thus, Casual/Masked Attention is used to zero out all following tokens when calcuating context within the sequence.
- Dropouts also occur by randomly ignoring selected layers to help avoid overfitting. Though the remaining weights are doubled as a result



	Your	journey	starts	with	one	step
Your	1.0					
journey	0.55	0.44				
starts	0.38	0.30	0.31			
with	0.27	0.24	0.24	0.23		
one	0.21	0.19	0.19	0.18	0.19	
step	0.19	0.16	0.16	0.15	0.16	0.15

Attention weight for input tokens corresponding to “step” and “Your”

Dropout mask with random positions to be dropped

	Your	journey	starts	with	one	step
Your	1.0					
journey						
starts	0.38	0.30	0.31			
with		0.24	0.24			
one		0.19		0.18		
step		0.16	0.16	0.15	0.16	

The dropout mask applied to the attention scores will zero out certain attention scores

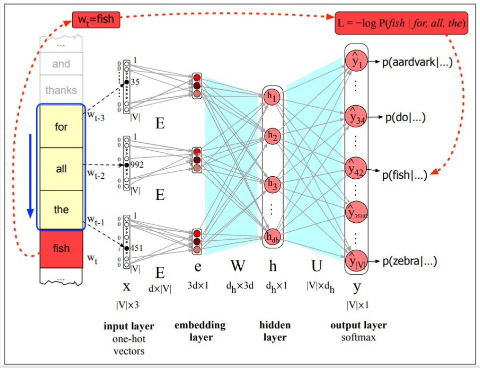
© 2024 Sebastian Raschka

DROPOUT

Dropout randomly "drops out" (sets to zero) certain neurons in a layer during training. It prevents overfitting by ensuring the model does not become overly dependent on specific neurons.

How it Works:

- For each training iteration (or mini-batch), a **dropout mask** is applied.
- Neuron outputs are set to zero with a probability p, the dropout rate.
- Remaining activations are **renormalized** to maintain the overall scale of the layer’s outputs.
- Dropout is **disabled during inference**, allowing the full network to be used.



FEEDFORWARD LAYER

What is a Feedforward Network?

- Fully connected submodule used in Transformer blocks.
- Consists of:
  - Input Linear Layer: Expands embeddings to higher dimensions.
  - Activation Function (e.g., GELU, ReLU, SwiGLU, etc.): Applies a non-linear transformation.
  - Output Linear Layer: Compresses embeddings back to original size.
- Purpose: Enhances the model's ability to learn richer representations while maintaining input-output dimension consistency.
- Common GPT Example: Embedding size expands from *768* → *3072* → *768.*

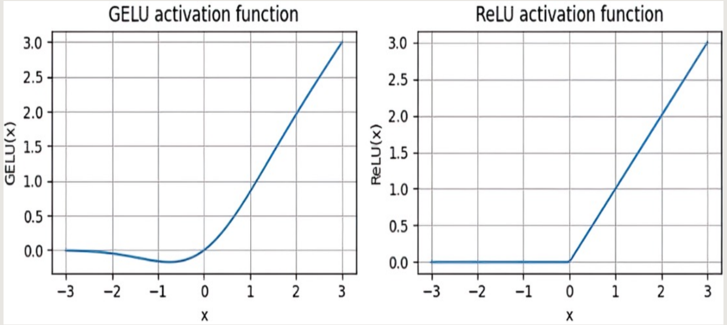
FEEDFORWARD LAYER

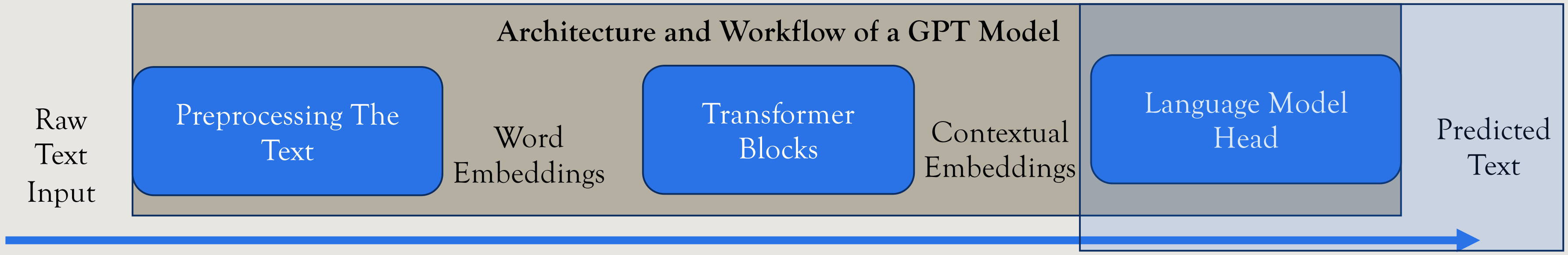
GELU vs. ReLU

- ***ReLU***:
  - Outputs max(0,x), ignoring all negative values.
  - Simple but can result in "dead neurons" (inactive units).
- ***GELU***:
  - Smooth function

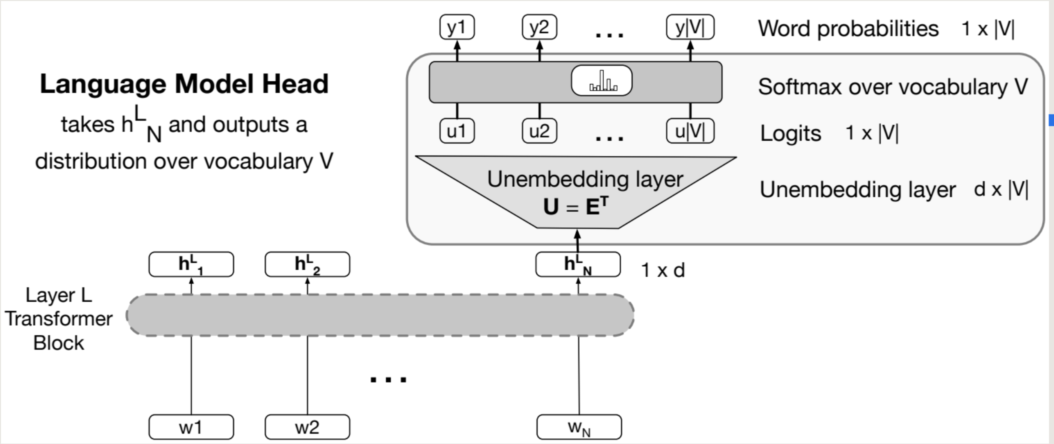
$$GELU(x) \approx 0.5 \cdot x \cdot \left(1 + tanh\left[\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715 \cdot x^3\right)\right]\right)$$

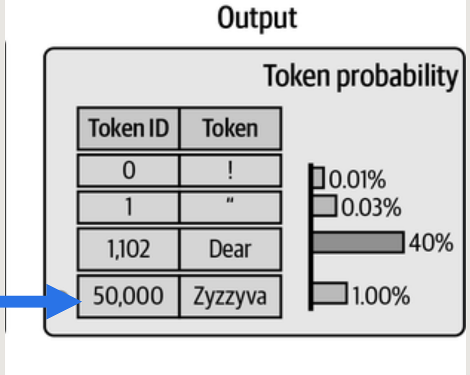
  - Allows small, non-zero contributions from negative inputs.
  - Enables smoother gradient flow, improving optimization in deep models.





LM HEAD OVERVIEW:


Language Model Head
takes h^L_N and outputs a distribution over vocabulary V
y1
y2
...
y|V|
Word probabilities 1 x |V|
Softmax over vocabulary V
u1
u2
...
u|V|
Logits 1 x |V|
Unembedding layer
$\mathbf{U} = \mathbf{E}^T$
Unembedding layer d x |V|
h^L_1
h^L_2
h^L_N
1 x d
Layer L Transformer Block
...
w1
w2
w_N



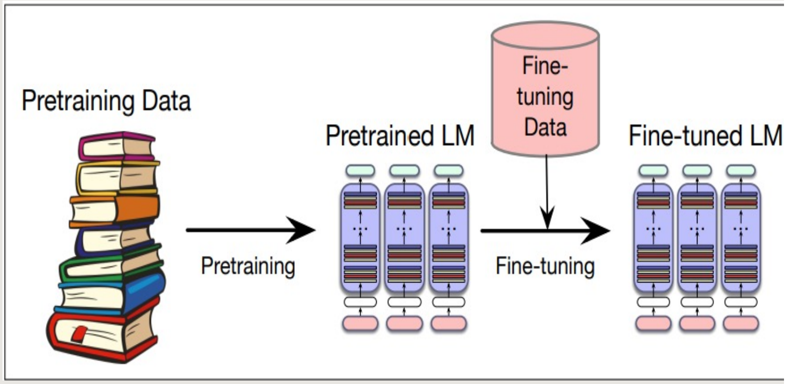
Output

Token probability

Token ID	Token	
0	!	0.01%
1	"	0.03%
1,102	Dear	40%
50,000	Zyzzyva	1.00%

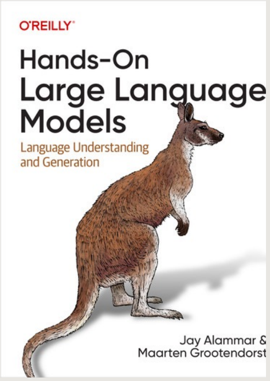
METHODS TO IMPROVE LLM TASK

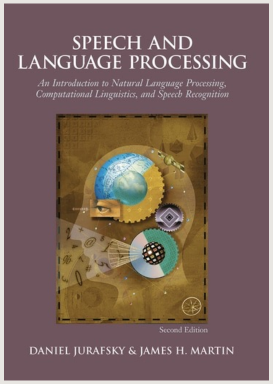
- Prompt Engineering
  - Guides model behavior without modifying its weights.
- Fine-Tuning
  - Adjusts model weights using domain-specific data
- Reinforcement Learning (RLHF - Reinforcement Learning with Human Feedback)
  - Trains the model using reward signals from human feedback.
  - Helps align responses with human preferences and reduces harmful outcomes outputs.
- Hyperparameter Tuning
  - Optimizes model settings like learning rate and batch size.
- Retrieval-Augmented Generation (RAG)
  - Enhances responses by retrieving external knowledge before generation.

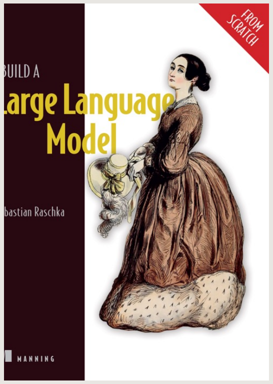


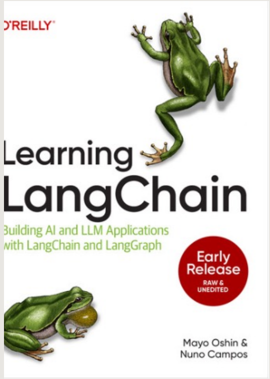
SOURCES

- Alammar, Jay, and Maarten Grootendorst. *Hands-on Large Language Models*. "O'Reilly Media, Inc.," 11 Sept. 2024.
- Jurafsky, Dan, et al. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Delhi, Pearson Education, 2008.
- Mayo Oshin, and Nuno Campos. *Learning LangChain*. O'Reilly Media, 30 Apr. 2025.
- Raschka, Sebastian. *Build a Large Language Model (from Scratch)*. Manning, 27 Aug. 2024.




SPEECH AND
LANGUAGE PROCESSING
An Introduction to Natural Language Processing,
Computational Linguistics, and Speech Recognition
Second Edition
DANIEL JURAFSKY & JAMES H. MARTIN





WHO ARE WE?

Jaylin Dyson, MSDS Student Researcher

Linkedin

Cameron Pykiet, MSDS Student Researcher

Linkedin

ACKNOWLEDGEMENTS

- We would like to acknowledge NSF HBCU-EiR 2302637 for funding our research.
- We would also like to thank South Dakota State University for organizing this event and giving us this opportunity.

- Thanks!
- Any Questions?