LARGE LANGUAGE MODELS

Lecture 2: Large Language Model (Basics)

- What is a Large Language Model (LLMs) ?
    - Deep Neural Networks designed to understand, generate and respond to human-like text e.g ChatGPT.
    - Trained on massive amounts of data.
- Why is it called “Large” Language Model
    - Large - Models have billions of parameters and it is growing tremendously
    - Language Models - These models do a wide range of NLP tasks; question answering, translation , sentiment analysis and much more
    - LLMs mostly deal with Language
- Why have LLMs become so popular now didn't we have the field of NLP before also?
    - LLM vs Earlies NLP models
        i) LLM - can do a wide range of NLP tasks
        ii) NLP models - designed for specific tasks like language translation etc
        iii) Another difference in a different context is; “Earlier language models could not write an email from custom instructions, a task which is trivial for modern LLMs”
- What makes LLMs so good? What is their Secret?
    - Transformer Architecture - What does it mean will be taught in subsequent lectures mostly giving attention to this particular paper released in 2017 - “Attention is all you need”.
- LLMs vs Generative AI vs Deep Learning vs Machine Learning
    - Try to paint a picture on what exactly are these and how would you define them and distinguish them appropriately - using venn diagram or concentric circles giving stark examples on how different and varied are they
    - Generative AI - Mix of LLMs and DL
        i) Using Deep Neural Networks to create new content such as text,images,videos etc. but LLMs only deal with text
    - LLMs represent a specific application of deep learning techniques leveraging their ability to process and generate human like text
- Application of LLMs
    - Chatbots/Virtual assistant
    - Machine translation
    - Novel text Generation
    - Sentiment Analysis
    - Content Creation

Lecture 3 : Stages of Building an LLM

- Creating an LLM
    - It involves two major processes which are: Pre-training and Fine-tuning
    - Pre-Training:

        - Training on a Large, Diverse Datasets - More generalized give insights on how Open AI trained GPT's and how pretraining essentially works
    - Fine-Tuning:
        - Refinement by Training on narrower dataset, specific to a particular task or domain. The reason is specific industries and specific domains require specific expertise in a particular knowledge base. No one uses generalized models. It is a matter of subject expertise. Other name for pretrained model is : Foundational Models
    - Pre-training and Fine-tuning Schematic
        - Step A: Data Collection (Corpus of raw data)
            - Internet text,books,media,research articles etc - Raw unlabeled text data -> trillions of words
        - Step B: Pre-training (Training the model)
            - LLMs getting trained on Unlabeled text data
            - Will have basic generalized capabilities will serve as foundational models Examples: ChatGPT, Gemini
        - Step C: Fine-tuning (Making it specifically talented in an area)
            - Pre-trained LLMs trained again on Labeled Data. Examples: Using ChatGPT to finetune and get a LLM expert in Law, Coding, Maths etc..

Lecture 4: Basic Introduction to Transformers

- Most Modern LLMs rely on the transformer architecture -> deep neural network architecture introduced in the 2017 paper
    - Attention is all you need - which is the foundation to the current GPT architecture
    - At that time it was proposed to do language translation tasks such as English to German translation and English to French translations.
- Simplified Toned-down Schematic of Transformer Architecture for Language Translation
    - Step 1: Input text to be Translated
    - Step 2: Preprocessing Step. Input step prepared for encoder
        - Done by a process called Tokenization for now, think of it as breaking down large sentences into smaller words and assigning each word a Unique number or ID.
    - Step 3 : Encoder - Most important step in the Architecture
        - Produces text encodings used by decoder
        - Vector Embedding - Every sentence is broken down into individual words till now and given numerical ID's but the most important part is we need to encode the semantic meaning between the words. Represent the tokens in a way that captures the semantic meaning
        - Very difficult task to perform
    - Step 4: Embeddings
        - Encoder returns embedded vectors as input to the decoder.
    - Step 5: Partial Output text
        - Model Completes translating one word at a time

    - Step 6: Preprocessing
        - Along with the vector embeddings even the partially translated text is fed into the decoder
    - Step 7: Decoder
        - Generates translated text one word at a time. by predicting the rest of the text.
        - Decoder is something similar like a neural network
    - Step 8: Output Layer
        - The complete output - Full translated text
- Transformer Architecture main items
    - Transformer Architecture predominantly consists of Encoder and Decoder
        - Encoder - Encodes input text into embedding vectors
        - Decoder - Generates output text from encoded vectors
    - A note on Self - Attention Mechanism
        - Key part of Transformers. Allows model to weigh importance of different words/ tokens relative to each other
        - Enables model to capture long range dependencies
        - Covered in detail later..
- Later Variations of Transformer Architecture
    - BERT - Predicts hidden/masked words in a given sentence
        - Really good at Sentiment analysis
    - GPT - Generates new word , predicting the next word
- Transformers vs LLMs
    - Not all Transformers are LLMs
    - Transformers can also be used for Computer Vision
    - Not all LLMs are Transformers also
    - LLMs can be based on recurrent and convolutional architectures as well.

Lecture 5: A Closer Look at GPT

- Zero Shot vs Few Shot Learning
    - Zero Shot - Ability to generalize to completely unseen tasks without any prior specific examples,The model performs a task without seeing any examples beforehand
        - Completing a task without an example
    - Few Shot- Learning from a minimum number of examples which the user provides as input
        - Completing a task with few provided examples
- Utilizing Large Datasets
    - Pretraining of GPT-3
    - A token is a unit of text which the model reads
    - The total pre-training costs for GPT's can be around millions of dollars
- GPT : Generative Pretrained Transformers
    - GPT 3 is a scaled up version of a transformer architecture
    - GPT models are simply trained on next word prediction tasks - unlabeled data

    - With this training, they can do a wide range of other tasks as well like translation,spelling correct etc..
    - Self-supervised Learning - self labeling
    - We don't collect labels for training data, but use the structure of the data itself
    - Auto regressive model: use previous outputs as inputs for future predictions
    - In GPT architecture there is no encoder we just have the decoder
        - Original Transformer : 6-encoder decoder blocks
        - GPT-3: 96 transformer layers, 175 B parameters
    - Emergent Behaviour:
        - Ability of a model to perform tasks that the model wasn't explicitly trained to perform

Lecture 6: Stages of building LLMs

- Stage 1: Building an LLM (Understanding basic mechanism)
    - Data Preparation and sampling
    - Attention Mechanism
    - LLM architecture
- Stage 2: Pretraining the LLM (Building a Foundational Model)
    - Training Loop
    - Model Evaluation
    - Load Pretrained weights
- Stage 3: Fine Tuning
    - Classifier
    - Personal Assistant

Lecture 7: LLM Tokenizer from scratch in Python

- Data Preparation and sampling
    - How do you prepare input text for training LLMs?
        - Step 1: Splitting text into individual words and subwords
        - Step 2: Convert these tokens and token IDs
        - Step 3: Encode token IDs into vector representation
- Now Let us look at Step 1: Tokenizing the Text
    - Dataset used: The Verdict by Edith Wharton
    - Convert large sentences into words or subwords
    - Python code has been implemented for that
- Now Let us look at Step 2: Tokens -> Token IDs
    - Complete training data set -> tokenized text -> Vocabulary
    - Vocabulary (A dictionary)
        - Sorting the tokens into alphabetical order and assigning each token a unique ID
        - So essentially a Vocabulary is a dictionary for all tokens. All tokens mapped to unique IDs
    - Process of Encoding:

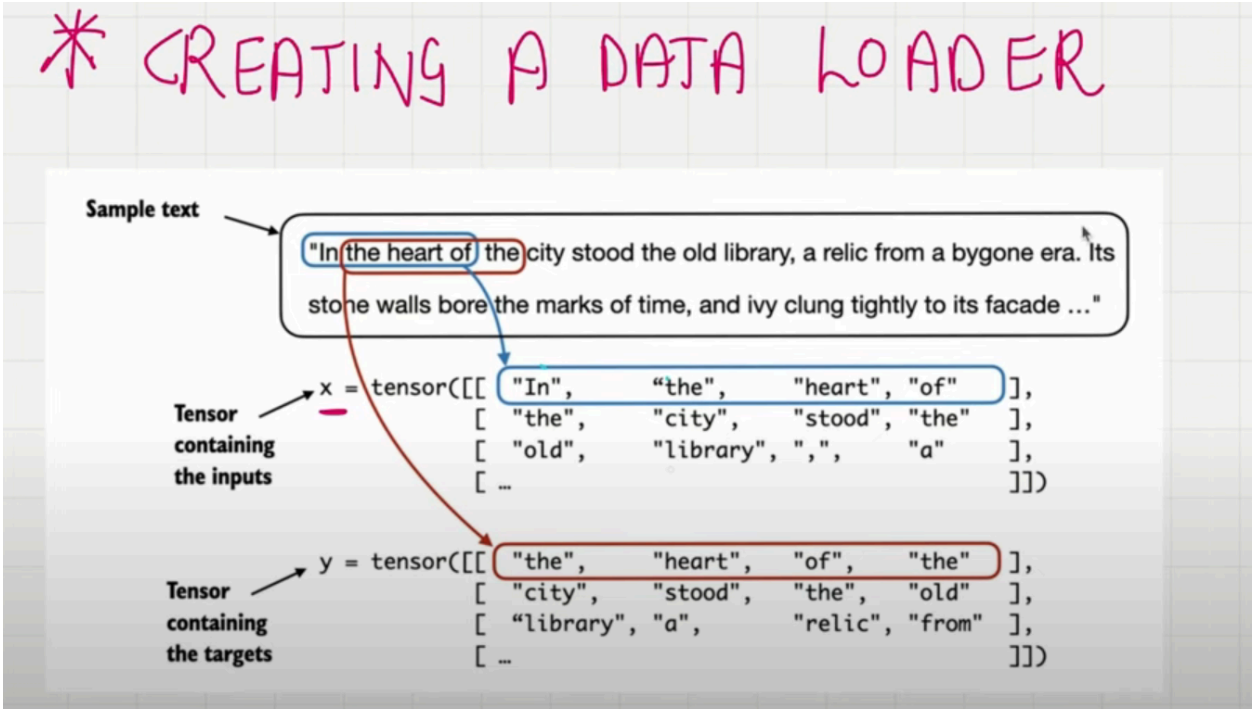
        - Sample text (The fox chased the dog) -> tokenized text(The|fox|chased|the|dog) -> token IDs (3 2 0 3 1)
    - Process of Decoding:
        - Exact vice-versa of Encoding Process

Lecture 8: Byte-pair encoding (BPE)

- Tokenization Algorithms:
    - 1. Word-based:
        - Every word in the sentence is a token.
        - Problems:
            - What to do with words that are out of vocabulary.
            - What to do with different meanings of similar words e.g:[boy,boys].
            - Root words aren't captured.
    - 2. Sub-word based:
        - BPE is an example of a sub-word based tokenizer.
        - RULES:
            - 1.Do not split frequently used words into smaller sub-words.
            - 2. Split the rare words into smaller,*meaningful* sub-words.
            - E.g. “boy” should not be split if it appears often. but if “boys” is rare split it into meaningful subwords: boy and s
        - Advantages:
            - 1. The subword splitting helps the model learn that different words with same root word like “token” , “tokens” and “tokenization” are similar in meaning
            - 2. It also helps the model learn that “tokenization” and “modernization” are made up of different root words but have the same suffix “ization” and are used in the same syntactic situations.
    - 3. Character based:
        - Every character in a word is a token. So vocabulary is very small
        - Problems:
            - The meaning associated with words is completely lost.
            - Also the tokenized sequence is much longer than the initial raw text e.g. dinosaur in a word-based tokenizer has only one ID. but for character based it splits it into 8 tokens id’s.
- BPE algorithm:
    - Most common pair of consecutive bytes of data is replaced with a byte that does not occur in data
    - Provide an apt example
- How is the BPE algorithm used for LLMs
    - BPE ensures that most common words in the vocabulary are represented as a single token while rare words are broken down into two or more subwords

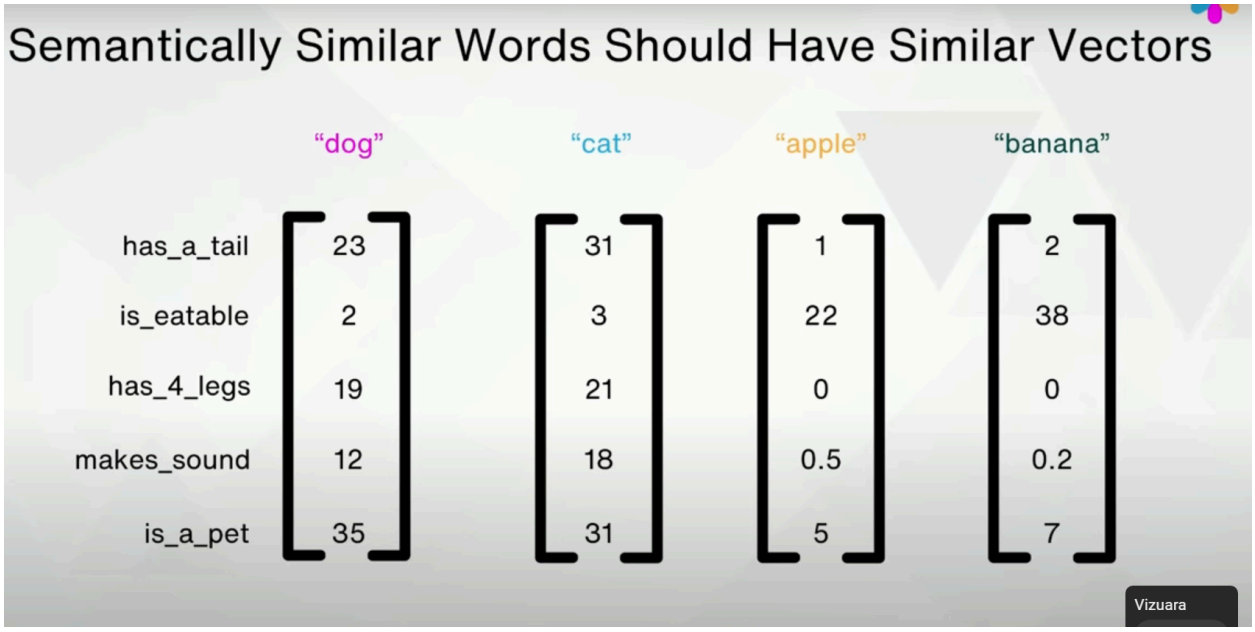
Lecture 9: Create Input - Output (Target) Pairs

- The last step before we create vector embeddings is to create input-target pairs.
- How does input-target pair look like give noteworthy examples
    - Given a text sample
    - Extract input blocks as subsamples that serve as input to the LLM
    - The LLMs prediction task during training is to predict the next word that follows the input block
    - During training we mask out all words that are past the target.
- So this is the reason they are called *self-supervised* models and *auto-regressive* in nature.
- Learned about context size,strides and batch size and implementation of this using **datasets dataLoaders** package from pytorch.
    - Context size = Maximum length of the input the LLMs sees at one time.
    - To implement efficient dataLoaders we collect inputs in a tensor x, where each row represents one input context. The second tensor y contains the corresponding prediction targets (next words), which are created by shifting the input by one position.
    - So essentially each input-output pair has 4 computations or prediction tasks here 4 = context size for example if context size is 6: there will be six prediction tasks because input size can start from 1 to at-most (context size) i.e. 6.



Lecture 10: Token Embeddings

1. Conceptual understanding of why token embeddings are needed?
    - Representing words numerically
        - Computers need numerical representation of words
        - How can we represent words in numbers?
            - Can we just assign random numbers to each word?
                - Cat -> 34
                - Book -> 29
                - Tablet -> 47
                - Kitten -> 354
            - The problem with random numbers assignment is that: cat and kitten here are semantically related. However the associated numbers cannot capture this relationship.
        - Semantically Similar words should have Similar vectors.



- Vectors - can capture those semantic relationships.
- Now how to come up with these vector embeddings? - Training a Neural network to create vector embeddings

2. Small hands on demo - playing with token embeddings - google's pre-trained word2vec-googlenews - 300

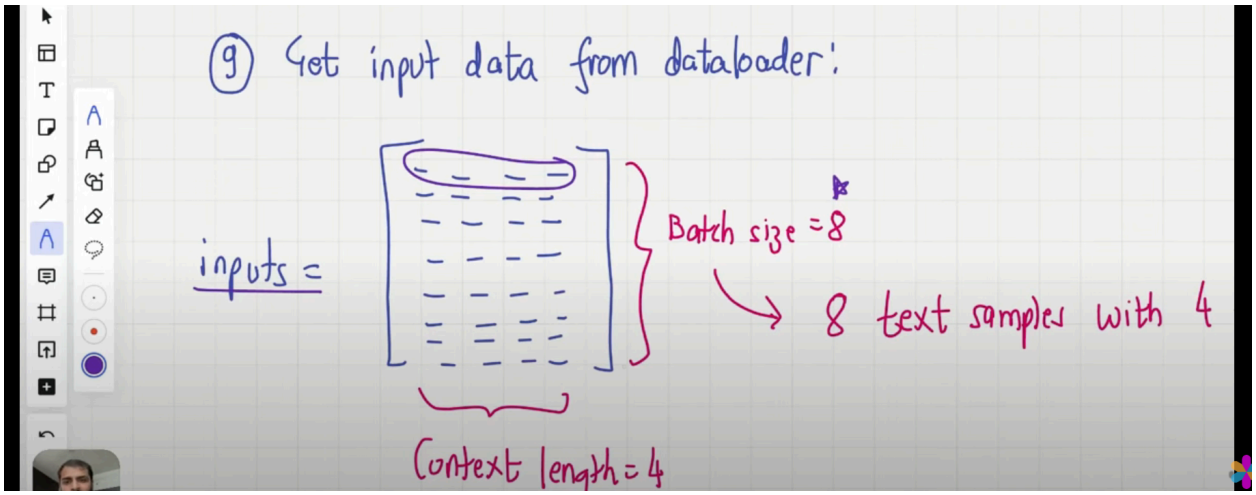
3. How are token embeddings created for LLMs
    - Initialize embedding weights with random values
    - This initialization serves as the starting point for the LLM learning process

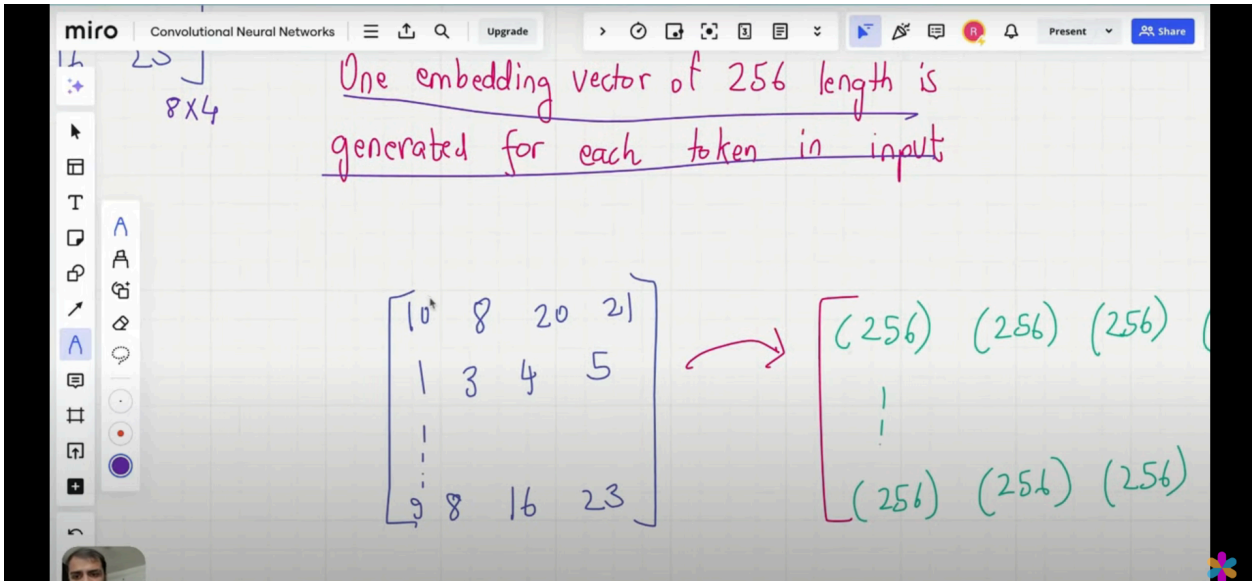
- The embedding weights are optimized as part of the LLM training process
- Create an Embedding layer weight matrix for e.g. when GPT-2 was trained it had: 50257 tokens or vocabulary and for each vocabulary a vector dimension of 768. So 50257(Rows) * 768(Columns) Vector dimension * Vocabulary size
- The embedding layer is a lookup operation that retrieves rows from the embedding layer weight matrix using token IDs

Lecture 11: Positional Embeddings

- In the embedding layer, the same token ID gets mapped to the same vector representation, regardless of where the token ID is positioned in the input sequence.
    - E.g. The **cat** sat on the mat, on the mat the **cat** sat
    - In sentences there are meanings between different words - which we captured via vector embeddings but the **position** of the words in the sentence also matters.
    - It is helpful to inject additional position information to the LLM

- Two types of positional embeddings:
    - Absolute:
        - For each position in input sequence, a unique embedding is added to the token embeddings to convey its exact location
        - Token Embeddings + Positional Embeddings = Input Embeddings
        - The positional vectors have the same dimension as the original token embeddings
    - Relative:
        - The emphasis is on the relative position or distance between tokens. The model learns the relationship in terms of “How far apart” rather than at which exact position.
        - Advantage: model can generalize better to sequences of varying lengths, even if it has not seen such lengths during training.
- Both types of positional encoding enable LLMs to understand the order and relationship between tokens, ensuring more accurate and context aware predictions
- The choice between the two depends on specific application and nature of data being processed
- Even the positional embeddings like the vector embeddings are optimized during the training process. The optimization is part of the model training itself.
- Visualizing the input data from data loader:





- Context_Length: A maximum of 4 tokens can be passed as Input at a time
- Batch_size: Parameters update after processing one Input box of size 8 - rows
- Each row has 4 tokens and a total of 8 rows are present - 32 tokens in one batch

Lecture 12: The Entire Data Preprocessing Pipeline of Large Language Models (Summary)

- Large Language Models predominantly deal with text and sentences but we can't directly feed them into our language models Hence the Data Preprocessing requirement.
- 4 - Fundamental Steps are present in the Data Preprocessing pipeline:
    - 1) Tokenization
    - 2) Token Embeddings
    - 3) Positional Embeddings
    - 4) Input Embeddings = Token Embeddings + Positional Embeddings

---

Lecture 13: Introduction to Attention Mechanism in LLMs

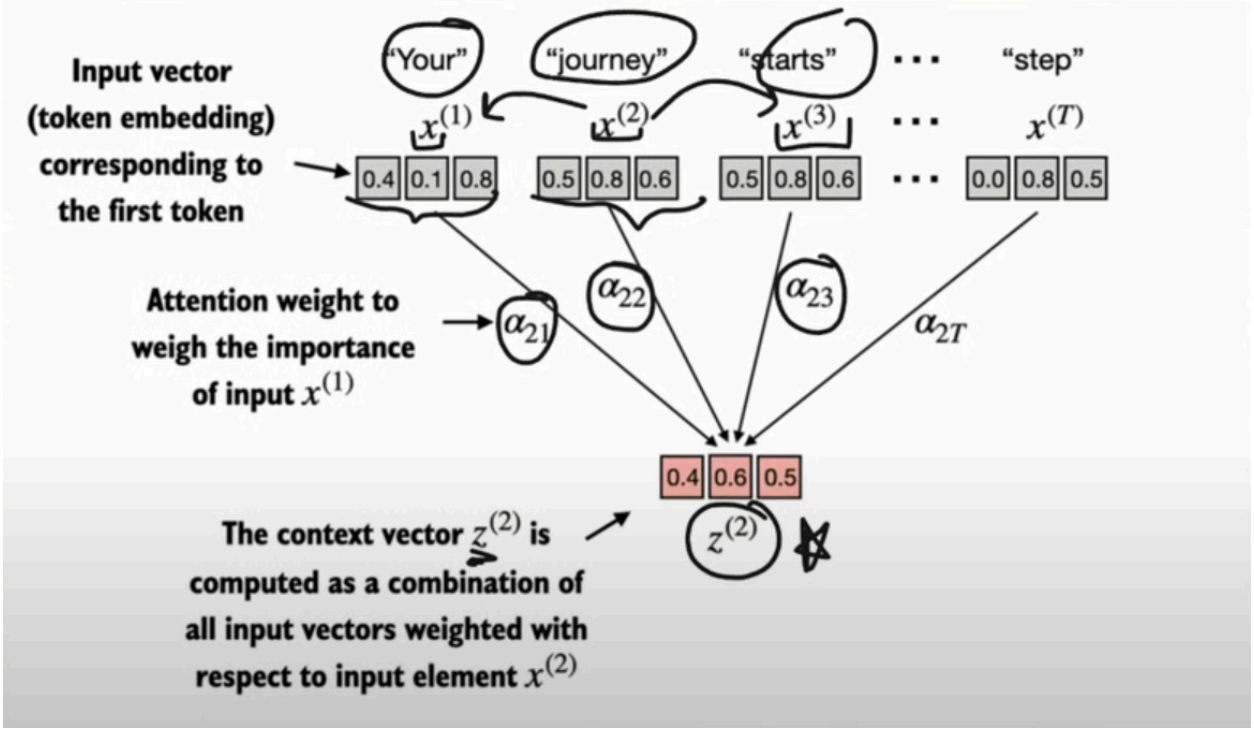
- 4 Types of attention Mechanism
    - Simplified Self Attention
    - Self Attention
    - Causal Attention
    - Multi-head Attention
- To address this issue that we cannot translate text word by word, it is common to use a neural network with two submodules:
    - Encoder:
        - Reads and processes text
    - Decoder:
        - Translates the text
- Before transformers , Recurrent Neural Networks (RNNs) were the most popular encoder-decoder architecture for language translation
- Here is how the encoder - decoder RNN works:
    - Input text -> Encoder (Processes input text sequentially) -> Updates hidden state at each step (hidden state is the one which give memory to the neural network so it can understand context) -> Final hidden state (encoder tries to capture sentence meaning) -> Decoder (uses the final hidden state to generate translated text one word at a time)
- The encoder processes the entire input text into a hidden state (memory cell). Decoder takes this hidden state to produce the output .
- Issue with RNNs: RNN cannot directly access earlier hidden states from the encoder during the decoding phase. It relies solely on the latest hidden state. This leads to a loss of context, especially in complex sentences where dependencies might span long distances.
- Capturing data dependencies with attention mechanisms
    - RNNs work fine for translating short sentences, but don't work for long texts as they don’t have direct access to previous words in the input.
    - One major shortcoming in this approach is that : RNNs must remember the entire encoded input in a single hidden state before passing it to the decoder.
- Attention mechanism:
    - At each decoding step, the model can look back at the entire input sequence and decide which parts are most relevant to generate current word
    - Dynamic focus on different parts of input sequence allows models to learn long range dependencies more effectively.
- Self Attention
    - In Self-Attention the “ self ” refers to the mechanisms ability to compute attention weights by relating different positions in a single input sequence
    - It learns the relationship between various parts of the input itself such as words in a sentence
    - This is in contrast to traditional attention mechanisms where the focus is on relationships between elements of 2 different sequences

- **Attention** is like asking: *"Which words from the input should I look at while decoding?"*

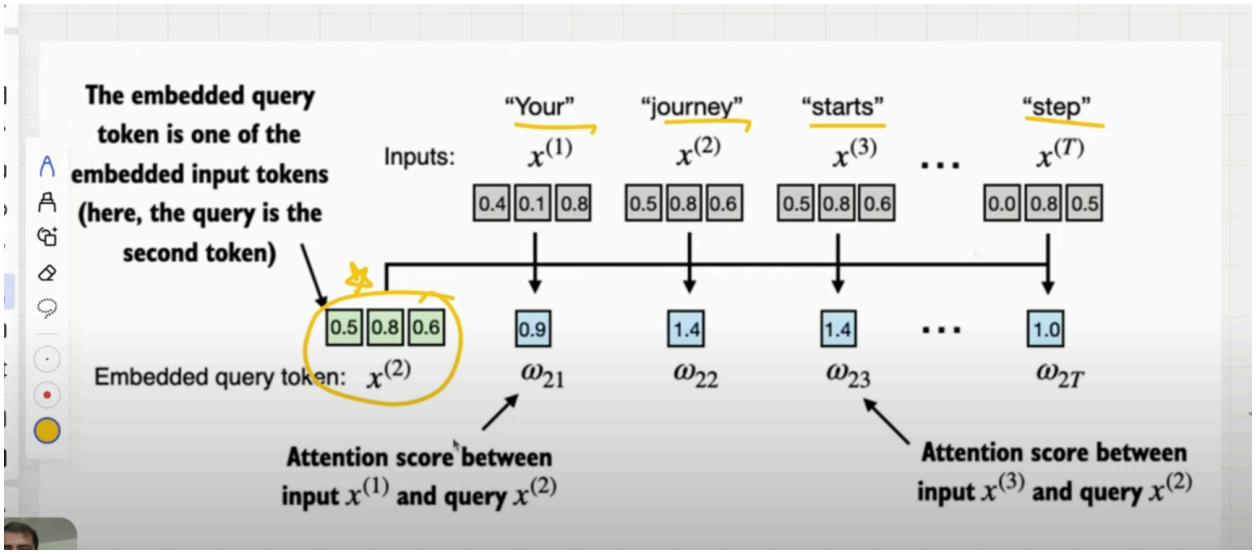
- **Self-Attention** is like asking: *"Which other words in this sentence help me understand this word?"*

Lecture 14: Simplified Self- Attention Mechanism (W/o Trainable weights)

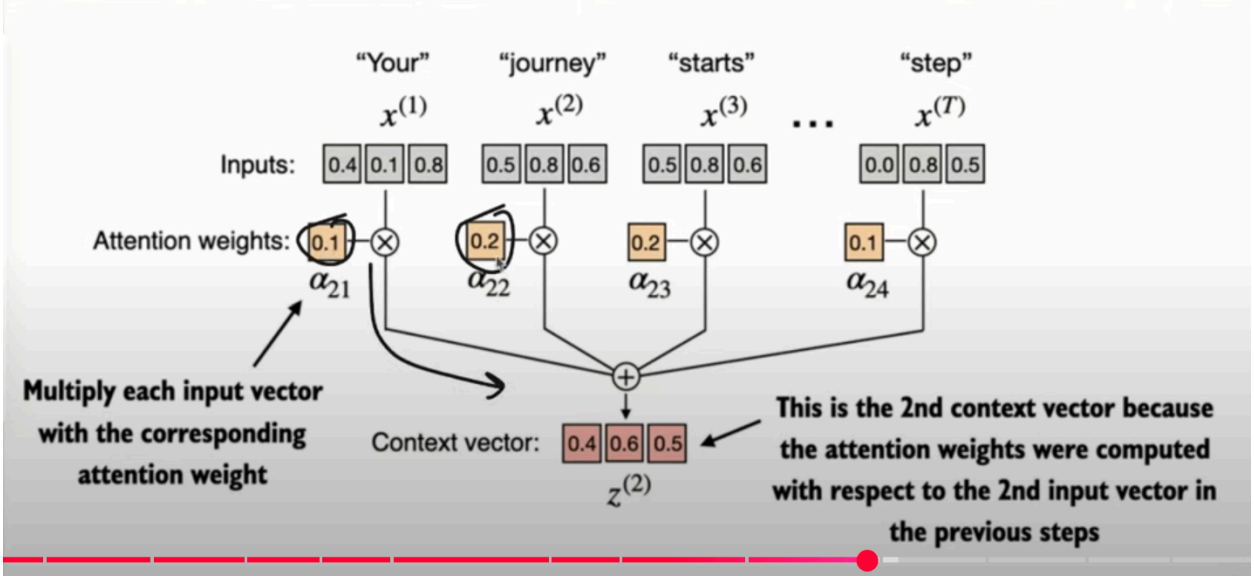
- Until Now we have covered how raw text are converted into vector embeddings through a series of process:
    - 1) Break down the whole text into tokens (BPE) - a subword tokenizer.
    - 2) Convert the tokens into token ID's
    - 3) Convert the token ID's into Vector Embeddings which contain both the semantic meaning of the word and the position of the word
- But the main problem arises when:
    - The cat that was sitting on the mat, which was next to the dog jumped → here the LLMs must understand that cat was the one which jumped, not the dog.
    - Taking an example sentence: "Your Journey Starts with one step"
        - The vector embedding captures the meaning of the word "Journey" but it does not capture how this word relates to the other words given in the sentence.
        - Or even better, we fail to capture how much attention we should give to each word relative to the word "journey".
        - So the most important part i.e Context is missing till now.
        - Goal: Take the Embedding Vector -> Context Vector (Enriched version of Embedding Vector).
        - Context Vector: Not only contains the semantic meaning but also how that particular word (token) is related to the other words in the sentence.



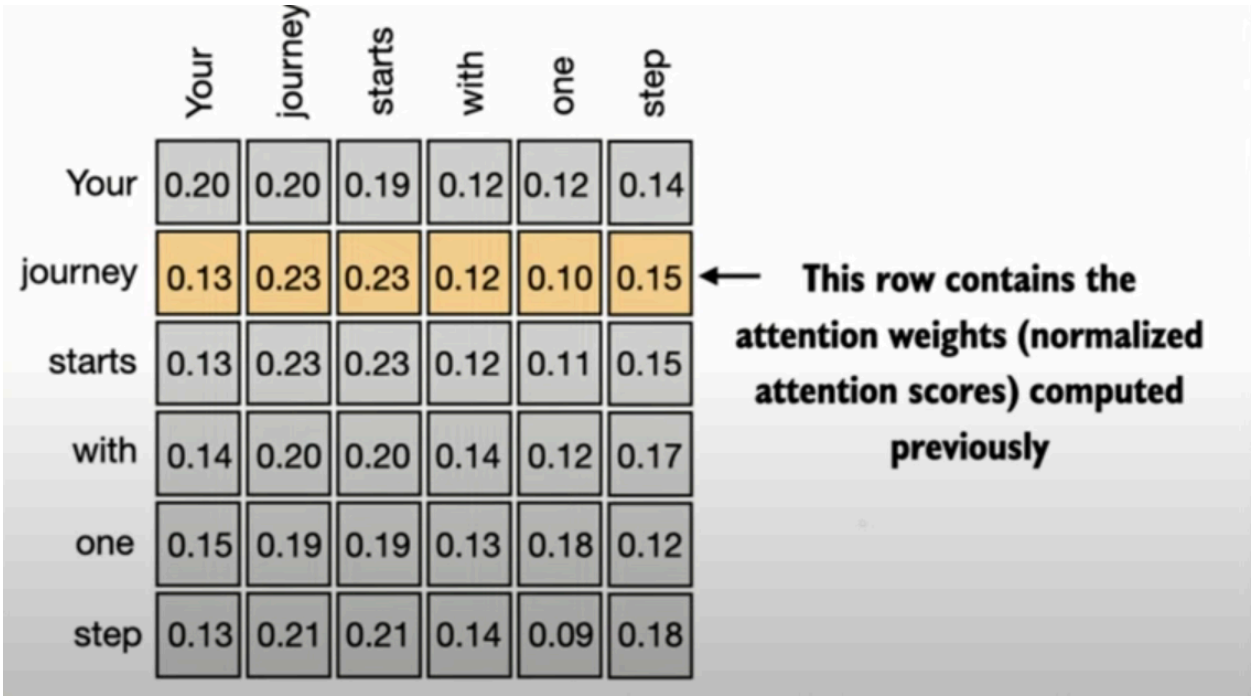
- Goal of self-attention:
  - Calculate a context vector (Z^i) for each element (X^i)
- "Your Journey Starts with one step"
- 1) Input Sequence (X):
  - X^1,X^2,...X^t - Token embeddings representation.
- 2) Focusing on Second Element (X^2) = Journey
  - Corresponding context vector is Z^2
  - The element we are looking at currently is also called "query".
  - Z^2 is an embeddings which contains information about X^2 and all other input elements X^1 to X^T
- 3) Later we will add trainable weights that help the LLM learn to construct these context vectors; so that they are relevant for the LLMs to generate the next token
- 4) First step of implementing self-attention is to compute the intermediate values W, also referred to as attention scores.



- 5) The intermediate attention scores are calculated between the query token and each input token - Using DOT PRODUCT
    - Dot Product quantifies how much two vectors are aligned
    - Higher the dot product, higher the similarity and attention scores between the two elements
- 6) After dot product we enter the process of normalization
    - This improves interpretability and makes sure value is between 0&1 because models can be trained better and can be used to tell percentage wise how much attention should journey give to “your” or “step” just like that.
- 7) After computing the normalized attention weights, we calculate the context vector Z^2 by multiplying the embedded input token X^i, with the corresponding attention weights and then summing the resultant vectors



- 8) Computing attention weights for all input tokens:
    - We can extend similar computation as before to calculate attention weights and context vectors for all inputs.
    - NOTE: **Attention Weights:** are the normalized version of Attention Scores

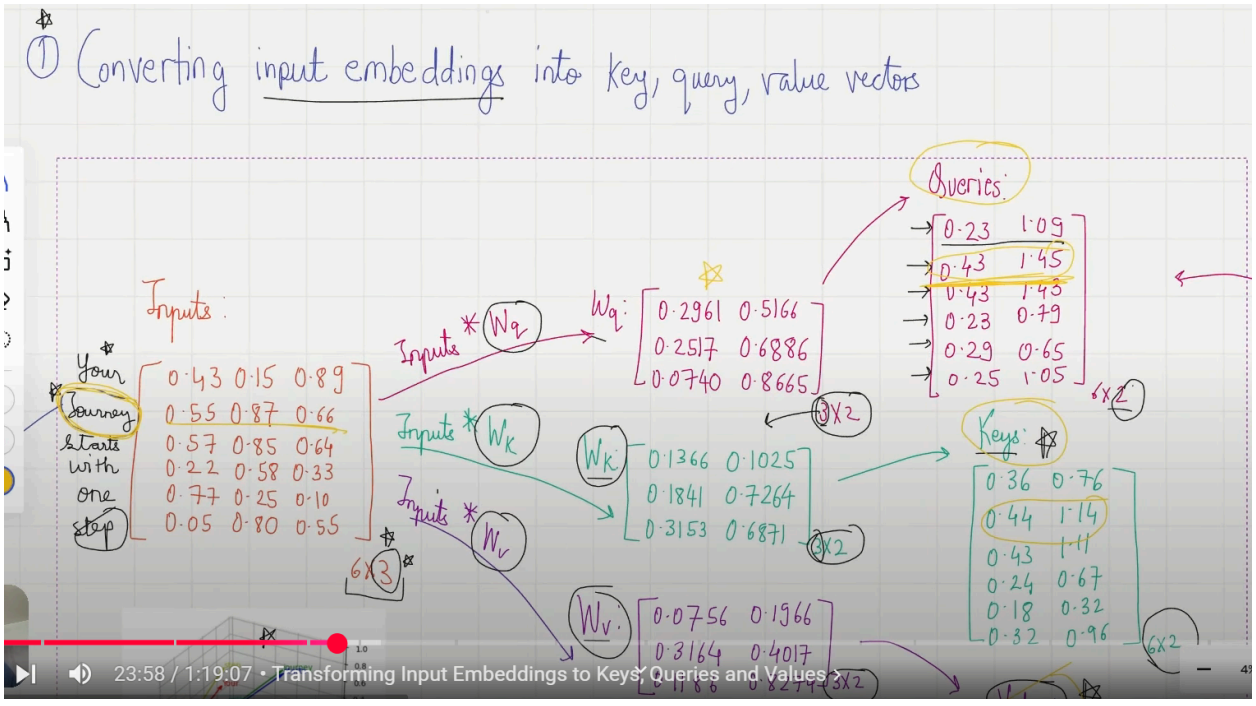


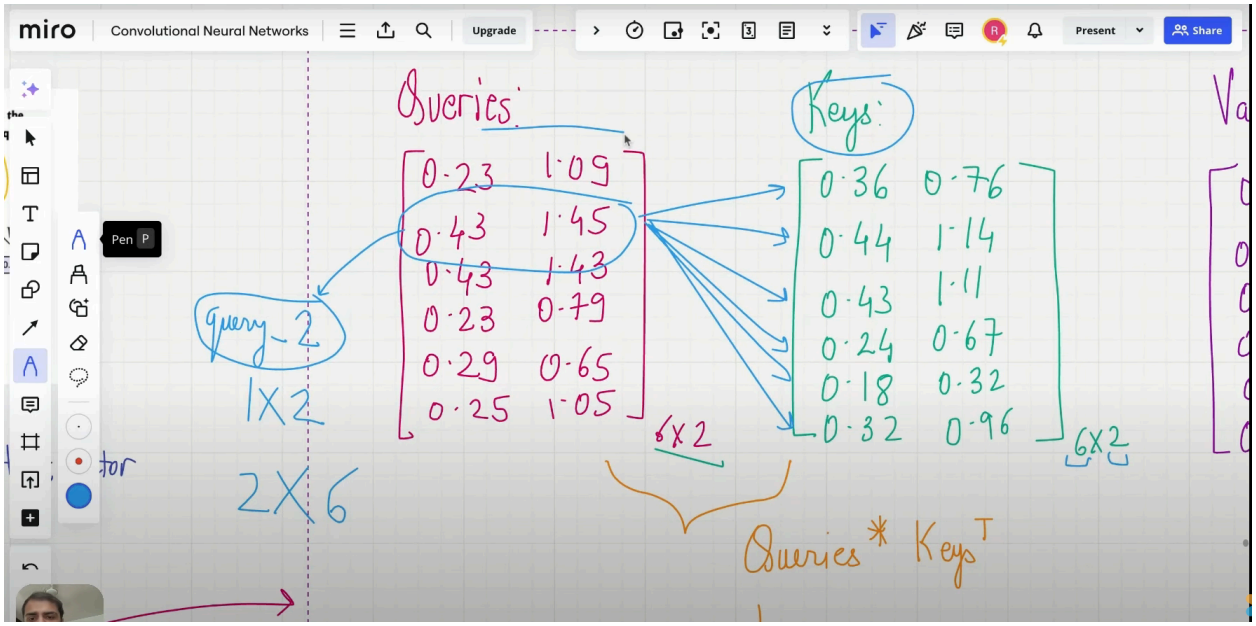
- Summary:
    - 1) Compute attention Scores - dot products between input embeddings
    - 2) Compute attention Weights - Normalizing using soft max function

- 3) Compute context vectors - The context vectors are computed as a weighted sum over the inputs

Lecture 15: Self-Attention Mechanism With Trainable Weights

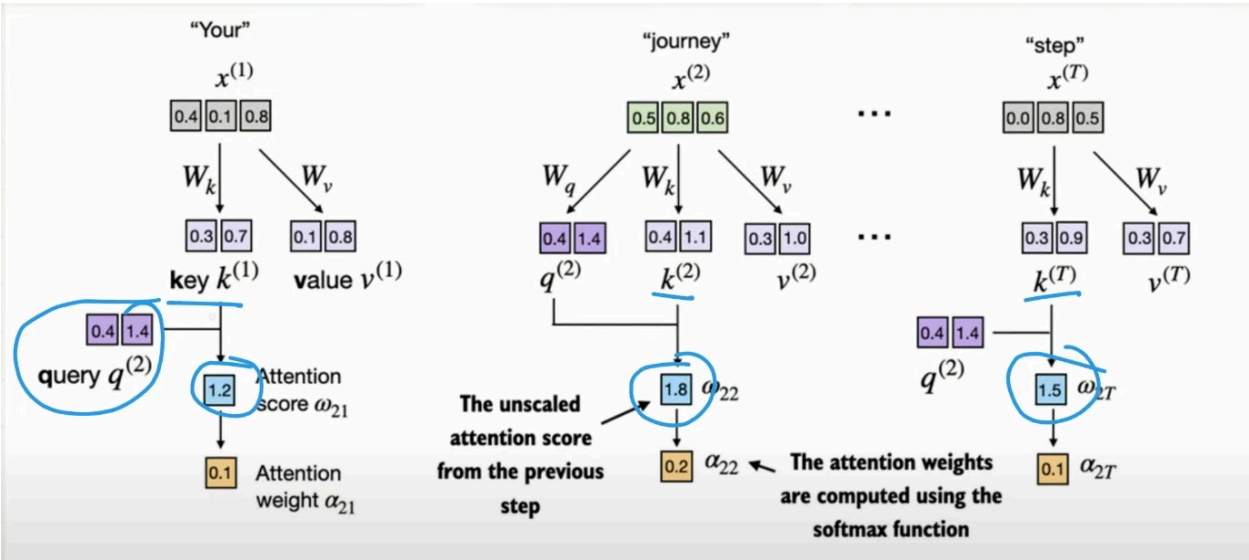
- In this section, we will learn about the self-attention mechanism used in the original transformer architecture, the GPT models and most other popular LLMs
- Self-attention mechanism is also called *"scaled dot-product attention"*
- Steps:
  - We want to compute context vectors as weighted sums over the input vectors specific to a certain input element
  - We will introduce weight matrices that are updated during model training
  - These trainable weight matrices are crucial so that the model can learn to produce good context vectors
  - We will implement the self attention mechanism step by step by introducing 3 trainable weight matrices: W_q, W_K, W_V
    - These 3 matrices are used to project the embedded input tokens X^i into Query, Key, and Value vectors



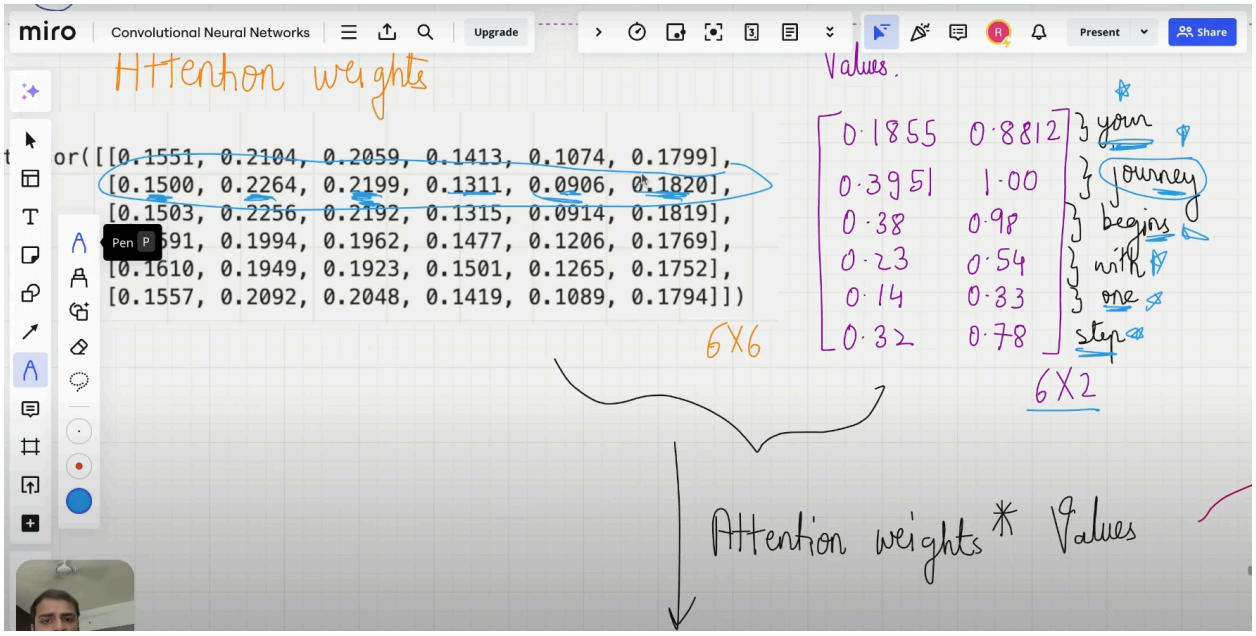


Step by Step rules:

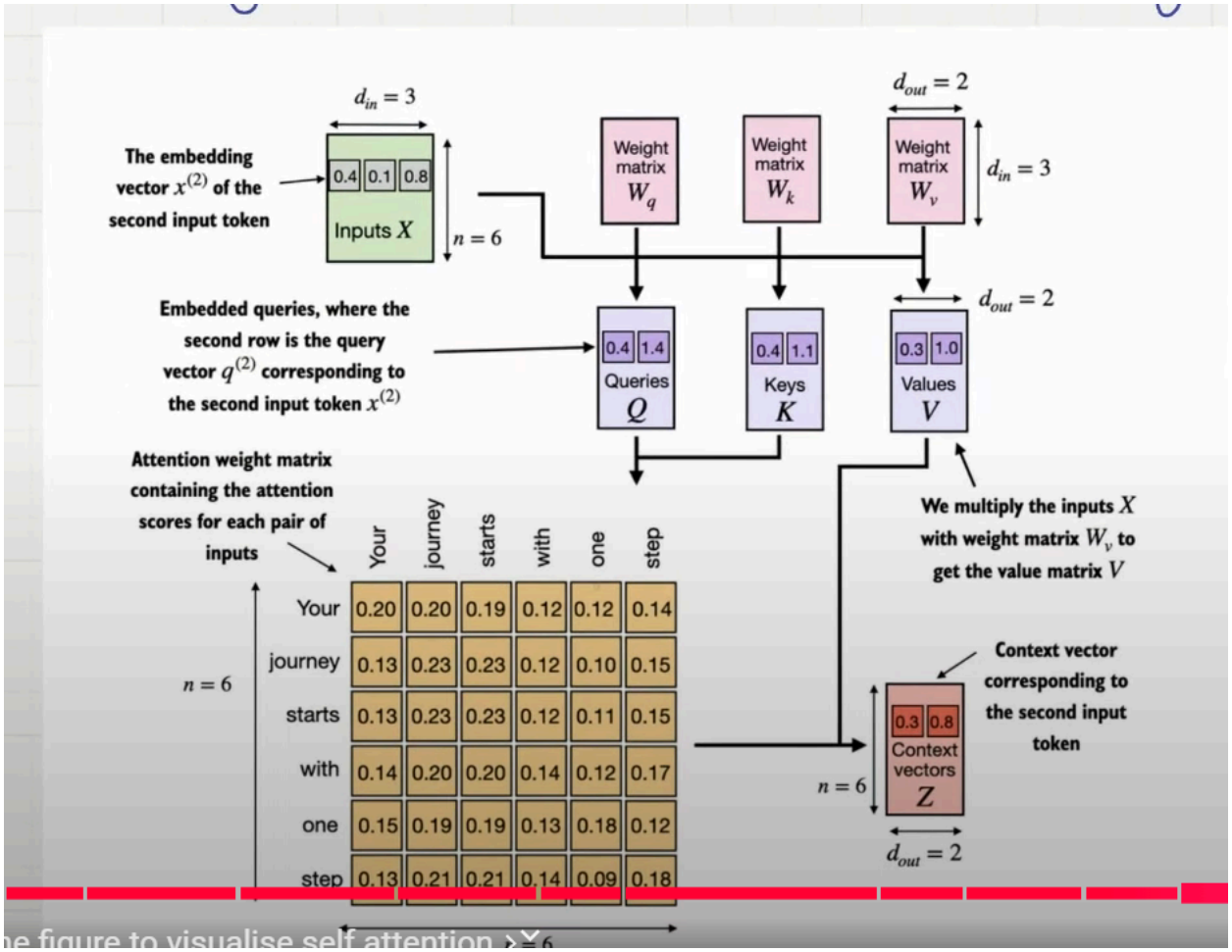
- We convert the Input Embedding Matrix into 3 Matrices (Key,Query,Values) by multiplying it with Weight matrices W_k,W_q,W_v
- Now once we get these 3 matrices we want to find the attention_scores for all the tokens
    - We do that by getting the Dot Product between Queries and Key Matrix.
- Once we get the attention scores by the dot product operation we scale the attention score matrix by square root of D where D is the dimension of keys here it is 2 and this is an important step to make sure learning is stable.
- After that we normalize the attention score matrix by softmax to get the attention weight matrix which is interpretable.
- Whatever done until now:



- Now Multiply the Attention weight matrix with the Value Matrix which will give you the **Context Vector** for each token.

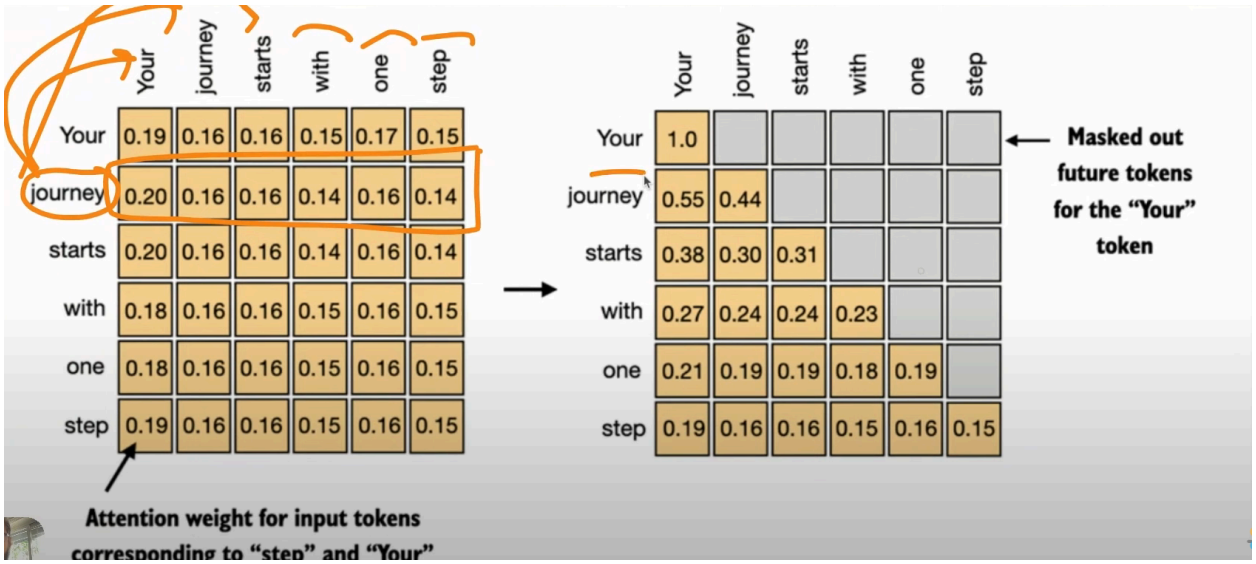


Final Schema:

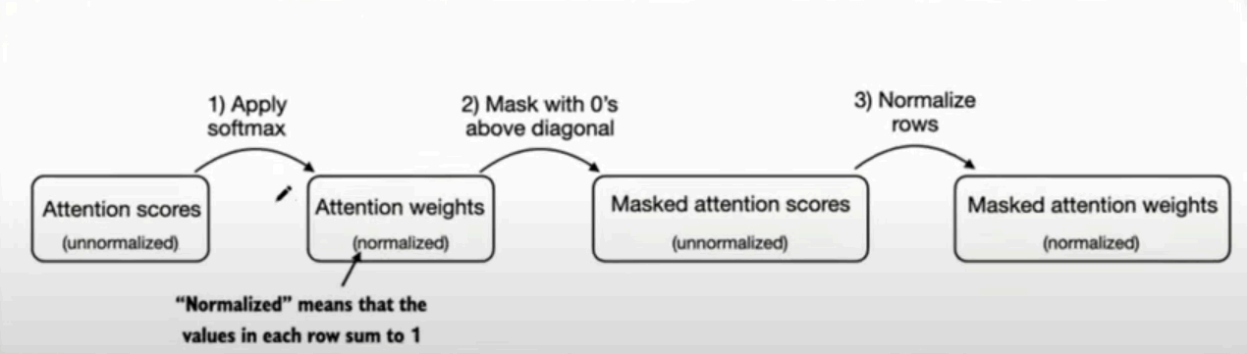


Lecture 16: Causal Attention

- Causal attention, also known as masked attention is a special form, of self attention
- It restricts the model to only consider previous and current inputs in a sequence, when processing any given token.
- This is in contrast to the self - attention mechanism, which allows access to the entire input sequence at once.
- When computing attention scores, the causal attention mechanism ensures that the model only factors in tokens that occur at or before the current token in the sequence
- To achieve this in GPT like LLMs, for each token processed, we mask out the future tokens, which come after the current token in the input text



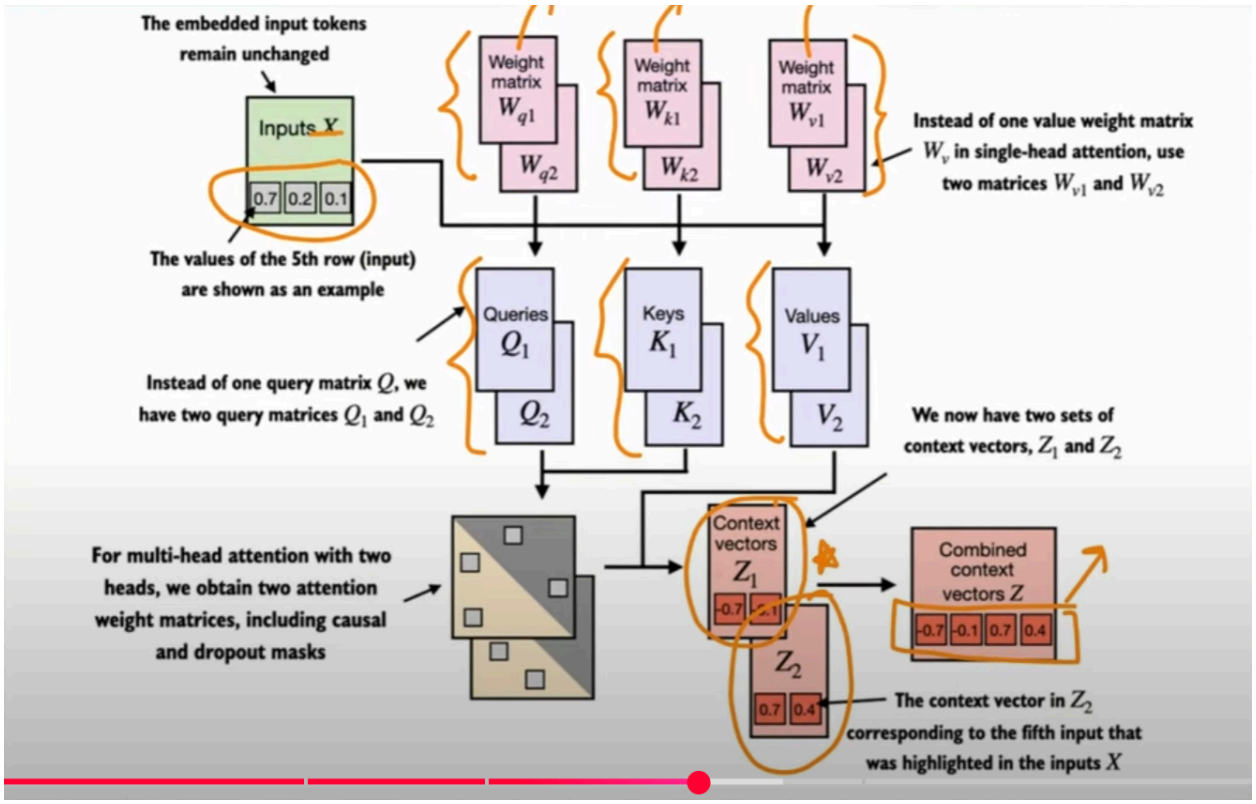
- Applying Causal Attention Mask
    - Get Attention weights -> Zero out the elements above the diagonal and normalize the resulting matrix. normalize here is to make sure the sum adds up to 1.
- Flow Map:



- But there is Problem. The Entire Point to mask out future elements is that these elements should not influence the current token. But we are already getting influenced because of the fact that in the 1st step itself we are applying softmax to get attention weights which already uses future elements which we want to mask - this is called the **Data Leakage Problem**
- We need to tackle this problem by
    - Attention Scores → Upper Triangular Infinity Mask → Softmax

Lecture 17: Multi Head Attention Part 1:

- The term “multi - head” refers to dividing the attention mechanism into multiple heads → each operating independently
- Stacking multiple single head attention layers
- Implementing multi-head attention involves creating multiple instances of the self-attention mechanism; each with its own weight and then combining their outputs
- This can be computationally intensive, but it makes LLMs powerful at complex pattern recognition tasks
-

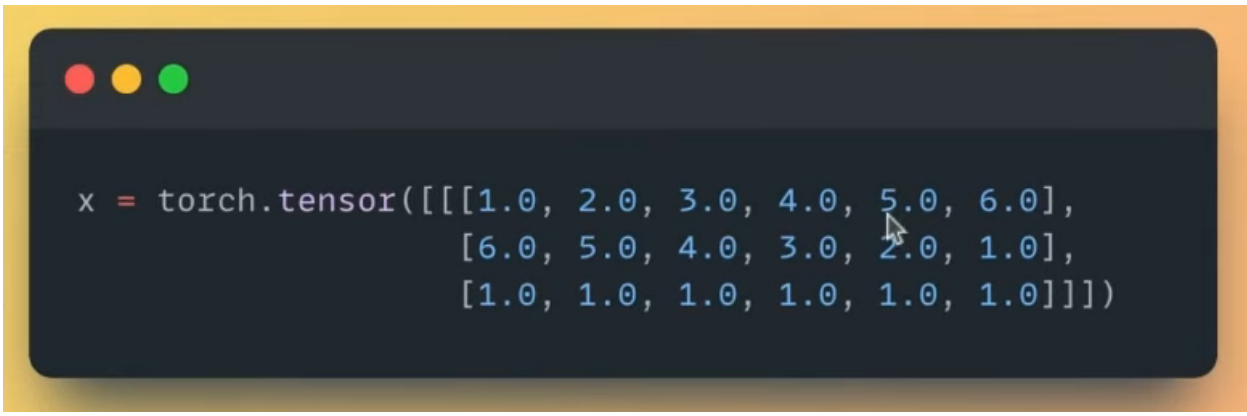


- Run the attention mechanism multiple times (in parallel) with different, learned linear projections:
    - The result of multiplying input data (like query, key, value vectors) by a weight matrix

Lecture 18: Multi Head Attention Part 2:

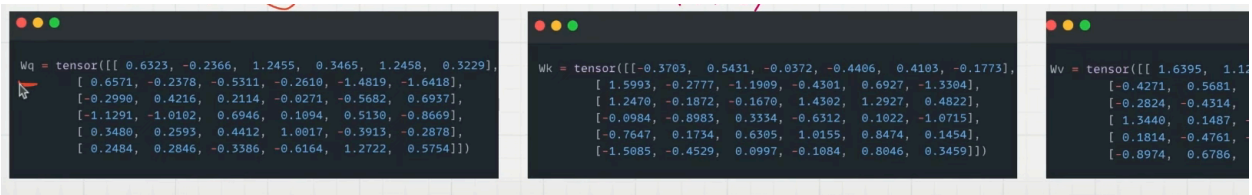
Let us take a simple example:

- Step 1: start with the Input.
    - b,num_tokens,d_in =(1,3,6)

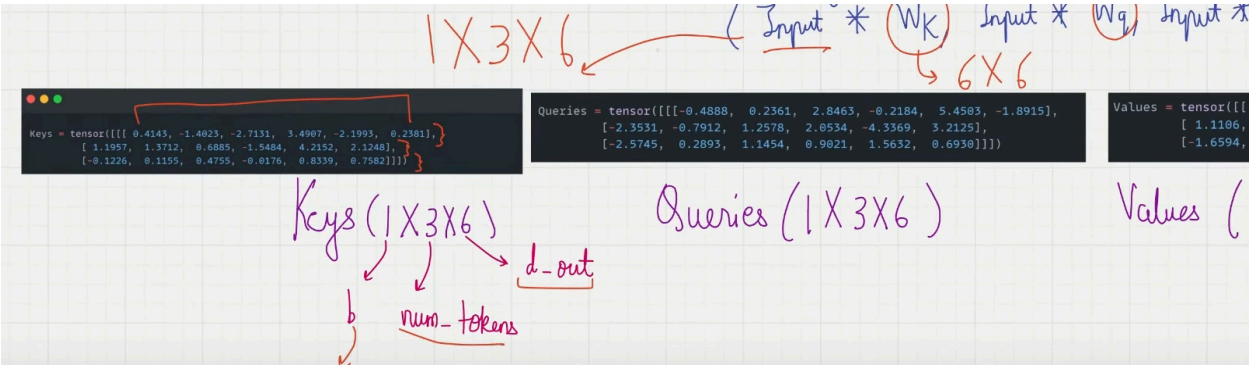
- Where b is the **batch size,** num_tokens is **no of tokens present in the input embedding tensor**, d_in is **input vector dimension which is basically the dimension of each tokens** present in the input embedding vector.
- 

- Step 2: Decide d_out, num_heads
  - Ultimate goal is to get a context vector from the input embeddings and we need to decide upon the output dimensions for the context vector which is **d_out**
  - In this example since we have 3 tokens the context vector will have a dimension like this 3 x d_out.
  - We have to decide d_out. For this example we take d_out=6 typically in GPT models (d_in = d_out).
  - **num_heads** is basically no of attention heads used. In GPT3 we use 96 attention heads in this example we take **num_heads=2**
  - Each head will have a dimension which is called **head_dim = (d_out/num_heads)** here it is 6/2 = 3.

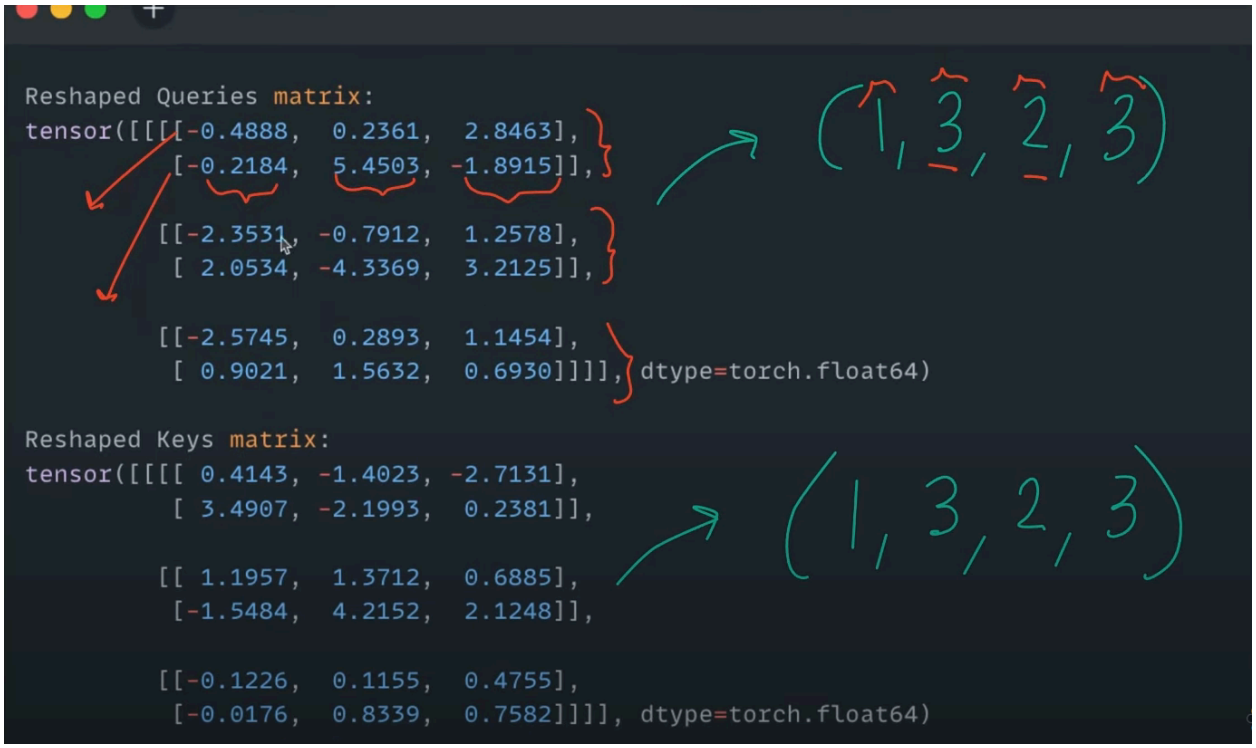
- Step 3: Initialize trainable weight matrices for key,query,value (W_k,W_q,W_v)
  - Dimensions of these trainable matrices are (d_in x d_out) here it is 6x6



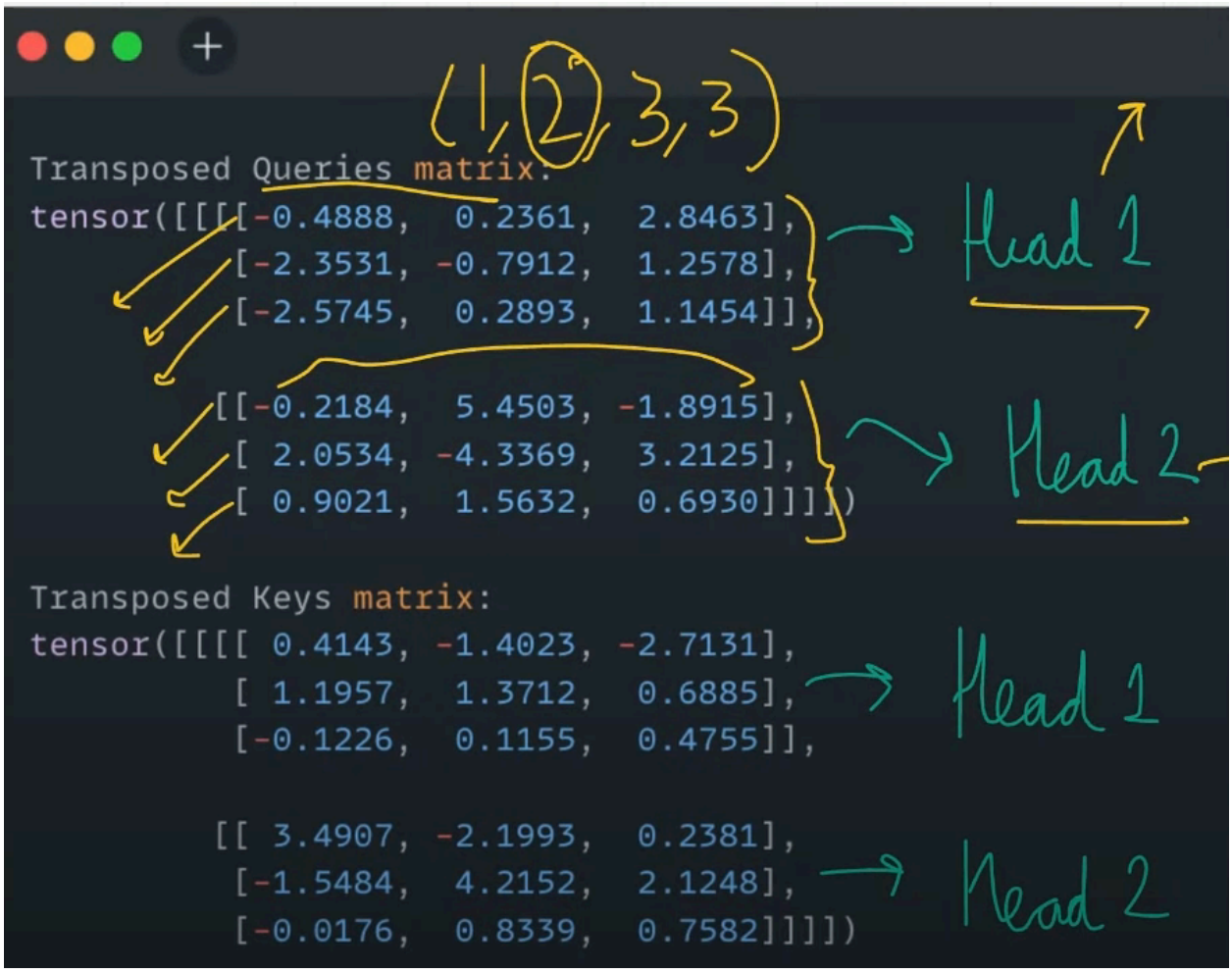
- Step 4: Calculate Keys,Queries,Value Matrices
  - Input * (Trainable Weight Matrices)
  - Input Matrix dimension = 3x6
  - Trainable Matrices dimension = 6x6



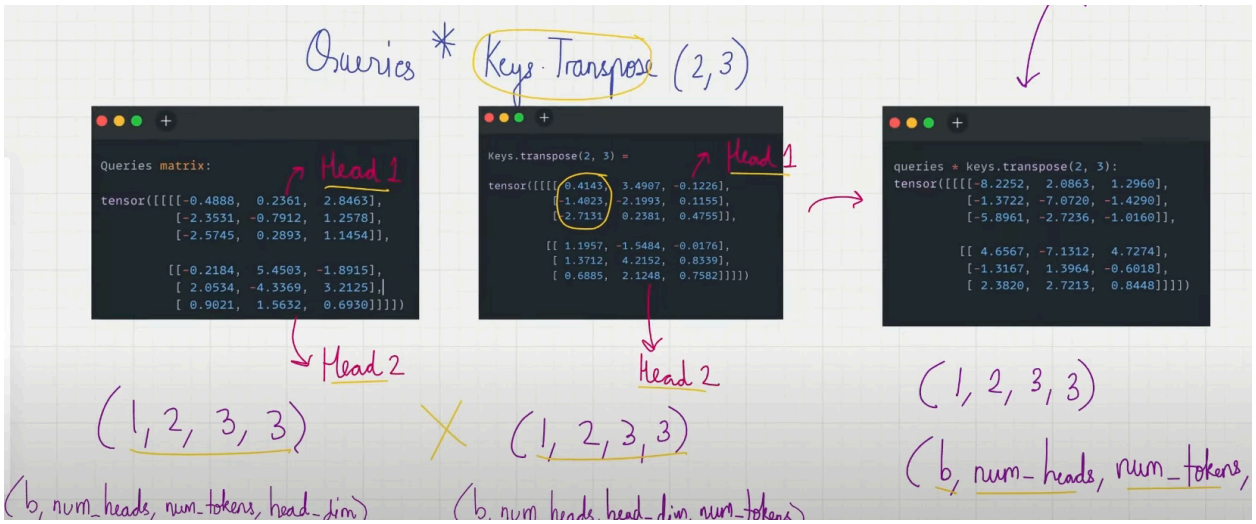
- Step 5: Unroll last dimension of Keys, Queries and Values to include num_heads and head_dim
    - head_dim =3
    - (b,num_tokens, d_out) → (b,num_tokens,num_heads,head_dim) **replacing** d_out with num_heads and head_dim
    - (1,3,6) → (1,3,2,3)



    - Interpreting 4 dimensional tensors use chat-GPT (1,3,2,3) -> 1 represents batch_size since it is 1 now leave it
    - 3 represents no of tokens so total of 3 rows where each token has 2 attention heads and each attention head having dimension of 3
- Step 6: Group matrices by “number of heads” instead of number of tokens
    - (b,num_tokens,num_heads,head_dim) → (b,num_heads,num_tokens,head_dim)
    - (1,3,2,3) → (1,2,3,3)

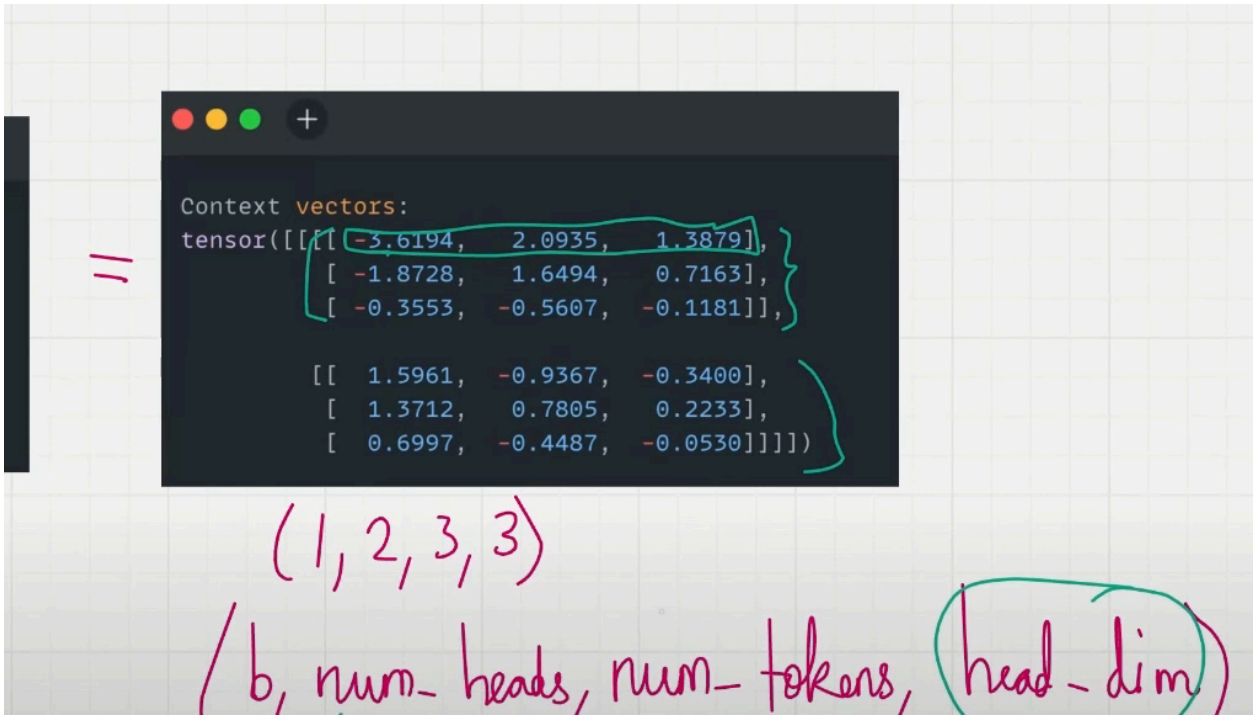


- Step 7: Find Attention Scores
    - Queries x Keys Transpose (2,3)

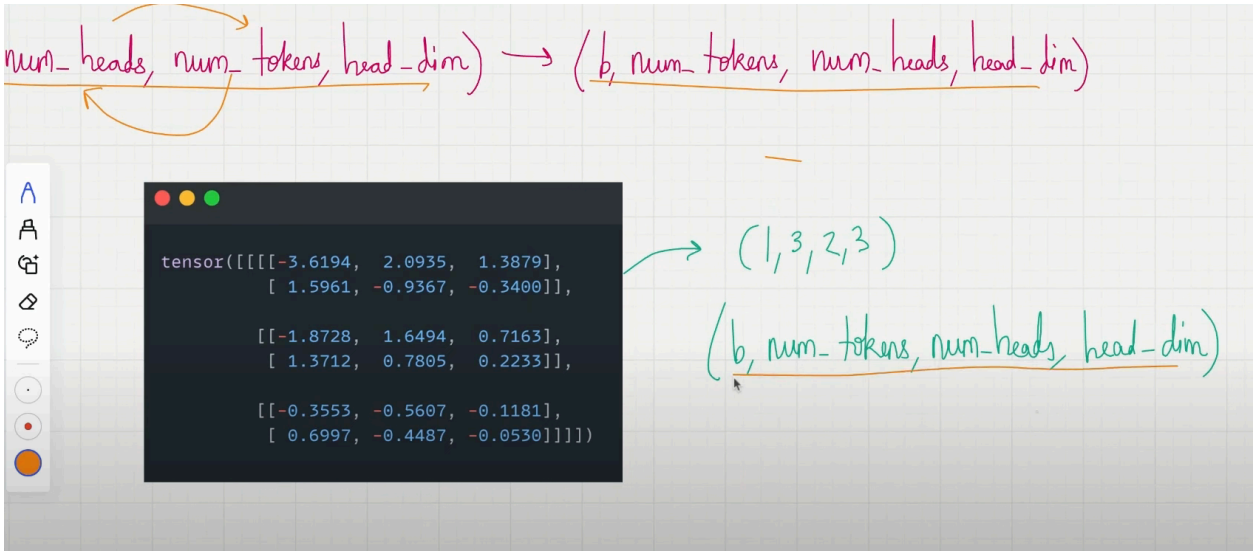


- Step 8: Find Attention Weights

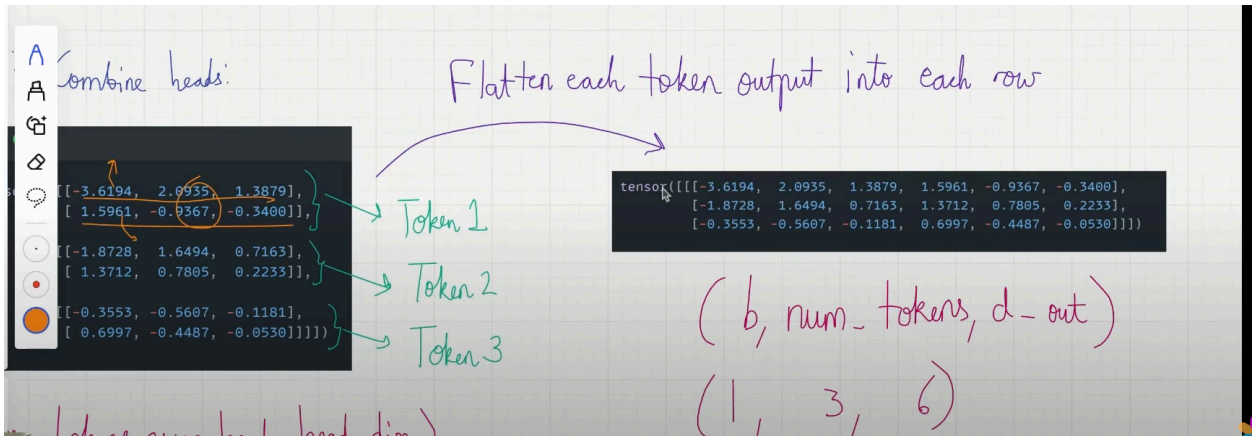
    - Mask attention scores to implement causal attention
    - Scale by root of head_dim and use softmax function
    - Implement dropout as well
- Step 9: Computing the Context vector
    - Context Vector = Attention weights * Values
    - Attention weights (b,num_heads,num_tokens,num_tokens) x Values (b,num_heads,num_tokens,head_dim)



- Step 10: Reformat context vectors
    - (b,num_heads,num_tokens,head_dim) → (b,num_tokens,num_heads,head_dim)
    - Transpose of 1st and 2nd Index

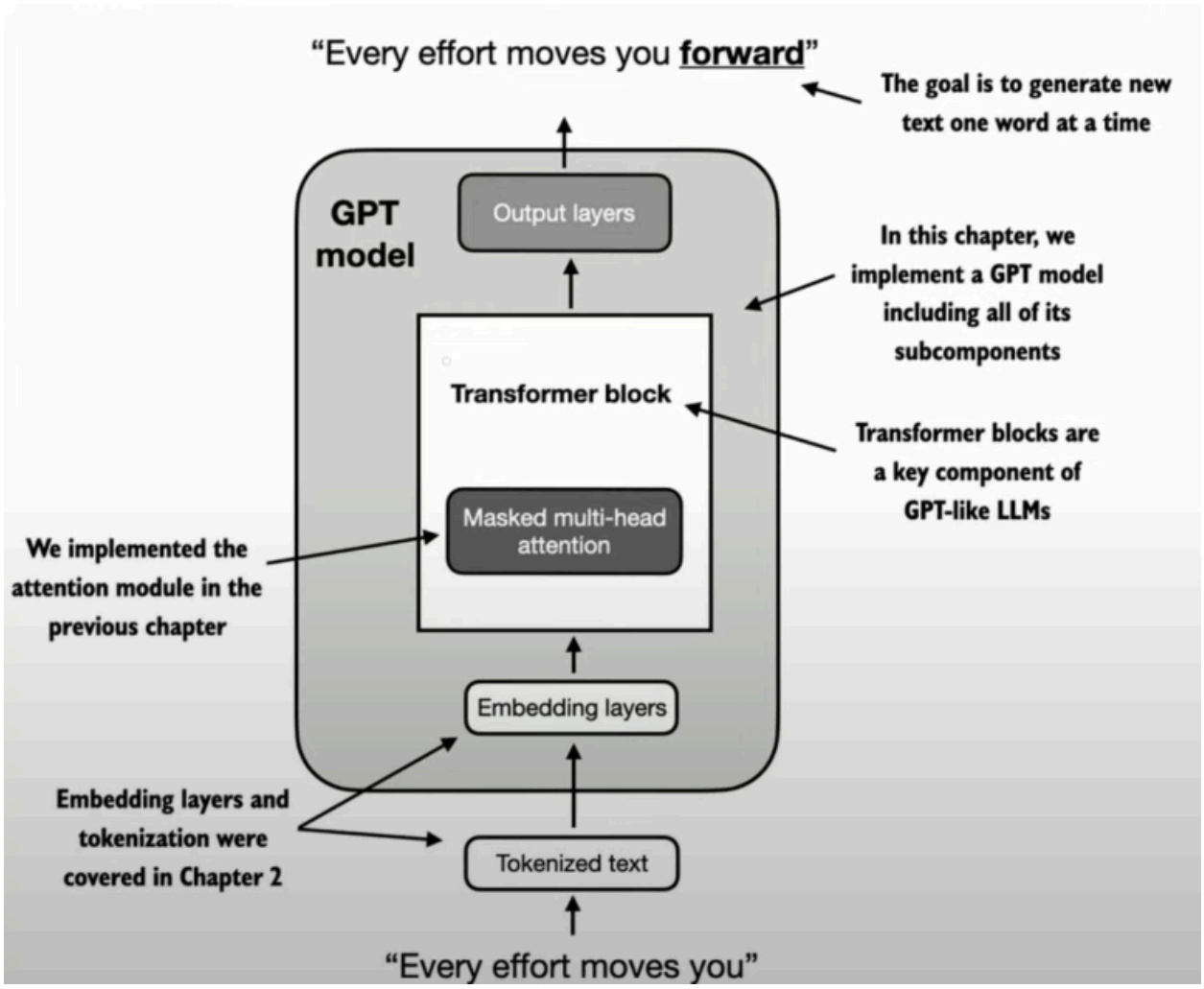


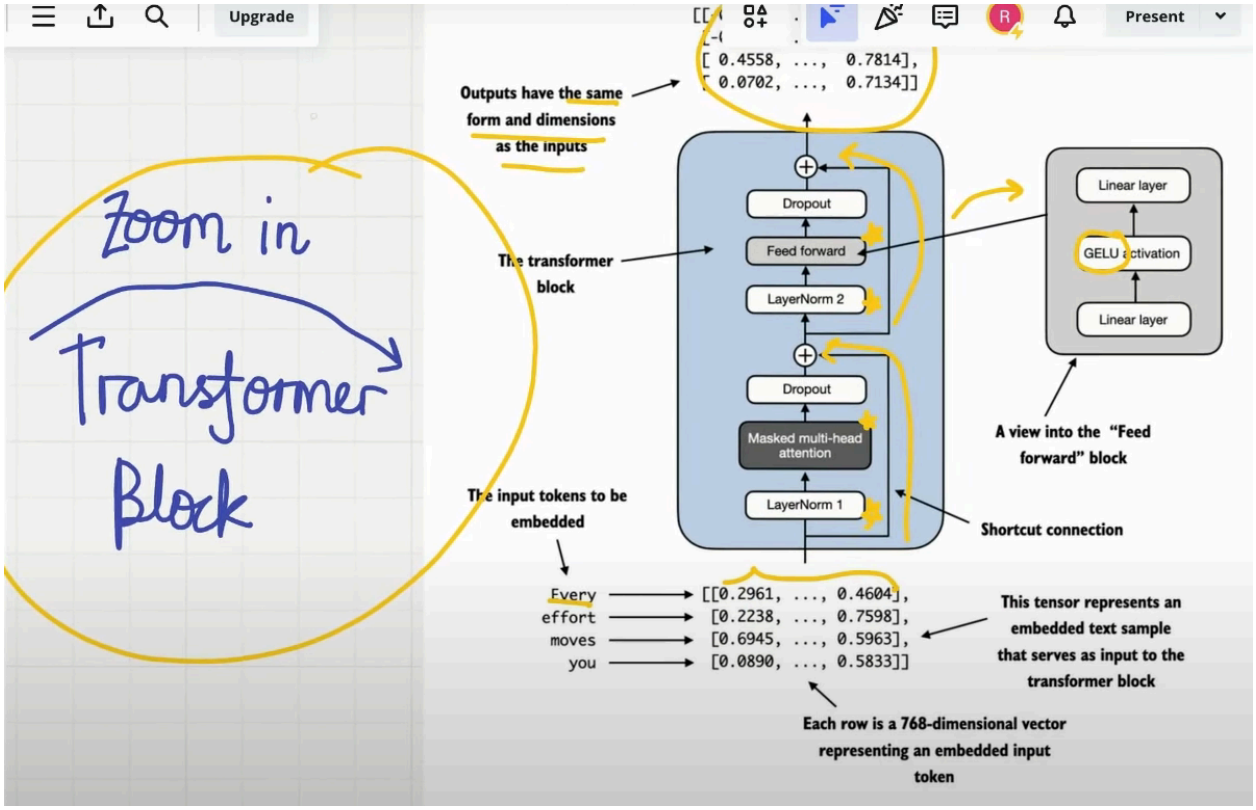
- Step 11: Combine Heads



Lecture 19: Birds eye view of the LLM architecture

After learning about the attention mechanism, Let us learn about the LLM architecture now.


"Every effort moves you forward"
The goal is to generate new text one word at a time
GPT model
Output layers
In this chapter, we implement a GPT model including all of its subcomponents
Transformer block
Transformer blocks are a key component of GPT-like LLMs
Masked multi-head attention
We implemented the attention module in the previous chapter
Embedding layers
Embedding layers and tokenization were covered in Chapter 2
Tokenized text
"Every effort moves you"



What are we yet to learn:

- Transformer block
- We will scale up to the size of a small GPT-2 model → 124 million parameters

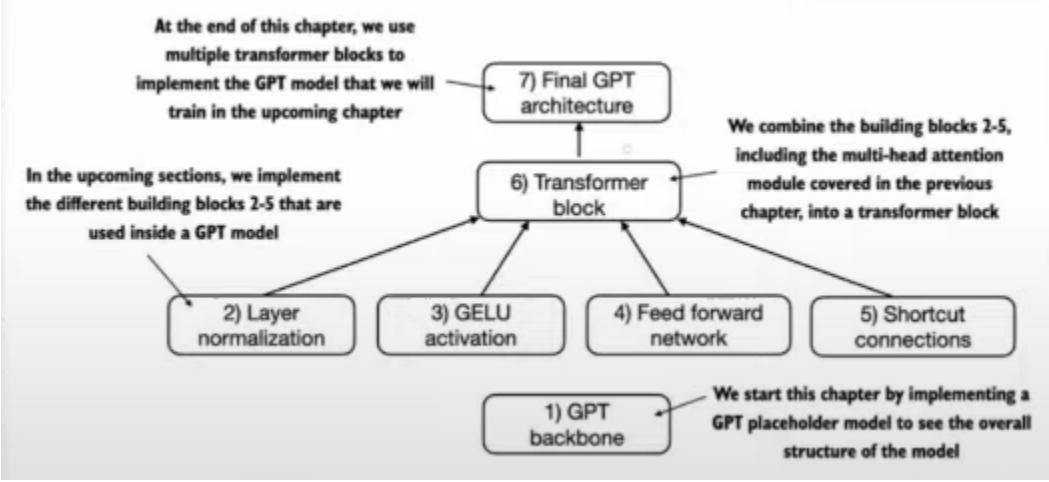
Configurations we are going to use:

- N_layers = no of transformer blocks

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,  # Vocabulary size
    "context_length": 1024,      # Context length
    "emb_dim": 768,       # Embedding dimension
    "n_heads": 12,        # Number of attention heads
    "n_layers": 12,       # Number of layers
    "drop_rate": 0.1,     # Dropout rate
    "qkv_bias": False     # Query-Key-Value bias
}
```

Using this configuration, we will build a GPT placeholder architecture. the skeleton for the code

Entire Transformer Block and things to do for the upcoming Lectures Layout



Lecture 20: Layer Normalization

1. Training deep neural networks with many layers can be challenging due to **Vanishing/Exploding gradients** problems.
2. This Leads to unstable training dynamics.
3. Layer normalization improves the stability and efficiency of neural network training

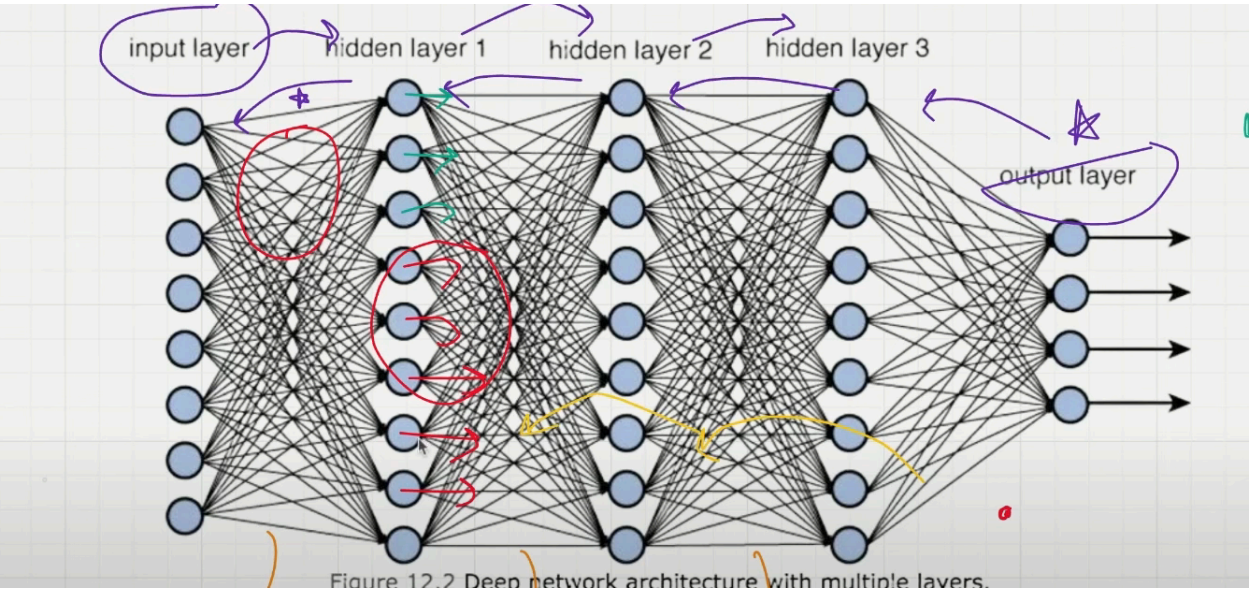


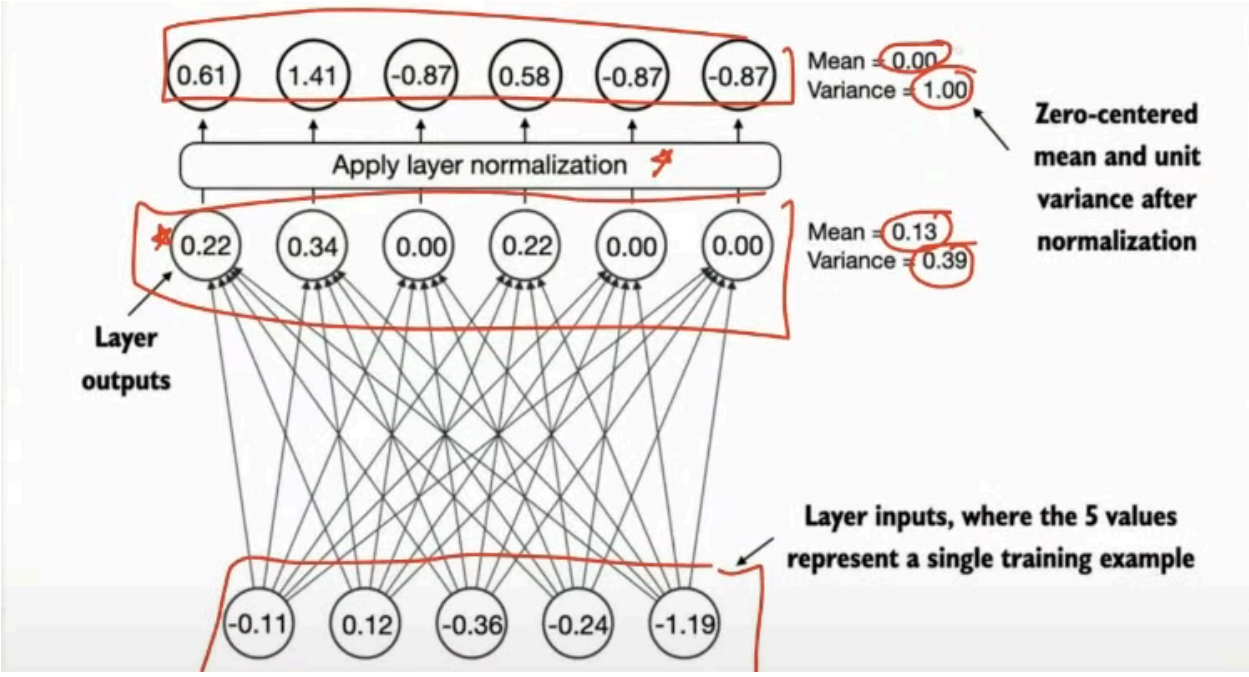
Figure 12.2 Deep network architecture with multiple layers.

Why we use Layer Normalization:

- If layer output is too large or small, gradient magnitudes can become too large or small - this affects the training
- As the training proceeds, inputs to each layer can change (Internal covariate shift) → this delays convergence

Main idea of Layer Normalization:

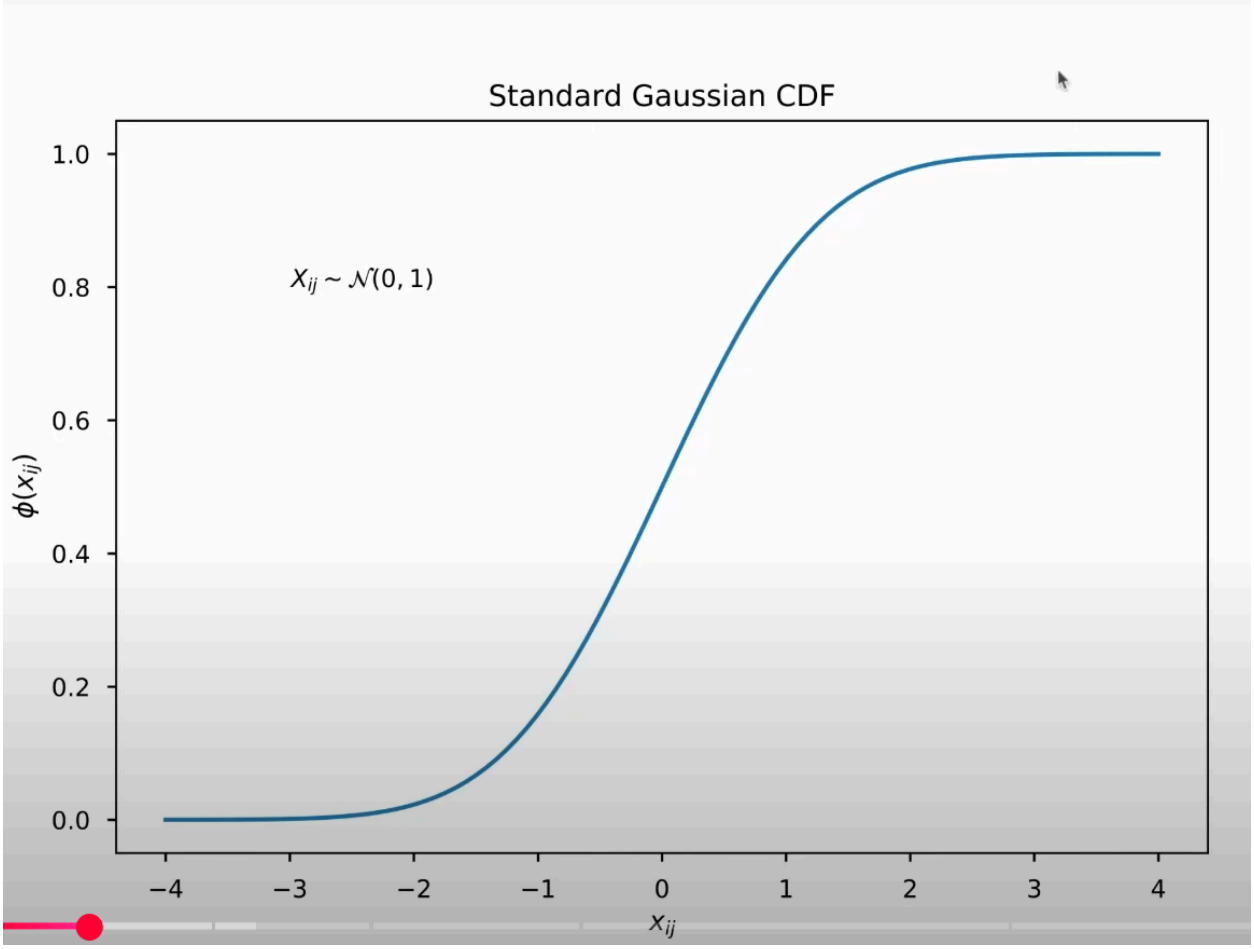
- Adjust **outputs** of neural network to have mean = 0 and variance = 1
- This speeds up the convergence



In GPT-2 and modern transformer architectures, layer normalization is typically applied *before and after the multi-head attention module and before the final output layer.*

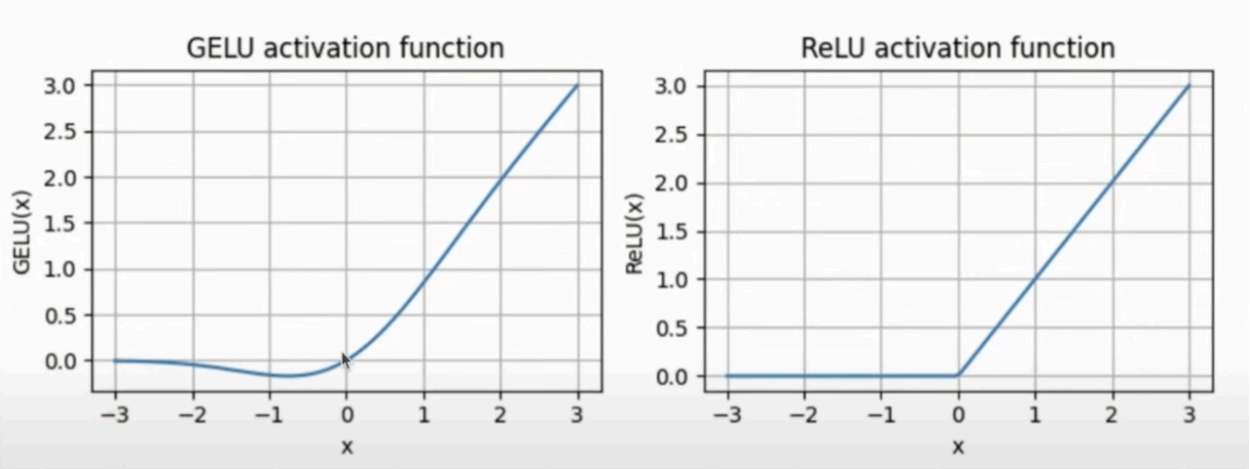
Lecture 21: GELU Activation Function

- We will Implement a small neural network sub-module that is a part of the LLM transformer block
- Two activation functions that are commonly implemented in LLMs are:
    - GELU
    - Swi GLU
    - Note: Have an understanding about the dead neuron problem. That's why we don't use the RELU activation function.
- GELU f'n = x * phi(x) where phi(x) → CDF of standard Gaussian distribution
-

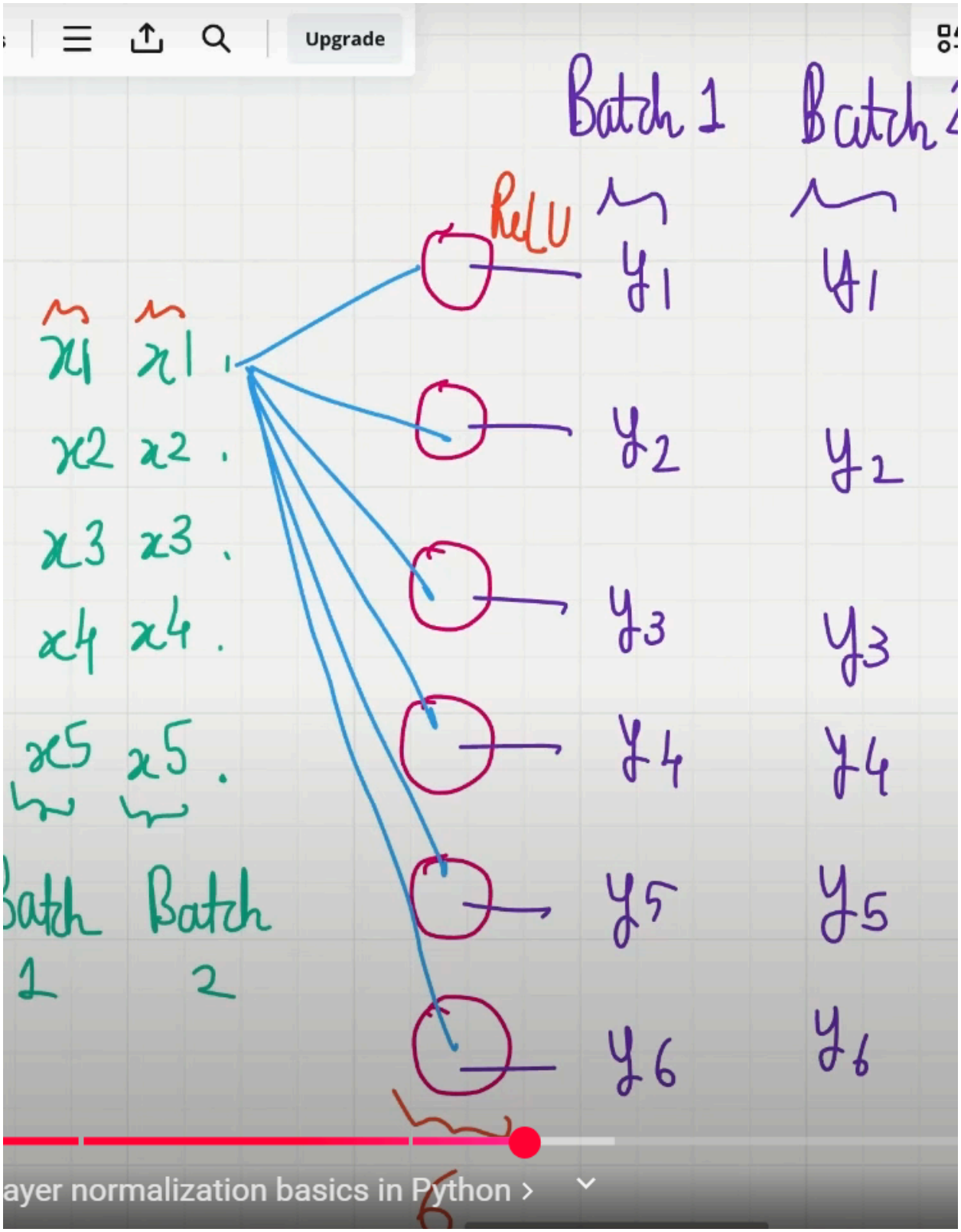


- But in GPT models we actually use GELU activation function approximation used for training GPT-2
- $$GELU(x) \approx 0.5 \cdot x \cdot (1 + \tanh[\sqrt{(2/\pi)} \cdot (x + 0.044715 \cdot x^3])$$

- Difference between RELU and GELU
-



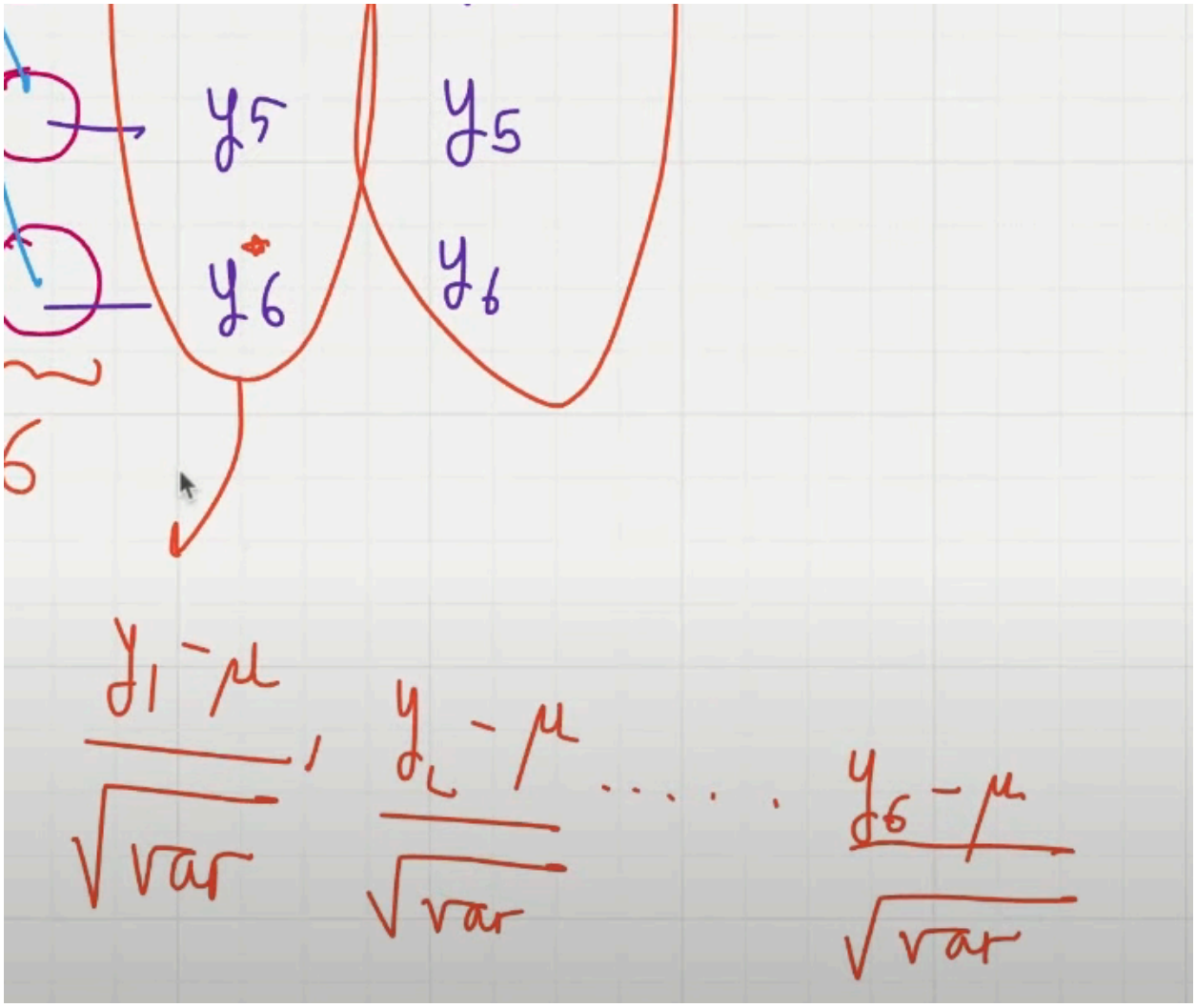
-


Upgrade
Batch 1
Batch 2
ReLU
x1 x1
x2 x2
x3 x3
x4 x4
x5 x5
Batch 1
Batch 2
y1 y1
y2 y2
y3 y3
y4 y4
y5 y5
y6 y6
ayer normalization basics in Python ›

X → Inputs input.shape = (2,5) → 2 batches of inputs each having 5 inputs
Y → Outputs output.shape = 6
Middle layer has 6 neurons - RELU activation function

Normalization:

- After getting all the Y values for each batches we apply the layer normalization



Every Y value getting replaced by normalization

Lecture 22: Shortcut Connections

- Also Known as Skip or Residual connections
- Shortcut connections were first proposed in the field of computer vision to solve the problem of **vanishing gradients.**

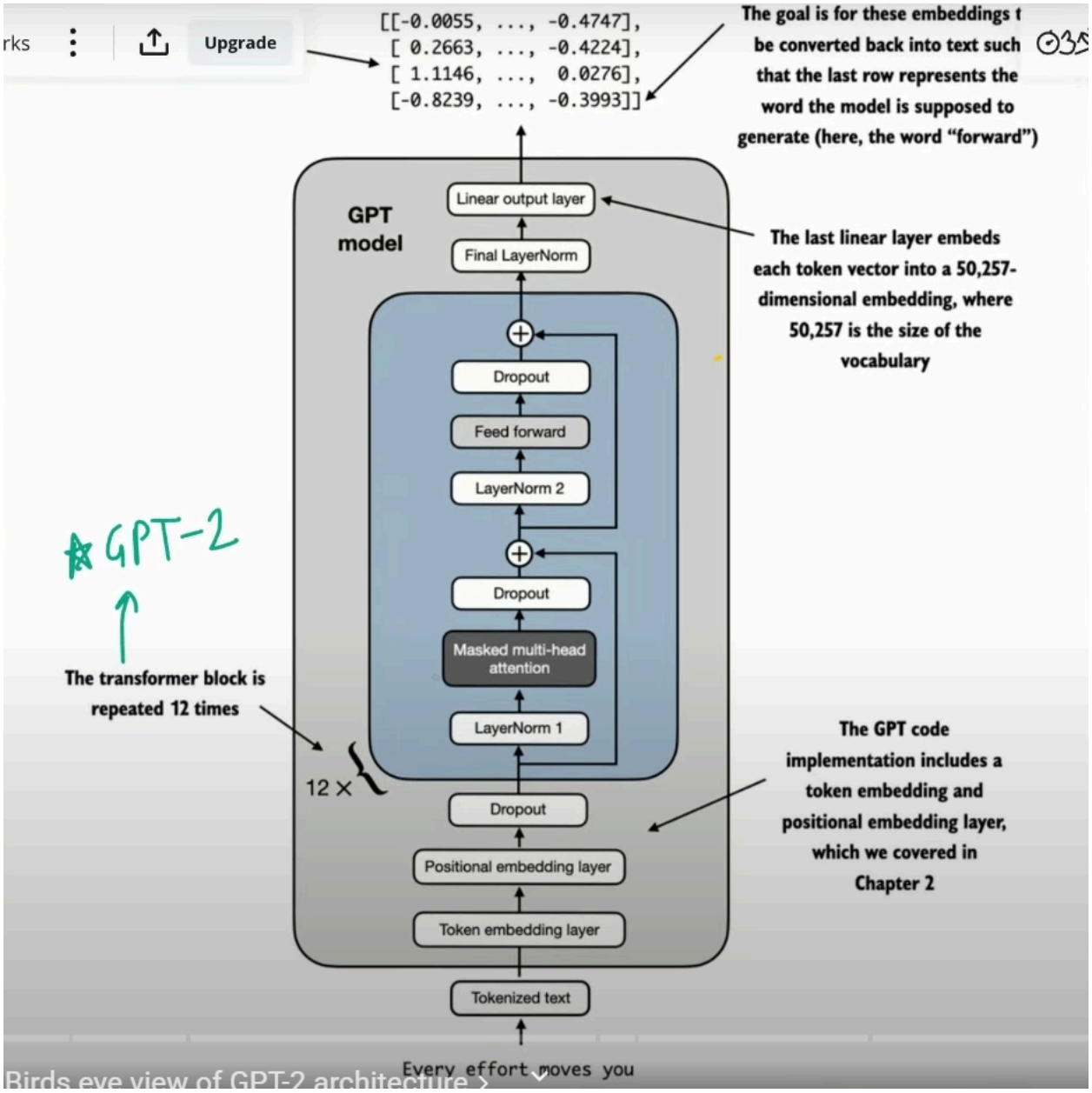
- Vanishing Gradients: Gradients become progressively smaller as they propagate backward → making it difficult to train earlier layers.
- Shortcut connections create an alternative path for the gradient to flow, by skipping one or more layers
    - This is achieved by adding the output of one layer to the output of a latter layer
    - Play a crucial role in preserving flow of gradients during training backward pass

Lecture 23: Coding the entire Transformer Block

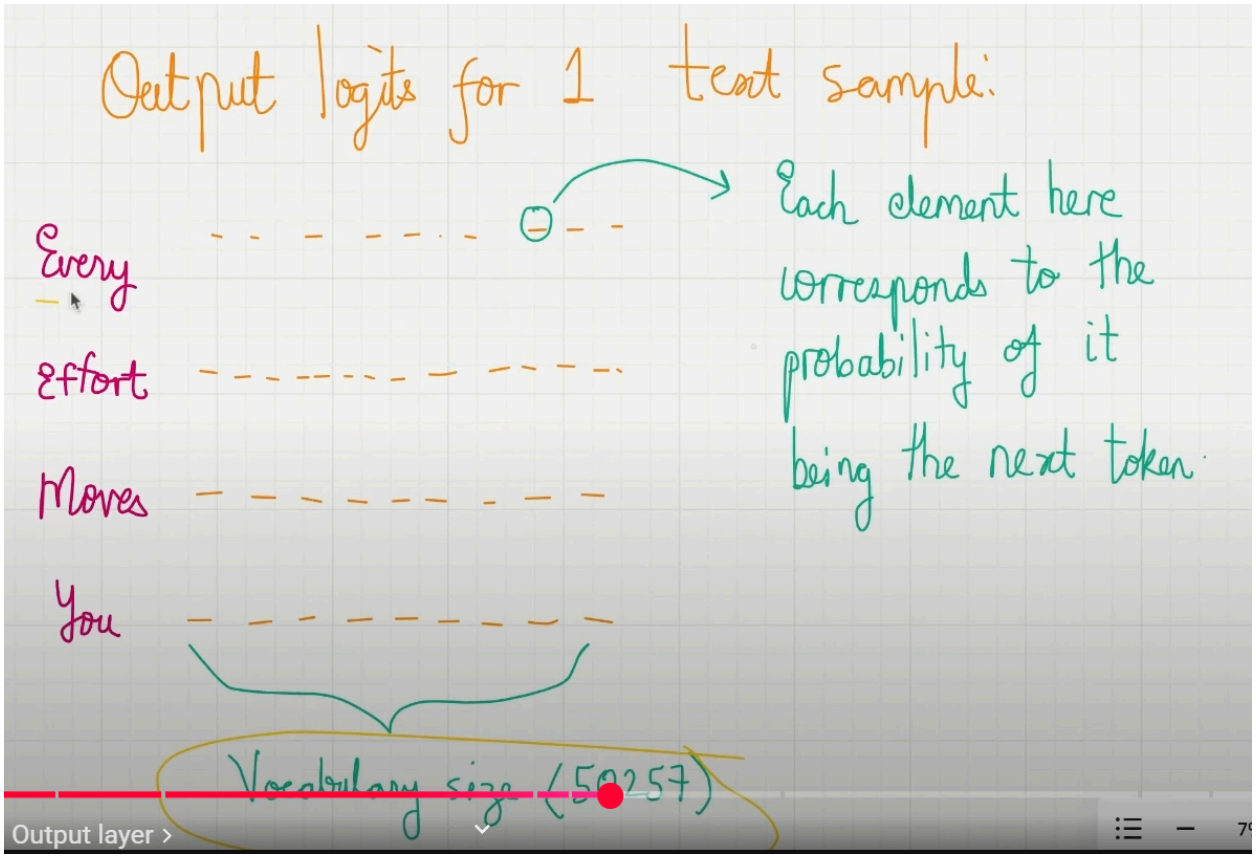
- The transformer block is the fundamental building block of GPT and other LLM architectures
- The transformer block is repeated 12 times in GPT-2 small (124M parameters)
- Transformer Block consists of:
    - Multi-head Attention
    - Layer Normalization
    - Dropout
    - Feed-Forward Layers
    - GELU-Activation
- When a transformer block processes an input sequence, each element is represented by a fixed size vector
- The operations within the transformer block such as multi-head attention and feedforward layers are designed to transform the input vectors such that their **dimensionality is preserved - IMPORTANT ADVANTAGE OF TRANSFORMER**
- Self attention → analyzes relationship between input elements
- Feed-Forward Network → Modifies data individually at each position

Lecture 24: Coding the 124 Million Parameter (GPT-2 Model)

- Assembling a fully working version of the original 124 M Parameters version of GPT-2
- Understanding the below Diagram (Recap)



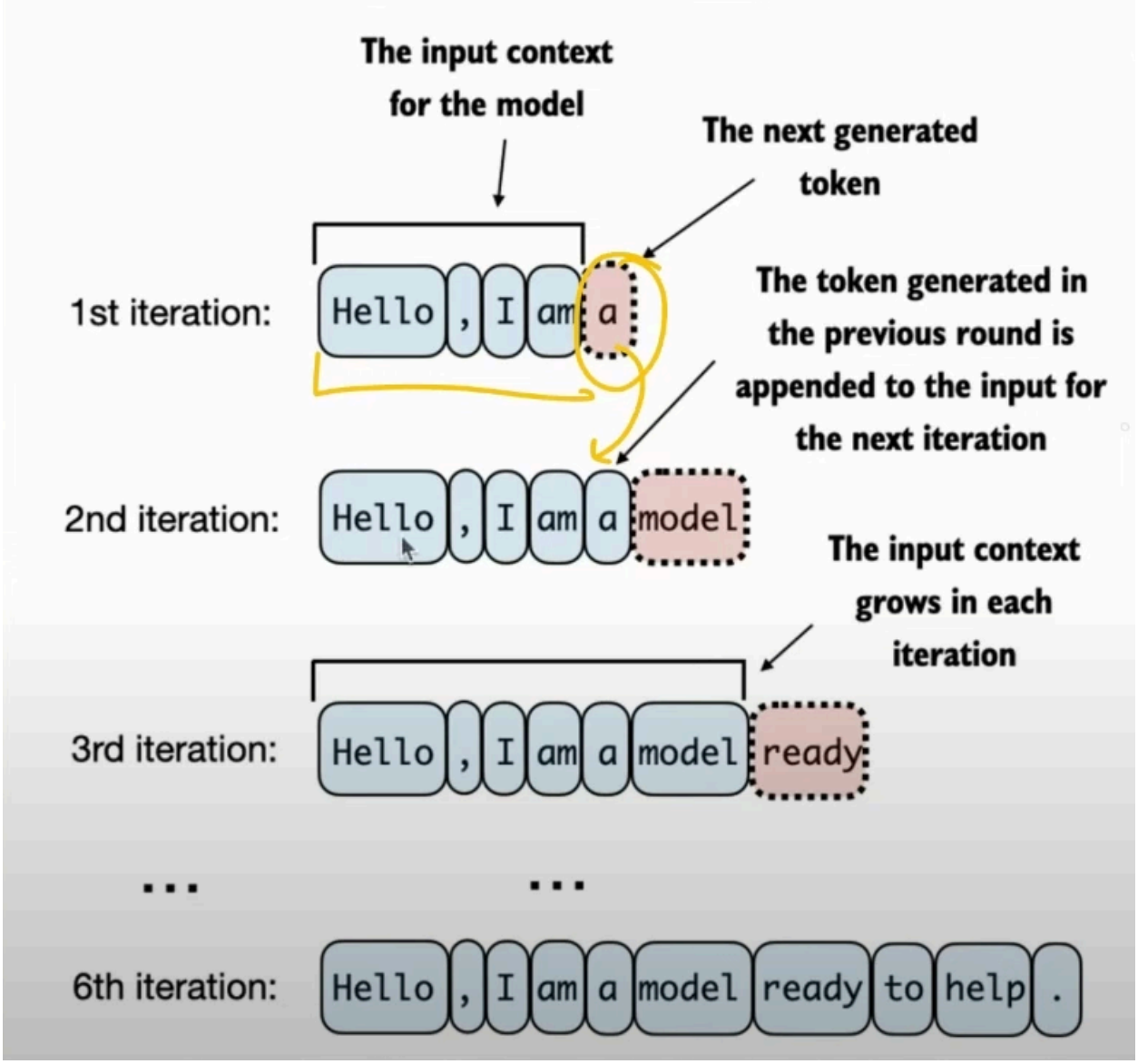
- As we have already explored in detail about the Input Block and Transformer Block we now look into the output stage
- At the very first part of the output stage:
    - The input to the output head will be passing the tokens in this example we take context size of 4. The Example is : “Every effort moves you”. each token with an embedding size of 768.
    - We pass this to a neural network and the dimension becomes embedding dimensions multiplied with the vocabulary size so in this case it is (768x50257)
    - The output will be a logits matrix which will look like this:



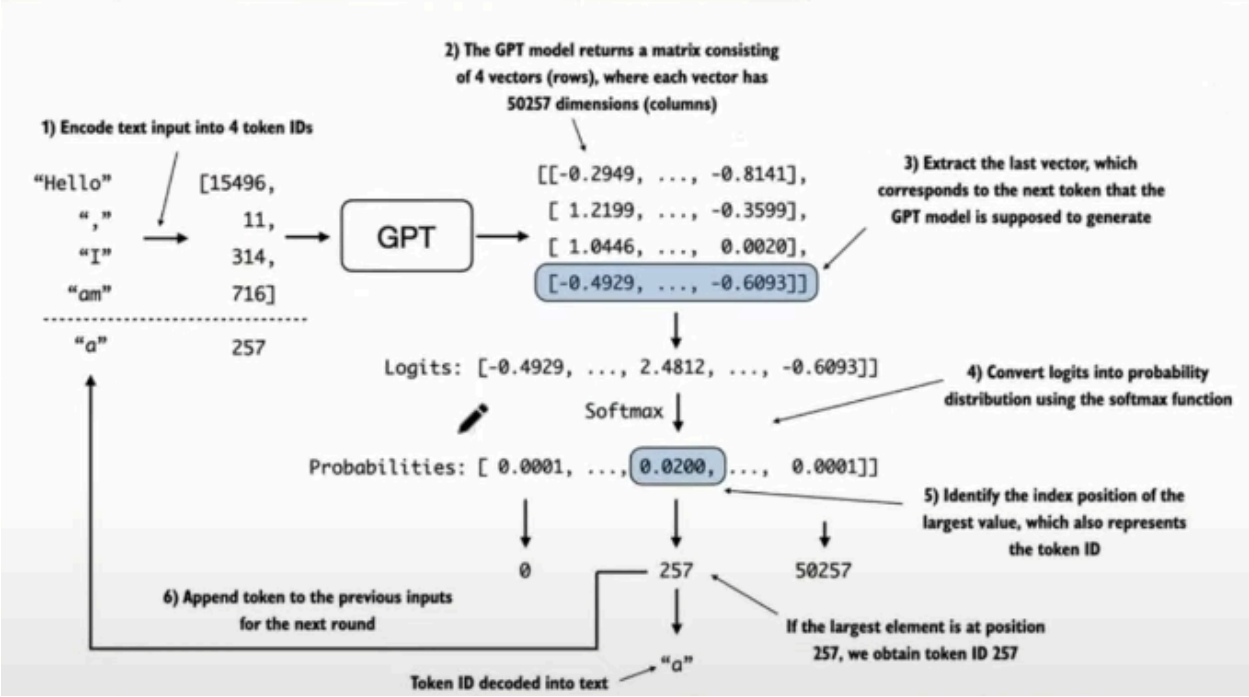
- Each element corresponds to the probability of it being the next token
- Context size = no of prediction tasks. In this example we do not just have 1 prediction task rather we have 4. Final tensor dimension:(1x 4 x 50257). 1 is the batch size

Lecture 25: Coding GPT- 2 To Predict the Next Token

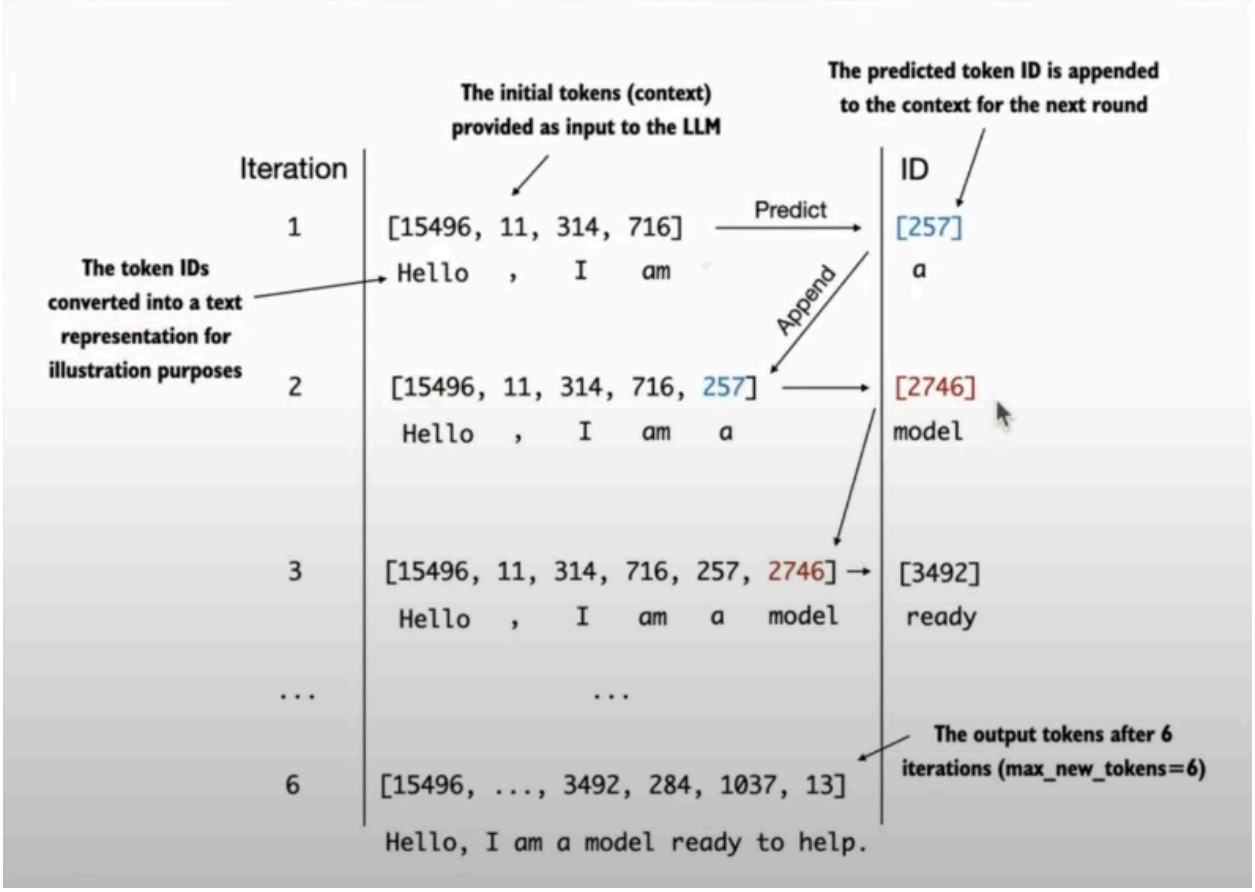
- The main tasks LLMs are trained to do is to predict the next token (word) given a context size below is a visual diagram



- Now we are looking into how the LLMs are made to predict the next word after the output layer has been formed which has a dimension of (1x4x50257)
- Steps at a Glance:
    - Step 1: Take the output from GPT model
    - Step 2: Extract the Last Vector here in this case it is the token **“You”**
    - Step 3:Convert the Logits Vector into Probabilities by applying Softmax
    - Step 4: Identify the Index position (Token ID) of the largest Value and Decode the Token ID which will give you the text
    - Step 5: Append Token ID to the Previous Inputs for the next round of Prediction Tasks



- This step-by-step process enables the model to generate text sequentially, building coherent phrases from the initial input context
- In practice, we repeat this process over many iterations, until we reach a **user specified number of generated tokens**



Completion of Stage 1: Building an LLM Architecture

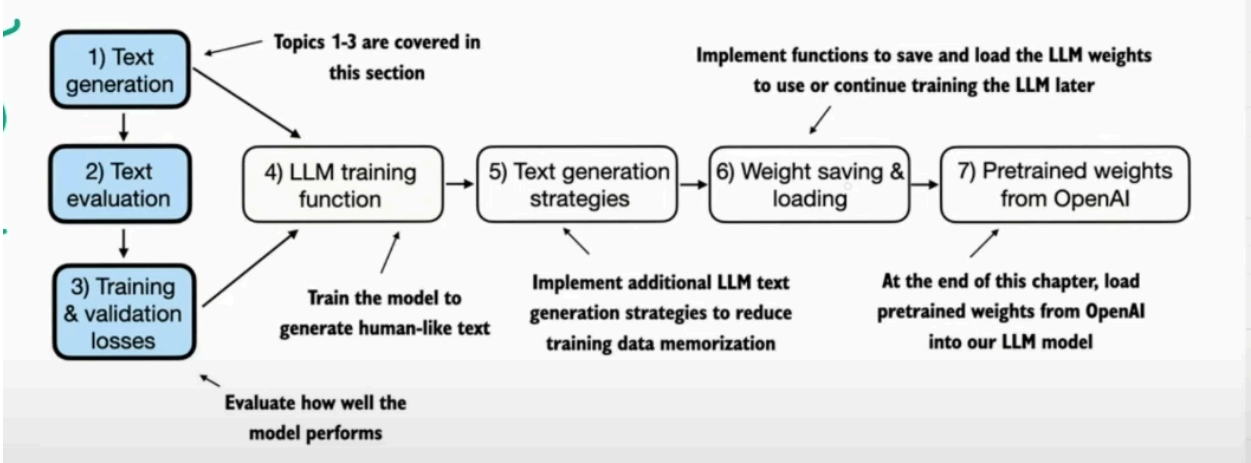
which included:

1. Data Preparation and Sampling
2. Attention Mechanism
3. LLM Architecture

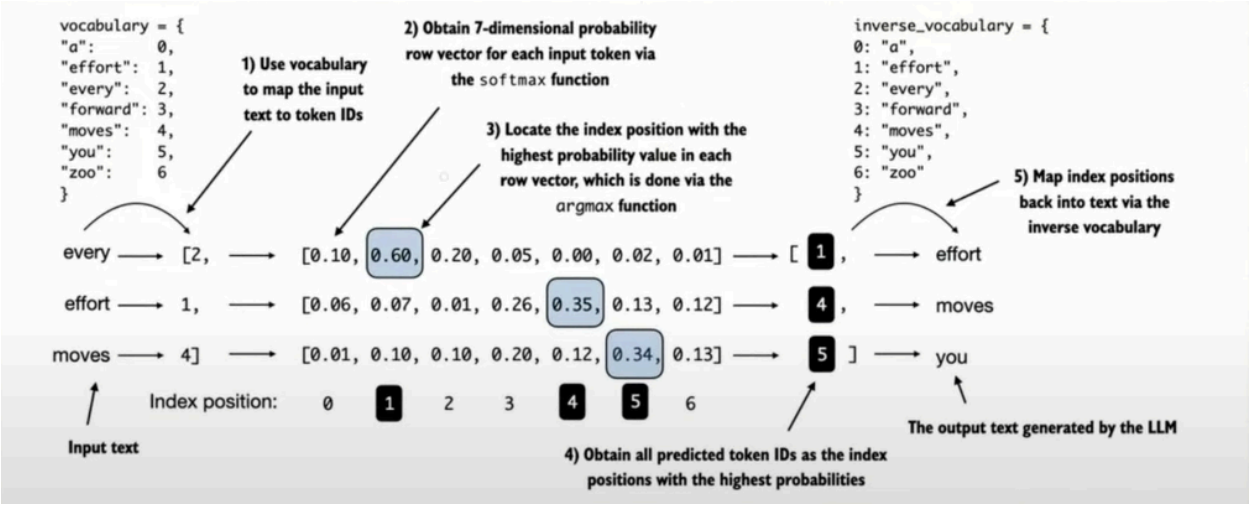
What have we achieved? and next steps:
Now our GPT Architecture can take a sentence and Predict the next word. Now we want to make our predictions better which happens through training the model. Uptill now we have a model where those 124 M parameters are randomized. We need to train the model to get the best possible results.

Lecture 26: Pretraining LLM's Loss Function

- How to measure LLM Prediction Loss?



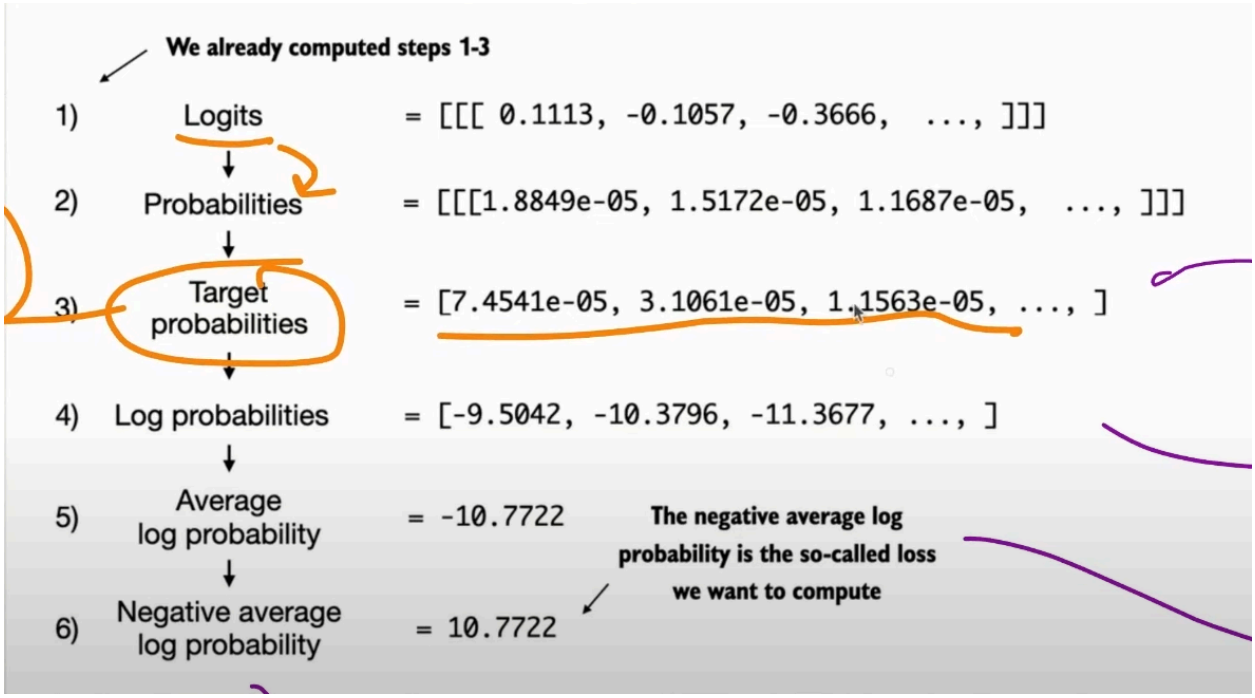
- Text generation is covered in detail in the previous Lectures on How LLMs predict the next word given an input.
- Recap of GPT workings:
    - Step 1: Tokenizer converts tokens into token IDs
    - Step 2: GPT model converts token IDs into Logits
    - Step 3: Logits are converted back into token IDs



- Now we want to make sure every **token ID predicted is as close to the target value.**
- In the above diagram 3 prediction tasks occur (context size = 3) “Every effort moves”
- Finding the loss between Targets and Predicted Outputs
    - Target Indices:
        - Since we know the token IDs of the target tensor from those token IDs we can get their Probabilities from the Logits Matrix although currently those values may not be the maximum.

```
targets = torch.tensor([[3626, 6100, 345  ],  # [" effort moves you",
                        [107,  588, 11311]]) #  " really like chocolate"]
```

- Basically from the above target tensor use those token IDs which basically serves as the indices in the logits matrix (here dimension 1x 4 x 50257) to extract those token probabilities which are the required prediction values
- Target Probabilities:
    - Batch 1: [p11,p12,p13]
    - Batch 2: [p21,p22,p23]
    - These probabilities were extracted from the target indices
    - Goal of our Training is to get all these probabilities values as close to 1 as possible as these values are not necessarily the highest value in the logits matrix.
    - Why do we want these values as close to 1?. Because these are the values we want the LLM to predict as outputs
    - Merge them together [p11,p12,p13,p21,p22,p23]



- The final value 10.7722 is the cross entropy loss and this is the loss we want to minimize as much as possible
- Perplexity: Another Loss Measure like cross entropy
    - Measures how well the probability distribution predicted by the model matches the actual distribution of words in the dataset.
    - More interpretable way of understanding model uncertainty in predicting next token.
    - Lower perplexity score = Better predictions
    - Perplexity = e^{Loss} here it is = 48725

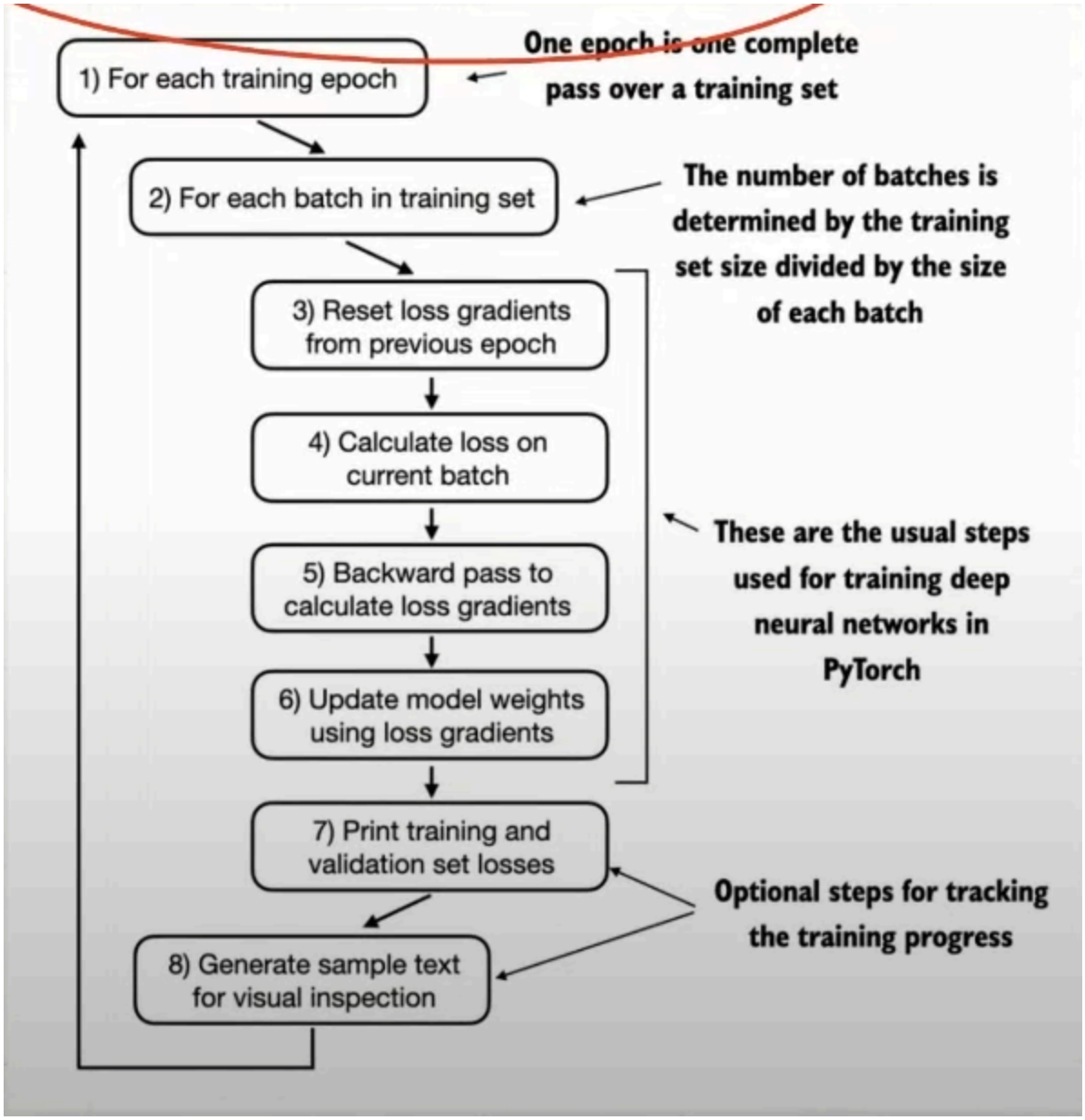
    - This number 48725 tells us that the model is roughly as uncertain as if it had to choose the next token randomly from about 48725 tokens in the vocabulary

Lecture 27: Evaluating LLM Performance on real dataset

- This Lecture is mostly a repeat of the previous lecture. Here we are using a real dataset (Verdict by Edith Wharton)
-

Lecture 28: Coding The Entire LLM Pretraining Loop

Schematic:



Lecture 29: Temperature Scaling in LLMs

- Until now, the generated token is selected corresponding to the largest probability score among all tokens in the vocabulary
- Leads to a lot of randomness and diversity in generated text
- We will learn 2 techniques to control this randomness
    - Temperature Scaling
    - Top - K Sampling
- Temperature Scaling:
    - Replace Argmax with probability distribution

    - Multinomial Probability distribution samples next token according to probability score
    - What is Temperature: Fancy term for dividing the logits by a number greater than zero
    - Scaled logits = logits / temperature . this scaled logits is passed through softmax –> multinomial sampling
    - If the temperature is too low, it leads to a sharper distribution. Almost no variability or creativity.
    - If Temperature value is too high leads to flattened distribution (more variety, but also more nonsense)
    - Why the name temperature? → low temperature leads to low randomness which is associated with low entropy. Higher temperature leads to higher randomness which is associated with high entropy

Lecture 30 : Top - K Sampling in LLMs

- Restrict the sampled tokens to the Top - K most likely tokens and exclude all other tokens