LLM

- A deep neural network trained on massive amounts of **text** data
- Goal is to **understand, generate, and respond** to human-like language
- Trained through next-word **prediction** task
- Large? Size of what?
    - Model size: parameter number
    - Dataset size: number of data/training text
- Applications
    - Can be general or domain-specific knowledge use
    - Chatbots and Virtual assistants
        - Process large volumes of specialised text
        - To be useful in professional domains
    - Automates text-heavy cognitive work
- Why build custom LLMs
    - Better performance on specific domains
    - Improved data privacy due to local deployment
    - Lower latency and reduced costs
- Training phases
    - Pre-training
        - Uses massive unlabeled text datasets
        - Objective: using next-word prediction task, gain a foundational model with general language understanding
        - Understand the language
        - Learns general linguistic knowledge: grammar, word choice, etc
        - Self-supervised learning
            - Labels are generated automatically from the data
            - No need for manual annotation
    - Fine-tuning
        - Adapts the foundation model to specific tasks
        - Uses smaller and labelled datasets
        - Types
            - Instruction fine-tuning
                - Data format: instruction - response pairs
            - Classification fine-tuning
                - Data format: text + class labels
- Properties of training data
    - designed to expose the models to diverse text, enabling them to learn language syntax, semantics, and context
    - Diverse topics and languages
    - Reflects real-world language complexity
    - Tokens
        - Converts meaningful units fo text, like words, subwords, punctuation, into tokens
- Core training objective
    - primarily **trained on a next-word prediction task**, which involves **predicting the next word** in a sequence. This seemingly simple task allows the models to **learn the relationships between words** and phrases,

enabling them to perform other tasks like translation, even though they were not explicitly trained for it.
    - Decoder-only architecture
        - GPT uses **only the transformer decoder**
        - Generates text **left-to-right**
        - Ideal for autoregressive generation
        - Suitable for text generation and next-word prediction tasks
    - Autoregressive nature
        - Each generated token becomes part of the next input
        - Ensures coherence and fluency
- Model scaling
    - Models are larger than the transformer
    - With more layers and parameters, stronger generalization
- Model types
    - Pretrained models
        - *The models are trained on massive datasets of text and code, allowing them to perform well on various tasks, including language syntax, semantics, and context.*
    - Base models
        - *Pretrained models that serve as a foundation for further fine-tuning on specific tasks.*
    - Fine-tuning
        - *A process of adapting a pretrained model to a specific task by training it on a smaller dataset related to that task.*
- Building a LLM
    - 3 main stages
        - Implement architecture & prepare data
        - Pretrain an LLM to create a foundation model
        - Fine-tune foundation model for specific tasks
    - Attention during generation
        - Model can attend to all previous tokens when generating the next work
        - The attention mechanism allows the LLM to selective access the entire input sequence when generating output
    - Emergent properties
        - Even though trained on next-word prediction, GPT models can:
            - Translate
            - Classify
            - Summarize
    - Fine tuning benefits
        - Improves performance on specialized tasks
        - Can outperform general-purpose LLMs in narrow domains
        - **Next-Word Prediction**
          → *The task of predicting the next word in a sequence, which is used to train GPT models.*

        - **Self-Supervised Learning**
          → *A type of machine learning where the model learns from the data*

*itself, without requiring explicit labels.*

- **Autoregressive Model**
  → *A type of model that generates text by predicting the next word in a sequence based on the words that have already been generated.*

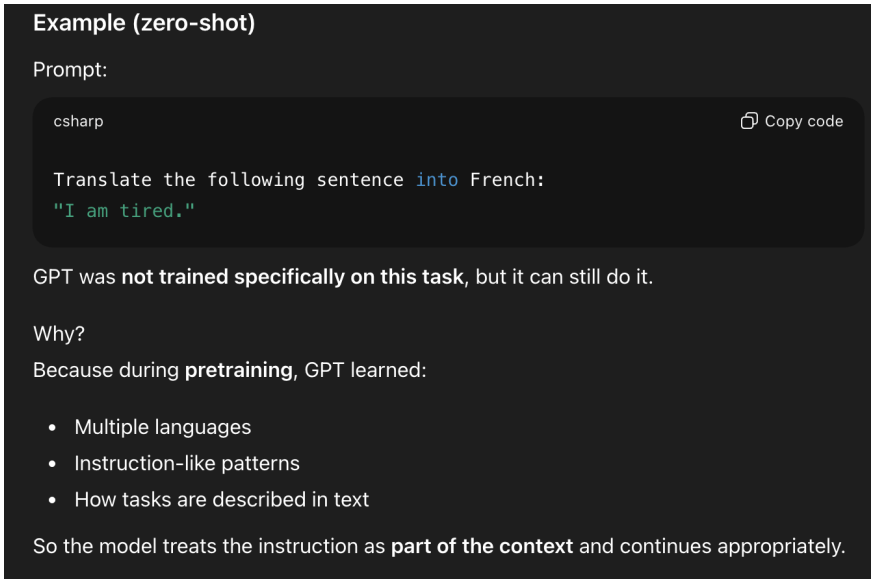
- **Decoder-Only Architecture**
  → *The architecture of GPT models, which uses only the decoder portion of the transformer architecture, making it suitable for text generation.*

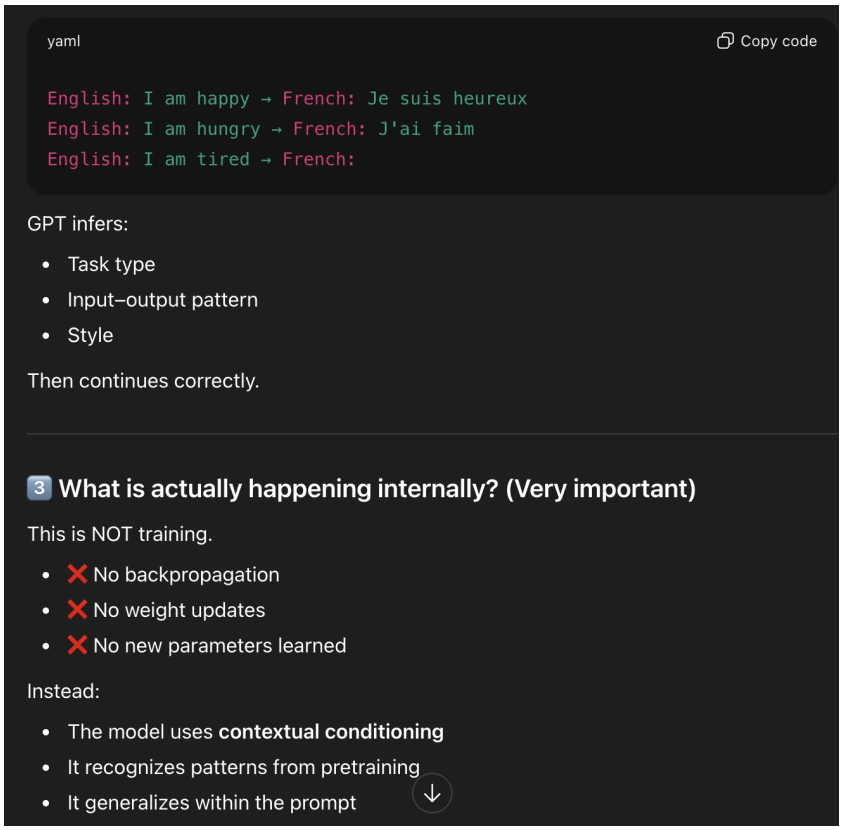
Transformer

- What is it
  - Deep neural network architectures that revolutionised natural language processing
  - Foundation of most modern LLMs
- Transformer uses self-attention
  - Allows the model to focus on different parts input sequence simultaneously
  - Capture subtle language nuances and long-distance dependencies
- Encoder-decoder structure
  - Encoder: processes input text into representations
  - Decoder: generates output text
  - Encoder processes the input text and converts it to numerical representations, while the decoder uses these representations to generate output text
- Self attention mechanism
  - For every word in a sentence, the model decides which other words matter most for understanding it. Not every word contributes equally to meaning
  - Self attention makes context dependent, and focuses on selectivity (giving weight to words on how important they are in the sentence)
  - Texts are more aware of the entire sentence and grammatical features are correct without contradiction or drifts or loss of structure.
  - Enables:
    - Long-range dependency modeling
    - Context-sensitive meaning

Zero-shot learning vs Few-shot learning

- Zero shot:
  - Performs tasks without examples

- 

- Few shot: learn from a small number of examples
- 


AI

- LLM are a part of generative AI
- Generative AI refers to systems that can create new content
- LLM are text-based content
- Hierarchy of AI
    - AI: machines performing tasks requiring human intelligence
    - Machine learning: systems that can learn patterns from data

- Deep learning: ML using deep neural networks: multiply layers to model complexed patterns in data
- LLMs: deep learning models that are specialised in language

**Deep Learning**
*→ A subset of machine learning that uses deep neural networks to model complex patterns and abstractions in data.*

**Generative AI**
*→ A type of artificial intelligence that can create new content, such as text, images, or audio.*

**Transformer**
*→ An architecture used in LLMs that allows them to pay selective attention to different parts of the input when making predictions, making them adept at handling the nuances of human language.*

**Large Language Model (LLM)**
*→ A type of artificial intelligence that uses deep learning to understand, generate, and respond to human-like text.*

## Traditional Machine Learning vs Deep Learning

| Traditional ML | Deep Learning / LLMs |
|---|---|
| Manual feature extraction | Automatic feature learning |
| Human-designed features | Learned representations |
| Limited scalability | Scales with data & compute |

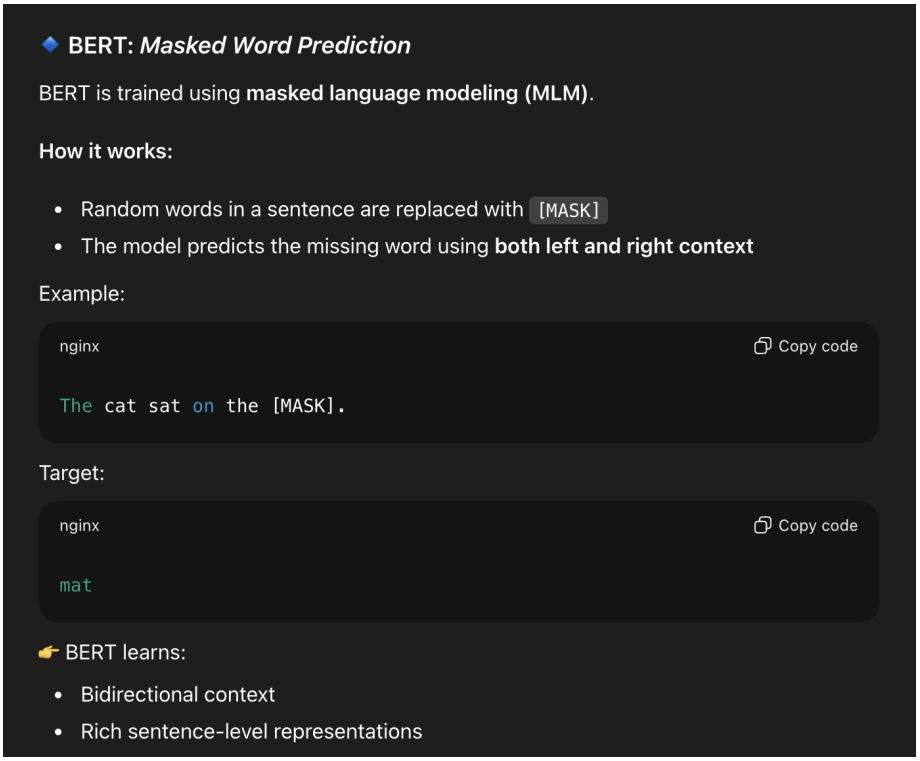
**Key exam point:**

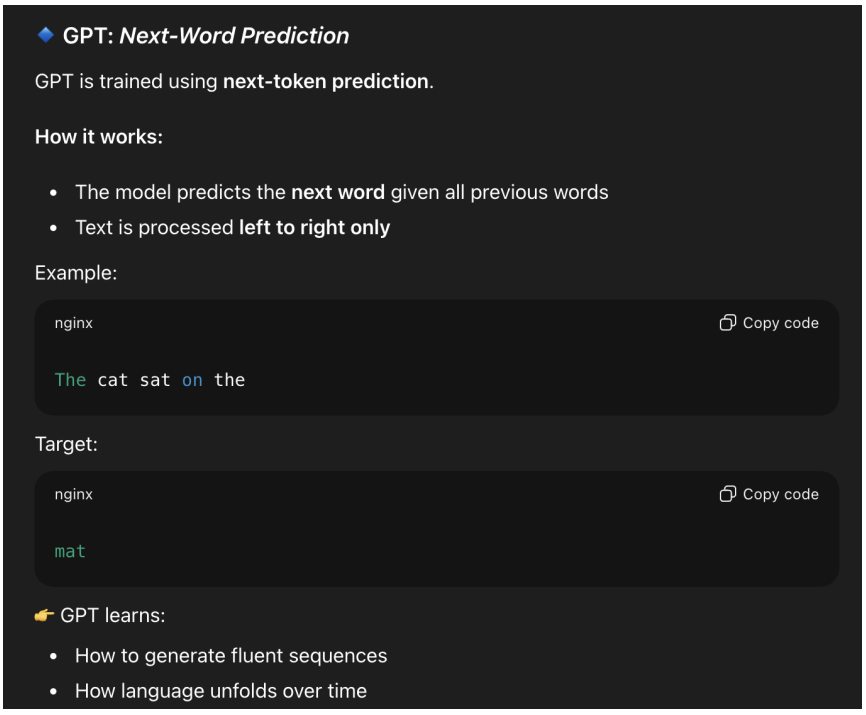
- Deep learning eliminates the need for **manual feature engineering**.'

## BERT vs GPT

| BERT | GPT |
|---|---|
| Encoder-based | Decoder-only |
| Masked word prediction | Next-word prediction |

Classification-focused Generation-focused

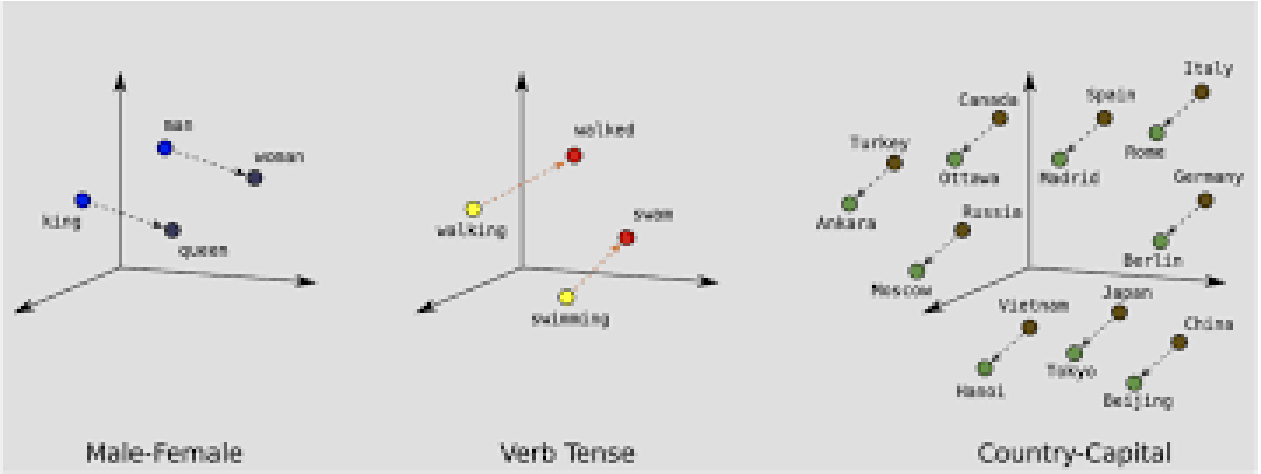



# 2.1 Understanding word embeddings

- Converting text -> numerical

- Deep learning models operates on numerical data, while text is categorical
- Therefore, word **embeddings need to convert words into continuous-valued vectors**
    - LLM embeddings are numerical (vector) representations of words, sentences, or documents that capture their semantic meaning, allowing AI to understand context and relationships between concepts,
- THis allows them to be compatible with the mathematical operations used in neural networks




- Word2Vec
    - Trains a neural network to predict the context of a word given the target word or vice versa
    - Approach assumes that words appearing in similar contexts tend to have similar meanings,
        - Results in clustered representations of related words in the embedding space
    - High dimensionality vs low dimensionality
        - High dimensionality capture more nuanced relationships between words but comes at the cost of computational efficiency
        - Low dimensionality offers faster processing but can sacrifice some semantic detail
- LLM and Embeddings
    - LLMs often generate their own embeddings as a part of the inner layer and optimize them during training
    - Allows embeddings tailored to specific task and data, leading to a better performance than using pre-trained embeddings
- Dimensionality methods
    - Visual perception and common graphical representations are limited to 3 dimensions or less
    - Visualizing high-dimension embeddings requires specialized techniques or dimensionality reduction methods

**Word embeddings: a method of representing words as continuous valued vectors, allowing deep learning models to process text data**

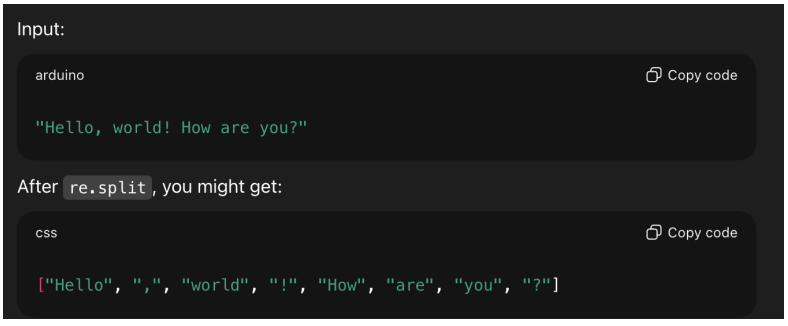
**Word2vec: algorithm that generates word embeddings by predicting the context of a word given the target word and vice versa, based on the idea that words appearing in similar contexts tend to have similar meanings**

**Embedding: the process of converting various data types, such as text, audio or video, into a dense vector representation that deep learning models can understand**

**Embedding size: dimensionality of word embedding, which determines the number of dimensions used to represent each word. Influences the complexity and computational efficiency of the model.**

2.2 Tokenizing text

- Tokenization
    - Preprocessing step for creating embeddings for an LLM
    - Involves: **splitting** input text into individual tokens (either words or special characters) to prepare the text for further processing and embedding creation
    - The “re.split” function can be used to split text based on specific patterns
        - `re.split(pattern, text)` splits a string wherever the pattern matches.
            - Allows for flexible, rule-based splitting
        - Can separate text based on words, punctuations, or whitespaces
        - Can separate words and punctuation into separate tokens
        - Preserves structure
        - Tokens are cleaner and more consistent
            -
    -




- Capitalization
    - Helps LLMs distinguish between proper nouns and common nouns
    - Helps understand sentence structure
    - Learn to generate text with proper capitalization
    - Preserving capitalization during tokenization is beneficial for training effective language models
- Removing whitespaces
    - Removing whitespaces reduces memory and computing requirements.

    - However, preserving whitespaces can be useful for training models that are sensitive to the exact structure of the text, such as python code (since it relies on indentation and spacing)

2.3 Converting Tokens into Token IDs

- Why is it important
    - Converting token into token IDs is an intermediate step before converting them to embedding vectors
    - This step allows for efficient representation and processing of text data within a language model
- Process
    - A vocabulary is created by tokenizing the entire training data set, sorting the unique tokens alphabetically, and assigning a unique integer to each token
    - This mapping allows for more efficient conversion between tokens and their corresponding integer representations
    - Encode method
        - takes text as input, splits it into tokens, and uses the vocabulary to convert these tokens into their corresponding integer IDs.
        - This process allows for **representing text data as a sequence of integers**, which can be processed by the language model
    - Decoding method
        - takes a sequence of token IDs as input and uses the inverse vocabulary to convert these IDs back into their corresponding text tokens.
        - This process allows for converting the output of the language model, which is a sequence of integers, back into human-readable text.
- Problem with small data set
    - Using a vocabulary built from a small training set can lead to issues when encountering new words or phrases not present in the training data.
    - This can result in errors during tokenization and decoding, highlighting the importance of using large and diverse training sets for building robust language models

Definitions

- **Tokenization: splitting text into individual units, called tokens, which can be words, punctuations, or other special characters**

---

- **Tokens: individual units of text that result from tokenization, representing words, punctuation or other special characters**

---

- **Regular expressions: used to define patterns in text, allowing for flexible and precise text manipulation, including tokenization**

---

- **Preprocessing: initial steps to prepare text data for further processing, such as tokenization, which makes the text suitable for use in language models**

2.4 adding special content tokens

- Special tokens
  - <|unk|> and <|endoftext|>. <|unk|>
    - Represents unkown words not in the training data
  - <|endoftext|>
    - Seperates unrelated text sources, helping the LLM understand the nature of the text sources
  - ***What does it mean as unrelated text sources, what does it mean to seperate them?***

`<|endoftext|>` is a **boundary marker**.

It tells the model:

> "STOP. The previous text is finished.
> What comes next is a NEW, UNRELATED text."

Example with separator:

arduino Copy code

```
"The electron has a negative charge.<|endoftext|>To boil pasta, add salt t
```

Now the model learns:

- Physics ends here
- Cooking starts fresh
- No continuity should be assumed

- Encountering new words
  - SimpleTokenizerV2, <|unk|>
  - When encountering a word not in the vocabulary, SimpleTokenizerV2 replaces it with the <|unk|> token, ensuring that all words are represented in the encoded text.
- <|endoftext|> token
  - The <|endoftext|> token acts as a **marker between unrelated text sources**, signaling the start or end of a particular segment. This helps the LLM understand that these texts, though concatenated for training, are distinct entities.
- Other common special tokens
  - [BOS] (beginning of sequence),
  - [EOS] (end of sequence)
  - [PAD] (padding).
    - used to extend shorter texts

**Definitions**

- **Vocabulary: a mapping from unique tokens to unique integer values, created by tokenizing the entire training dataset and sorting the tokens alphabetically**

- **Token ID: integer representation of tokens, used as an intermediate step before converting tokens into embedding vectors**
- **Tokenizer: a class that implements methods for encoding text into token IDs and decoding token IDs back into text**
- **Training set: the data set used to build the vocabulary and train the model**

**2.5 byte pair encoding**

- BFE tokenizers
    - **Break down unknown words into smaller subword units or individual characters**
    - Allows handling of any word without needing special l <|unk|> token
    - **Ensures that the tokenizer and the LLM can process any text, even if it contains words not present in the training data**
- **The BPE tokenizer used in these models has a vocabulary size of 50,257, with the <|endoftext|> token assigned the largest token ID.**
- The BPE tokenizer breaks down unknown words into smaller subword units or individual characters. This allows it to represent any word as a sequence of known subword tokens or characters, enabling it to process any text without needing a special token for unknown words.
- The code uses the **tiktoken library**, which is an open-source Python library that efficiently implements the BPE algorithm based on Rust code

2.6 Data sampling with sliding window

- input-pairs
    - Input-pairs are essential for training LLMs because they provide the model with examples of text sequences with their corresponding next words.
    - This allows the LLM to learn the relationship between words and predict the most
- Sliding window
    - The sliding window approach involves iterating through a text sequence and extracting overlapping chunks of text as inputs.
    - Each input chunk is paired with the corresponding next word as the target
    - The window slides across the text, creating multiple input-target pairs for training
- Strider
    - The stride parameter determines the **step size of the sliding window.**
    - A smaller stride results in more overlapping input chunks, while a larger stride creates less overlap.
    - The choice of stride influences the amount of data generated and the potential for capturing long-range dependencies in the text
- Max_length
    - The max_length parameter defines the **size of the input chunks** extracted from the text.
    - It **determines the number of tokens included** in each input sequence.
    - A larger max_length allows the LLM to process longer contexts, but it also increases the computational cost of training.
- PyTorch's Dataset and DataLoader

- PyTorch's Dataset and DataLoader classes provide a convenient and efficient way **to manage and iterate over large datasets.**
- They allow for **batching**, **shuffling**, and **parallel data loading**, which are crucial for optimizing the training process of LLMs.

**Definitions**

- **Byte pair encoding: a tokenization scheme that breaks down words into smaller subword units or individual characters, allowing it to handle unknown words by representing them as sequences of a subword tokens or characters**
- **Subword units: smaller units of text that a BPE tokenizer breaks words into, which can be individual characters or combinations of characters**
- **out-of-vocabulary words: words that are not present in the tokenizer's predefined vocabulary**
- **Vocabulary size: total number of unique tokens that a tokenizer can recognize and represent**

**2.7 Creating Token Embeddings**

- Embedding vectors
  - essential for training GPT-like LLMs because these models are deep neural networks that rely on the backpropagation algorithm for learning.
  - Backpropagation requires continuous vector representations, which embedding vectors provide.
  - ***What is backpropagation, what are embedding vectors***

# 1️⃣ What is Backpropagation?

## Intuition first (plain English)

> **Backpropagation is how a neural network learns from its mistakes.**

The model:

1. Makes a prediction
2. Checks how wrong it is
3. Sends that error backward through the network
4. Slightly adjusts its internal weights to do better next time

- ***Embedding vectors are numerical representations of words (or tokens) that capture their***

## 2️⃣ What is an embedding vector?

An **embedding vector** is the **numerical representation of a token**.

Example:

```
css

Token ID 1532 → [0.12, -0.87, 0.33, ..., 0.05]
```

So the pipeline is:

```
arduino

text → token → token ID → embedding vector
```

👉 The embedding vector is what the neural network actually works with.

***meaning in a continuous space.***

- Embedding weights
  - Embedding weights are initially assigned random values.
  - These random values serve as the **starting point** for the LLM's learning process.
  - During training, the embedding weights are **optimized through backpropagation** to improve the model's performance.
- Embedding Layer
  - The embedding layer acts as a **lookup table**.
  - When given a token ID, it **retrieves the corresponding embedding vector from its weight matrix.**
  - This embedding vector is a continuous representation of the token, allowing the LLM to process it effectively.
- Weight matrix
  - The embedding layer's weight matrix has a number of rows equal to the vocabulary size, representing each unique token.
  - The number of columns corresponds to the embedding dimension, which determines the size of the embedding vector for each token.

Definition:

- **Context size: number of tokens that the LLM uses as input to predict the next word**
- **Input-target pairs: a set of data used to train an LLM, where the input is a sequence of tokens and the target is the next in the sequence**

- **Sliding window: a technique used to create input-target pairs from a text dataset by moving a window of tokens across the text**
- **Stride: number of positions the input window is shifted when creating the next batch of input-target pairs**

**2.8 Encoding word positions**

1. LLMs' self-attentional mechanism: Positional embeddings
    a. lacks a notion of token order.
    b. To address this, **positional embeddings** are introduced, which provide information about the position of each token within a sequence.
2. Absolute and relative positional embeddings
    a. Absolute embeddings assign a unique embedding to each position in a sequence, indicating the **exact location**
    b. **Relative positional embeddings** focus on the **relative distance** between tokens, allowing the model to **generalize better to sequences of varying lengths.**
3. GPT and Positional Embeddings
    a. GPT models use absolute positional embeddings that are optimized during training.
    b. These **embeddings are not fixed or predefined** but are **learned alongside the model's other parameters.**
4. Embedding process
    a. Token embeddings are generated by **mapping token IDs to vectors**.
    b. Positional embeddings are then added to these token embeddings, resulting in input embeddings that **incorporate both token identity and positional information.**
5. Layers
    a. The **token_embedding_layer** converts token IDs into embedding vectors,
    b. while the **pos_embedding_layer** generates positional embeddings based on the position of each token in the sequence.

$$\text{input embedding}_i = \text{token embedding}_i + \text{positional embedding}_i$$

# 3.1 The problem with modeling long sequences

- Challenge
    - Lies in the grammatical structures of different languages
    - Direct word-by-word translation often fails to capture the meaning and context due to differences in sentence structure and word order
- Encoding-Decoding process
    - The encoder processes the entire input text and encodes its meaning into a hidden state.
    - The decoder then uses this hidden state to generate the translated text, one word at a time
- Encoder-decoder RNNs

    - Encoder-decoder RNNs rely solely on the current hidden state during decoding, which can lead to a loss of context, especially when dependencies span long distances in complex sentences.
- Hidden state
    - The hidden state is a compressed representation of the input sequence, capturing the meaning of the entire text. It acts as a memory cell that the decoder uses to generate the translated output.

**The challenge in machine translation arises from differences in grammatical structure and word order across languages, making direct word-by-word translation ineffective for preserving meaning and context. To address this, encoder–decoder RNN models use an encoding–decoding process in which the encoder compresses the entire input sentence into a fixed-size hidden state. This hidden state acts as a memory representation of the input sequence and is passed to the decoder, which generates the translated output word by word. However, because the decoder relies solely on this single hidden state, information can be lost, especially in long or complex sentences where important dependencies span long distances.**

# 1️⃣ What is the hidden state?

## Plain-English intuition

> **The hidden state is the model's internal memory at a given time step.**

It's a **vector of numbers** that tries to summarize:

- What the model has read so far
- The meaning and context of the sequence up to that point

You can think of it as:

- A **compressed note** the model writes to itself
- A **memory snapshot** of the sentence so far

3.2 Capturing Data Dependencies with Attention Mechanisms
- RNNs struggle with long texts because
    - They need to remember the entire encoded input in a single hidden state before encoding
    - This makes it difficult to retain information from earlier parts of the input
- Why self attention is important

- It is crucial component of LLMs because of the transformer architecture, enabling them to capture long-range dependencies and understand the relationships between words in a sentence

**RNNs struggle with long texts because encoder–decoder architectures compress the entire input sequence into a single hidden state, which the decoder relies on during generation. This fixed-size representation makes it difficult to retain information from earlier parts of long or complex sentences, as important details may be lost during compression.**
**Self-attention addresses this limitation by allowing each word to directly attend to other words in the sentence, enabling the model to capture long-range dependencies and understand relationships between words.**
**By dynamically weighting which words are most relevant, self-attention provides a more effective representation of sentence-level context.**

**Definitions**

- **Attention Mechanism**
  - **A technique which allows a neural network to focus on specific parts of an input sequence when generating an output**
  - **Enables it to capture long-range dependencies and improve performances on tasks like machine translation**
- **Self attention**
  - **Type of attention mechanism where each element in a sequence can attend to all other elements in the same sequence, allowing the model to learn relationships and dependencies within the output**
- **Transformer structure**
  - **Neural network architecture that relies on self-attention mechanisms to process sequential data, enabling it to handle long-range dependencies and outperform traditional RNNs in tasks like machine translation and text generation**
- **GPT series**
  - **Family of large language models based on the transformer architecture, known for their impressive capabilities in generating human-like text, translating languages and performing various other language-related tasks**

3.3 Attending to different parts of the input with self-attention

- Self in input
  - The 'self' in self-attention refers to the mechanism's ability to compute **attention weights by relating different positions** within a single input sequence.
  - It assesses and **learns the relationships and dependencies** between various parts of the input itself
- Context vector
  - A context vector is an enriched embedding vector that **incorporates information from all other elements in the input sequence.**

        - It represents an enhanced understanding of each element by **considering its relationships with other elements.**
- Attention scores
    - Attention scores are intermediate values that **represent the similarity** between the query element and each other element in the input sequence.
        - When a model processes one element (token) in a sequence, it needs to answer:
            - **“Which other elements are relevant to me right now?”**
    - They are calculated using dot products and indicate the degree of attention or focus on each element.
        - Dot product: a mathematical operation that takes **two vectors** and returns **a single number (a scalar)**. It tells you **how similar or aligned the two vectors are**.
- Normalization
    - Normalization is applied to attention scores to obtain attention weights that sum up to 1.
    - This normalization is a convention that is useful for interpretation and maintaining training stability in an LLM.
- dot()
    - The dot() function has been removed. This calculates the dot product of the query
- Softmax function and normalization
    - The softmax function is used to normalize attention scores, ensuring that the attention weights are always positive and sum to 1. It provides a more stable and interpretable representation of attention weights as probabilities or relative importance
- Softmax and normalize
    - Softmax function is used to normalize attention scores, ensuring that the attention weights are always positive and sum to 1
    - Softmax function converts a set of real numbers into a probability distribution
    - Provides a more stable and interpretable representation of attention weights as probabilities or relative importance
- How to calculate context vectors
    - Context vectors are calculated by multiplying the embedded input tokens with their corresponding attention weights and then summing the resulting vectors.
    - This weighted sum combines information from all input elements, creating an enriched representation of each element.

Self-attention allows a model to focus on different parts of the same input sequence when processing a token. The term *“self”* refers to the fact that attention is computed by comparing each token with other tokens within the **same sentence**, enabling the model to learn relationships and dependencies among them.

Attention scores represent how similar or relevant other tokens are to the current token being processed. When the model considers a particular token (the query), it asks which other tokens in the sequence are important for understanding it. These attention scores are calculated using the dot product, which measures how closely related two vectors are.

The raw attention scores are then normalized using the softmax function. This normalization ensures that all attention weights are positive and sum to one, allowing them to be interpreted as relative importance values and helping maintain training stability.

A context vector is an enriched version of a token's embedding that incorporates information from all other tokens in the input sequence. It is computed by multiplying each token's embedding by its corresponding attention weight and summing the results. In this way, tokens that are more relevant contribute more strongly to the context vector.

Unlike basic embedding vectors, which only represent a token's identity, context vectors capture how a token relates to other tokens in the sentence. As a result, self-attention enables the model to understand long-range dependencies and determine which words are most important in the sentence as a whole.

**Definitions**

- **Context vector**
    - **An enriched embedding vector that incorporates information about a specific input element and all other elements in the input sequence, creating a more comprehensive representation**
- **Attention scores**
    - **Intermediate values calculated as dot products between the query element and all other input elements, representing the similarity or attention between them**
- **Attention weights**
    - **Normalized attention scores that sum up to 1, representing the relative importance or contribution of each input element to the context vector.**

3.4 Implementing self-attention with trainable weights

- Matrices and value vectors
    - These matrices are used to project the embedded input tokens into query, key, and value vectors, respectively. This projection allows the model to learn relationships between different parts of the input sequence and determine the importance of each input element for generating context vectors.
- How to calculate attention scores
    - Attention scores are calculated by taking the dot product of the query vector with each key vector. These scores are then normalized using the softmax function to obtain attention weights, which represent the relative importance of each input element for the current query.
- Scaling
    - Scaling by the square root of d_k helps to prevent small gradients during backpropagation, especially when dealing with large embedding dimensions. This scaling ensures that the softmax function operates in a more stable range, leading to more effective model training.
- Classes?

- Both classes implement the self-attention mechanism. A, _v1 uses manual weight initialization with nn.Parameter, while SelfAttention_v2 utilizes nn.Linear layers for weight matrices, which offers optimized weight initialization and improved training stability.

In self-attention, the embedded input tokens are first projected into three different vector representations called **queries**, **keys**, and **values**. This projection is performed using learned weight matrices, which allow the model to transform the same input embeddings in different ways depending on their role in attention. Queries represent the token currently being processed, keys represent all tokens that can be attended to, and values contain the information that will be combined to form the final contextual representation. By learning these projections, the model can identify relationships between different parts of the input sequence and determine which tokens are most important for understanding a given token.

Attention scores are computed by taking the **dot product** between a query vector and each key vector. These dot products measure how similar or relevant each key is to the query. The resulting raw scores indicate how much attention the current token should pay to every other token in the sequence. To convert these scores into meaningful importance values, they are passed through the **softmax function**, which normalizes them so that they are all positive and sum to one. These normalized values are called **attention weights** and represent the relative importance of each token for the current query.

Before applying the softmax function, the attention scores are scaled by dividing them by the square root of the key dimension dkd_kdk. This scaling is necessary because dot products grow larger when vector dimensions increase, which can cause the softmax function to produce extremely small gradients. Scaling keeps the values within a stable range, preventing gradient vanishing during backpropagation and leading to more effective and stable training.

Once the attention weights are obtained, the **context vector** is computed as a weighted sum of the value vectors. Each value vector contributes information to the final context vector in proportion to its attention weight. This produces a contextualized representation of each token that incorporates relevant information from the entire input sequence.

In practice, self-attention can be implemented in different ways. One implementation manually defines the query, key, and value weight matrices using trainable parameters, giving full control over their initialization. Another implementation uses built-in linear layers to perform these projections, which provides optimized weight initialization and improved numerical stability. While both approaches implement the same self-attention mechanism, using linear layers is generally preferred for cleaner design and more stable training.

3.5 Hiding future words with causal attention

1. What is casual attention
    a. Causal attention, also known as masked attention, restricts a model to consider only previous and current inputs in a sequence when processing any given token. This is in contrast to standard self-attention, which allows access to the entire input sequence at once

2. Diagram vs causal attention
    a. The diagram shows that in causal attention, we mask out the attention weights above the diagonal so that for a given input, the LLM can't access future tokens when computing the context vectors using the attention weights. (Figure 3.19)
3. Casual attention mask
    a. The causal attention mask is applied to the attention weights by zeroing out the elements above the diagonal, effectively preventing the model from attending to future tokens. This ensures that the model's predictions are based solely on past and current information.
4. Information leakage and masked positions
    a. Information leakage occurs when masked positions still influence the softmax calculation. However, renormalizing the attention weights after masking effectively nullifies the effect of masked positions, ensuring that there is no information leakage from future tokens.
5. Dropout technique
    a. Dropout is a technique used to prevent overfitting by randomly dropping out hidden layer units during training. In causal attention, dropout is typically applied after calculating the attention weights, randomly zeroing out some of the weights and scaling up the remaining ones.
6. Register_buffer method
    a. The register_buffer method ensures that the causal mask is automatically moved to the appropriate device (CPU or GPU) along with the model, avoiding device mismatch errors during training.

Definition

- Casual attention
    - A specialized form of self-attention that restricts a model to only consider previous and current inputs in a sequence when processing any given token
- Trainable weight matrices
    - Used to project input tokens into query, key, and value vectors, which are then used to compute attention scores and weights
- Mask
    - Used to selectively hide certain values during computation, often used to implement causal attention

3.6 Extending single-head attention to multi-head attention

- Multi-head attention allows LLMs to process information from different perspectives by running the attention mechanism multiple times with different learned linear projections. This enables the model to capture more complex patterns and relationships within the input data.
- The MultiHeadAttentionWrapper class creates multiple instances of the CausalAttention module, each representing a separate attention head. It then combines the outputs from these heads by concatenating them.
- The MultiHeadAttentionWrapper stacks multiple single-head attention modules, while the MultiHeadAttention class integrates multi-head functionality within a single class. The MultiHeadAttention class splits the input into multiple heads by reshaping the projected query, key, and value tensors and then

- The output projection layer in the MultiHeadAttention class is used to project the combined outputs from all attention heads back to the original embedding dimension. This layer is not strictly necessary but is commonly used in many LLM architectures.
- The MultiHeadAttention class is more efficient because it performs matrix multiplications for the queries, keys, and values only once, instead of repeating them for each attention head as in the MultiHeadAttentionWrapper.

Causal attention, also called **masked self-attention**, is a mechanism that ensures a language model can only attend to **past and current tokens** when processing a sequence. Unlike standard self-attention, which allows every token to attend to all other tokens in the sequence, causal attention explicitly prevents a token from accessing information from future positions. This constraint is essential for autoregressive language models, such as GPT, where text is generated one token at a time and future words should not influence the prediction of the current word.

In causal attention, this restriction is implemented using a **causal attention mask**. Conceptually, attention weights are arranged in a matrix where each row corresponds to a query token and each column corresponds to a key token. To prevent access to future tokens, all attention weights **above the diagonal** of this matrix are masked out. This means that for any given token position, the model can only attend to tokens at the same position or earlier positions in the sequence. As a result, when computing context vectors, the model is unable to incorporate information from tokens that come later in the sequence.

The causal attention mask works by setting the attention scores for future positions to very large negative values (or effectively zero after masking). This is done before applying the softmax function. Without proper handling, masked positions could still influence the output through the softmax normalization process, leading to **information leakage**, where future tokens indirectly affect current predictions. To avoid this, the attention scores are re-normalized after masking using softmax, which ensures that masked positions receive zero probability and contribute no information to the context vector.

Dropout is often applied after attention weights are computed in causal attention. This technique randomly removes some attention connections during training, forcing the model to rely on multiple contextual cues rather than overfitting to specific token relationships. The remaining attention weights are scaled accordingly so that the overall magnitude of information remains stable. Applying dropout at this stage improves generalization while preserving the causal structure of the model.

Finally, the **register_buffer** method is used to store the causal attention mask within the model. This ensures that the mask is treated as a persistent, non-trainable component of the model and is automatically moved to the correct device (such as CPU or GPU) during training or inference. This prevents device mismatch errors and simplifies model deployment without affecting the learning process.

Definitions

- Masked attention

    - A specialized form of self-attention that restricts a model to only consider previous and current inputs in a sequence when processing any given token
- Dropout
    - A technique used during training to randomly ignore hidden layer units, preventing overfitting by ensuring that a model does not become overly reliant on any specific set of units
- Multi-head attention
- The process of dividing the attention mechanism into multiple independent heads, each with its own set of weights to enhance pattern recognition and improve model performance.

Multi-head attention extends single-head self-attention by allowing the model to attend to the input sequence **from multiple perspectives simultaneously**. Instead of computing attention once, the model **runs the attention mechanism multiple times in parallel** using different learned linear projections for the queries, keys, and values. Each attention head can **focus on different types of relationships** within the input, such as syntactic structure, semantic similarity, or long-range dependencies. By combining these different attention patterns, the model is able to capture more complex and nuanced representations of the input data.

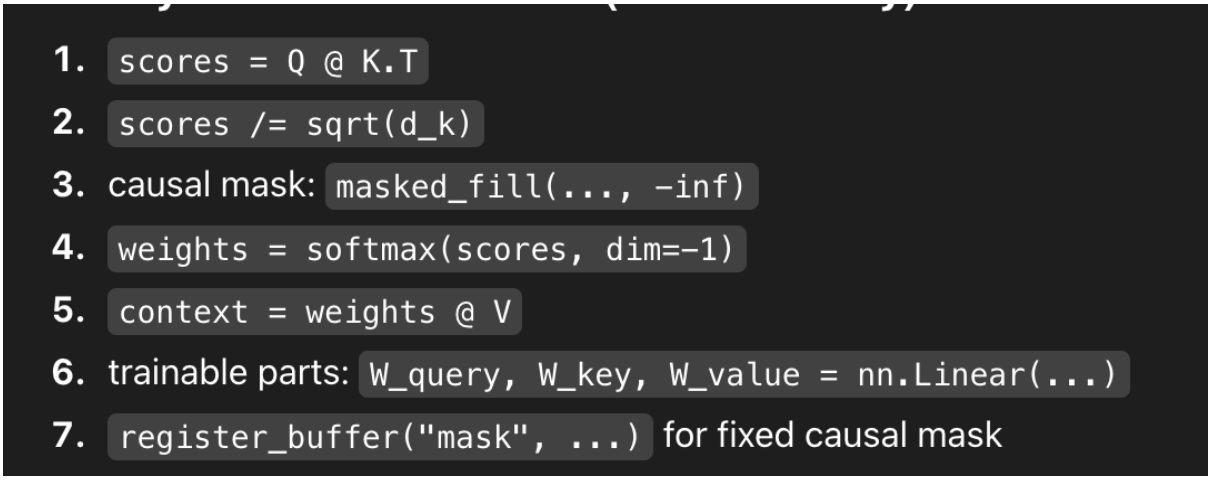
One way to implement multi-head attention is through a **wrapper approach**, where **multiple independent single-head attention modules are created**. In this design, each attention head is a separate instance of a causal attention module, and the outputs of all heads are concatenated to form a combined representation. This approach is conceptually simple and clearly shows how multiple attention heads operate independently, but it is less computationally efficient because the query, key, and value projections are computed separately for each head.

A more efficient implementation integrates multi-head attention within a single class. In this approach, the input embeddings are first projected into queries, keys, and values **once**, and these projections are then reshaped to split them into multiple attention heads. Each head operates on a different slice of the projected tensors, allowing attention to be computed in parallel within the same module. This design avoids redundant matrix multiplications and is therefore more computationally efficient than stacking multiple single-head modules.

After attention is computed independently for each head, the resulting outputs are concatenated and passed through an **output projection layer**. This linear layer maps the combined multi-head representation back to the original embedding dimension, ensuring compatibility with subsequent layers in the model. Although this output projection is not strictly required, it is commonly used in large language model architectures because it allows the model to mix information across heads and maintain a consistent embedding size throughout the network.

Overall, multi-head attention enhances the expressive power of self-attention by enabling the model to capture diverse relationships in the input sequence, while integrated implementations improve efficiency by sharing projection computations across heads.

dot = score. -> softmax = weights -> added together = context vector




## 4.1 Coding an LLM structure

- GPT model
  - A GPT model is designed to **generate new text one word at a time**.
  - It achieves this by using a **large deep neural network** architecture that **learns patterns** from a massive dataset of text and then uses these patterns to **predict** the next word in a sequence.
- Key components of GPT
  - The key components of a GPT model include **embedding layers, transformer blocks, and an output layer**.
  - **Embedding** layers convert words into numerical representations,
  - **transformer** blocks process these representations to capture relationships between words, and the
  - The **output** layer predicts the probability of each word in the vocabulary.
- Parameters in LLMs
  - Parameters in LLMs refer to the trainable weights of the model.
  - These weights are adjusted during training to minimize a loss function, allowing the model to learn from the training data and improve its ability to generate text.
- GPT-3
  - GPT-3 is a larger model with more parameters and trained on more data than GPT-2.
  - However, GPT-2 is more suitable for learning LLM implementation because its pretrained weights are publicly available, and it can be run on a single laptop computer, unlike GPT-3, which requires a GPU cluster.
- GPT_config_124m
  - The GPT_CONFIG_124M dictionary defines the configuration of the small GPT-2 model.
  - It includes **parameters** like vocabulary size, context length, embedding dimension, number of attention heads, number of layers, dropout rate, and

**query-key-value bias**, which determine the model's architecture and behavior.

- Dummy GPT
    - The DummyGPTModel class provides a **placeholder architecture** for the GPT model, outlining the main components and their order.
    - It serves as a **starting point for building the full GPT** model by defining the data flow and providing a framework for implementing the individual components.

4.2 Normalizing activations with layer normalization

- Layer normalizations
    - Layer normalization aims to **stabilize and accelerate neural network** training by **adjusting the activations** of a layer to have a **mean of 0 and a variance of 1**.
    - This normalization process helps prevent vanishing or exploding gradients, ensuring consistent and reliable training.
    - No clue what “vanishing or exploding gradients" mean, no `clue why this helps stabilise.
- When to apply layer normalizations
    - Layer normalization is commonly applied both before and after the multihead attention module in GPT-2 and modern transformer architectures.
    - It is also applied before the final output layer
- Biased variance and unbiased variance
    - Biased variance calculation divides by the number of inputs *n*,
    - unbiased variance uses *n* – *1* in the denominator to correct for bias.
    - In LLMs, where the embedding dimension *n* is large, the difference between these approaches is negligible, and the biased method is preferred for compatibility with GPT-2's normalization layers and TensorFlow's default behavior.
    - What is biased and unbiased variance? What is their main use?
- Layer normalization vs batch normalization
    - Layer normalization normalizes across the feature dimension
    - batch normalization normalizes across the batch dimension.
    - Layer normalization offers greater flexibility and stability, especially in scenarios with varying batch sizes or resource constraints, making it suitable for LLMs.

Layer normalization is a technique used in large language models to stabilize and speed up training by normalizing the activations within a layer so that they have a mean of zero and a variance of one. This helps prevent **vanishing gradients**, where gradients become too small to meaningfully update model weights, and **exploding gradients**, where gradients become excessively large and destabilize training. By keeping activation values within a controlled range, layer normalization ensures that gradients remain well-scaled during backpropagation, leading to more stable and reliable learning. In transformer architectures such as GPT-2, layer normalization is typically applied before and after multi-head attention blocks and before the final output layer to maintain numerical stability throughout the network.

Variance can be computed in two ways: **biased variance**, which divides by the number of inputs *n*, and **unbiased variance**, which divides by *n − 1* to correct for statistical bias in small samples. In large language models, the embedding dimension *n* is very large, making the difference between biased and unbiased variance negligible. As a result, the biased variance is preferred for efficiency and consistency with existing implementations such as GPT-2 and TensorFlow's default normalization behavior.

Layer normalization differs from batch normalization in how normalization is applied. **Layer normalization normalizes across the feature dimension for each individual input**, while **batch normalization normalizes across the batch dimension** using statistics computed from multiple examples. Because language models often operate with variable sequence lengths, small batch sizes, or even batch size one during inference, layer normalization provides greater flexibility and stability, making it better suited for transformer-based LLMs.

**Definition**

- **Transformer block:**
    - **The core building block of a gpt model, consisting of masked multi-head attention module and a feedforward neural network, which are applied sequentially to input data**
    - What is feedforward neural network
        - A feedforward neural network takes an input, transforms it through layers, and produces an output — with information flowing in only one direction.
- **Layer normalization**
    - **Technique used to normalize the output of each layer in a neural network, ensuring that the data has a mean of zero and a standard deviation of 1**
    - **Helps to improve stability and performance of model**

- **Parameters**
    - **Trainable weights of a neural network model, which are adjusted during the training process to minimise a specific loss function**
- **Logits:**
    - **The output of a neural network before applying the softmax function, represents the normalised probability of each possible output class**

4.3 Implementing a feed forward network with GELU activations

- GELU
    - The GELU activation function is a smooth, nonlinear function that approximates ReLU but with a non-zero gradient for almost all negative values.
    - Unlike ReLU, which outputs zero for any negative input, GELU allows for a small, non-zero output for negative values.
    - **What is GELU, what is ReLU**
- Feedforward module

        - The FeedForward module is a small neural network consisting of two linear layers and a GELU activation function.
        - It expands the embedding dimension into a higher-dimensional space, applies a nonlinear transformation, and then contracts back to the original dimension, allowing for richer representation learning.
        - **What does it mean to expand to a higher dimensional space, and how does that allow for richer representation learning**
        - This allows the model to capture more complex relationships within the data.
- Uniformity in input and output
    - The uniformity in input and output dimensions simplifies the architecture by enabling the stacking of multiple layers without the need to adjust dimensions between them, making the model more scalable.

In transformer-based language models, the feedforward module is a small neural network applied to each token independently and is responsible for learning richer feature transformations after self-attention. It consists of two linear layers with a **GELU (Gaussian Error Linear Unit)** activation function in between. GELU is a smooth, nonlinear activation that behaves similarly to ReLU but differs in that it does not completely discard negative inputs; instead, it allows small, non-zero outputs for negative values, which leads to smoother gradients and more stable training. ReLU, by contrast, outputs zero for all negative inputs, which can cause neurons to become inactive during training. The feedforward module first **expands the embedding dimension** into a higher-dimensional space, applies a nonlinear transformation using GELU, and then **projects the representation back** to the original embedding dimension. This expansion allows the model to represent more complex patterns by giving it more capacity to combine and transform features before compressing them again. Keeping the input and output dimensions the same ensures architectural uniformity, allowing multiple transformer layers to be stacked efficiently without requiring additional dimension-matching operations, which improves scalability and simplifies model design.

### Step 5: Concrete analogy (very effective)

Think of reading a paragraph:

- **Self-attention** → highlighting relevant sentences
- **Feedforward network** → interpreting and rephrasing what those highlights mean

Attention selects.

FFN transforms.

**After self-attention integrates contextual information from other tokens, the feedforward network applies nonlinear transformations to refine and enrich each token's representation, enabling the model to learn more complex feature combinations.**

## What does GELU actually do?

### Short answer (intuition)

> **GELU decides how much of each feature should pass through, in a smooth and probabilistic way.**

It controls **which signals are emphasized, weakened, or softly suppressed** after self-attention.

**A feedforward network enhances token representations by applying nonlinear transformations after self-attention. Within this network, the GELU activation function smoothly scales features, allowing important signals to pass strongly while softly suppressing less useful ones, which improves learning efficiency and stability.**

4.4 Adding shortcut connections

- Vanishing gradient problem
    - The vanishing gradient problem occurs when gradients become progressively smaller as they propagate backward through the layers of a deep neural network
    - This makes it difficult to effectively train earlier layers. This can hinder the learning process and prevent the model from achieving optimal performance.
    - **What are gradients**
- Shortcut connections
    - Shortcut connections, also known as skip connections, create alternative paths for gradients to flow through the network by bypassing certain layers.
    - This helps preserve the flow of gradients during the backward pass, mitigating the vanishing gradient problem and enabling more effective training of deeper networks.
- Implementing shortcut connections
    - In the code, shortcut connections are implemented by adding the output of a layer to the output of a later layer.
    - This is done conditionally based on the use_shortcut attribute, allowing for the inclusion or exclusion of shortcut connections during the forward pass.
- Print gradients function
    - The print_gradients function calculates and displays the mean absolute gradient values for each layer in the model.
    - By comparing the gradient values of models with and without shortcut connections, we can see that shortcut connections help maintain a more consistent gradient flow across layers, preventing the gradients from vanishing in earlier layers.

    - The bigger gradient values on the right are because of the shortcut connections -these stop the gradient values from getting vanishingly small through the layers.

In neural networks, **gradients** are numerical signals that indicate how much each model parameter (weight) should change in order to reduce prediction error. During training, these gradients are computed through backpropagation and flow backward from the output layer to earlier layers. The **vanishing gradient problem** occurs when gradients become progressively smaller as they propagate backward through many layers, causing early layers to receive extremely weak learning signals. As a result, these layers update very slowly or stop learning altogether, which limits the performance of deep models. **Shortcut connections**, also known as skip connections or residual connections, address this problem by creating alternative pathways that allow gradients to flow directly across layers, bypassing intermediate transformations. In practice, this is implemented by adding the input of a layer to the output of a later layer during the forward pass. This direct addition ensures that gradient information can travel backward without being repeatedly scaled down, preserving stronger gradients in earlier layers. When analyzing gradient values using a function such as `print_gradients`, models with shortcut connections consistently show larger and more evenly distributed gradient magnitudes across layers. These larger gradient values demonstrate that shortcut connections successfully prevent gradients from becoming vanishingly small, enabling deeper networks—such as transformers—to train more effectively and reliably.

4.5 Connecting attention and linear layers in a transformer block

- Transformer block
    - A transformer block consists of multi-head attention, feed forward layers, layer normalization, and dropout.
    - The multi-head attention analyzes relationships between elements in the input sequence, while the feed forward network modifies the data individually at each position.
    - Layer normalization ensures consistent scaling, and dropout prevents overfitting.
- Pre-LayerNorm
    - Pre-LayerNorm refers to applying layer normalization before the multihead attention and feed forward layers.
    - This approach, unlike PostLayerNorm, has been shown to improve training dynamics and lead to better performance in transformer models.
- Shortcut connections
    - Shortcut connections add the input of the block to its output, allowing gradients to flow more easily through the network during training.
    - This helps to prevent vanishing gradients and improves the learning of deep models.
- Transformer block preservations
    - The transformer block maintains the input dimensions by applying operations that do not alter the shape of the input sequence.
        - This allows for a one-to-one correspondence between input and output vectors, enabling its use in various sequence-to-sequence tasks.

    - While the transformer block preserves the physical dimensions of the input sequence, the content of each output vector is re-encoded to incorporate contextual information from across the entire input sequence.
        - This allows the model to capture complex relationships between elements in the sequence.

A transformer block is the fundamental building unit of transformer-based language models and consists of **multi-head self-attention**, **feedforward (linear) layers**, **layer normalization**, **dropout**, and **shortcut (residual) connections**. The multi-head attention mechanism enables the model to analyze relationships and dependencies between different elements in the input sequence, allowing each token to incorporate contextual information from other tokens. After attention mixes information across the sequence, the feedforward network processes each token **independently**, applying nonlinear transformations to refine and enrich its representation. Layer normalization is applied to stabilize training by keeping activations well-scaled, while dropout helps prevent overfitting by randomly deactivating parts of the network during training. In **Pre-LayerNorm** architectures, layer normalization is applied *before* the attention and feedforward sublayers, which improves gradient flow and training stability compared to Post-LayerNorm designs. Shortcut connections add the input of each sublayer to its output, ensuring that gradients can flow directly through the network and preventing vanishing gradients in deep models. Importantly, transformer blocks **preserve the input dimensions**, meaning there is a one-to-one correspondence between input and output vectors. Although the shape of the sequence remains unchanged, the **content of each output vector is re-encoded**, incorporating global contextual information from the entire sequence, which allows the model to capture complex linguistic and structural relationships.

Coding the GPT model

- GPT model Class
    - The GPTModel class assembles the complete GPT architecture, utilizing the previously defined TransformerBlock class as a building block.
    - It combines token and positional embeddings, applies multiple TransformerBlock layers, and finally projects the output into the vocabulary space to predict the next token
- LayerNorm layer
    - The LayerNorm layer is applied after the transformer blocks to normalize the output, ensuring that the data has a consistent scale and distribution.
    - This helps to stabilize the learning process and improve the model's performance.
- Weight tying
    - Weight tying is a technique where the weights of the token embedding layer are reused in the output layer.
    - This reduces the number of trainable parameters, leading to a smaller model footprint and potentially faster training.
- Output of GPT model

    - The output of the GPTModel is a tensor of shape [batch_size, num_tokens, vocab_size], representing the logits for each token in the vocabulary.
    - To convert these logits into text, we need to apply a softmax function to obtain probabilities, then select the token with the highest probability for each position in the sequence.
- Increasing number of transformer blocks
    - Increasing the number of transformer blocks in a GPT model generally leads to a larger model with more parameters, requiring more computational resources. However, it can also improve the model's ability to capture long-range dependencies in the input text, potentially leading to better performance on tasks like text generation.

The `GPTModel` class assembles the complete GPT architecture by stacking multiple **TransformerBlock** modules as its core building components. It begins by converting token IDs into dense numerical representations using **token embeddings**, which are then combined with **positional embeddings** to encode both word identity and order information.

The resulting embeddings are passed through a sequence of transformer blocks, where each block **progressively refines** the representations by integrating contextual information through self-attention and nonlinear feedforward transformations.

After the transformer blocks, a **LayerNorm** layer is applied to normalize the final representations, ensuring consistent scale and stable training behavior. The normalized outputs are then projected into the vocabulary space using an output linear layer, producing a tensor of shape [batch_size,num_tokens,vocab_size][ \text{batch_size}, \text{num_tokens}, \text{vocab_size} ]**[batch_size,num_tokens,vocab_size**] that contains logits representing the model’s unnormalized predictions for each possible next token.

During text generation, these logits are converted into probabilities using the softmax function, and a token is selected at each position based on these probabilities. **Weight tying** is commonly used in GPT models, where the same weight matrix is shared between the token embedding layer and the output projection layer, reducing the total number of trainable parameters and improving efficiency without sacrificing performance. Increasing the number of transformer blocks results in a deeper and more expressive model capable of capturing longer-range dependencies in text, though it also increases computational cost and memory requirements.

**Definitions**

- **Multi-head attention**
    - **Allows the model to attend to different parts of the input sequence simultaneously, capturing complex relationships between words and phrases**
- **Layer normalization**
    - **A technique that normalizes the activations of a layer to have a mean of zero and a standard deviation of 1, improving stability and performance of the model**
- **Shortcut connection**

        - **Allows the gradient to flow directly from the input to the output layer, preventing the vanishing gradient problem and enabling the training of deeper models.**

4.7 Generating text

- GPT model conversion
    - The GPT model converts its output tensors into text by **decoding them, selecting tokens** based on a probability distribution derived from the softmax function, and then **converting these tokens back** into humanreadable text.
        - The softmax function transforms the output logits into a probability distribution, where each value represents the likelihood of a particular token being the next in the sequence.
        - **This allows the model to select the most probable token for generation.**
- Generate_text_simple function
    - The generate_text_simple function implements a simple generative loop for a language model.
    - It iteratively **predicts** the next token based on the current context, **appends** it to the input sequence, and **repeats** this process until a specified number of new tokens are generated.
- Attributes of softmax
    - The softmax function is **monotonic**, meaning it preserves the order of its inputs.
    - Therefore, applying torch.argmax directly to the logits tensor would yield the same result as applying it to the softmax output, as the position of the highest value remains unchanged.
- Greedy decoding
    - Greedy decoding refers to the strategy of always selecting the most likely token at each step.
    - While this approach can be efficient, it can also lead to repetitive or predictable text, as the model always chooses the most obvious continuation.
- Training
    - The model generates incoherent text because it has not yet learned the relationships between words and their contexts. Training is crucial for the model to develop the ability to generate meaningful and coherent text.

During text generation, a GPT model converts its output tensors into human-readable text by predicting tokens step by step and decoding them back into words or subwords. At each step, the model outputs **logits**, which are unnormalized scores for every token in the vocabulary. These logits are passed through the **softmax function**, which transforms them into a probability distribution where each value represents the likelihood of a token being the next one in the sequence. A simple generation loop, such as the `generate_text_simple` function, repeatedly uses the current context to predict the next token, appends this token to the input sequence, and continues until the desired number of tokens has been generated. Because the softmax function is **monotonic**, it preserves the ordering of logits, meaning that selecting the token with the highest probability using softmax is equivalent to directly applying `argmax` to the logits. When the model always selects the most probable token at

each step, this strategy is called **greedy decoding**. While greedy decoding is computationally efficient, it often produces repetitive or overly predictable text because it never explores less likely alternatives. If a GPT model has not been properly trained, its generated text will appear incoherent, as training is essential for learning meaningful relationships between words, context, and structure in language.

## 5.1 Evaluating generative text models

- Generate_text_simple function
    - The generate_text_simple function generates text by taking a starting context, converting it to token IDs, feeding it into the GPT model, and then converting the model's output logits back into token IDs, which are then decoded into text.
- Text generation loss
    - Text generation loss is a numerical measure used to evaluate the quality of generated text.
    - It quantifies the difference between the model's predicted output (token probabilities) and the actual desired output (target tokens).
    - A lower loss indicates better text generation quality.
    - What is a model’s predicted output, what are target tokens, what are token probabilities?
- Backpropagation
    - Backpropagation is a technique used to update the model's weights during training.
    - It uses the text generation loss to adjust the weights so that the model produces outputs closer to the target tokens, thereby minimizing the loss and improving the quality of generated text.
    - What is text generation loss?
- Cross entropy loss
    - Cross entropy loss is a measure of the difference between two probability distributions, typically the true distribution of labels (tokens) and the predicted distribution from the model.
    - It is used to evaluate the performance of LLMs by quantifying how well the model's predicted probability distribution matches the actual distribution of tokens in the dataset.
- Perplexity
    - Perplexity is a measure derived from cross entropy loss that provides a more interpretable way to understand the uncertainty of a model in predicting the next token.
    - It represents the effective vocabulary size about which the model is uncertain at each step. A lower perplexity indicates less uncertainty and better model performance.

In evaluating generative text models (like GPT), a function such as `generate_text_simple` generates text by taking an initial prompt (context), converting it into **token IDs**, feeding those IDs into the model, and obtaining **logits** for the next token;

those logits are turned into a probability distribution over the vocabulary, a next token is selected, appended to the context, and the process repeats until enough tokens are generated, after which the final token IDs are decoded back into text.

Model quality is commonly evaluated using **text generation loss**, which measures how different the model's predicted token distribution is from the **target tokens** (the correct next tokens from the dataset).

Training uses **backpropagation** to compute gradients of this loss with respect to the model's parameters and update weights so future predictions better match targets.

The standard loss for next-token prediction is **cross-entropy**, which penalizes the model when it assigns low probability to the correct next token.

**Perplexity** is a more interpretable transform of cross-entropy that reflects how "uncertain" the model is on average when predicting the next token; lower perplexity means the model is more confident and typically performs better.

### Super simple training picture

Suppose the training text is:

"**I like cats**"

- Input (context): "I like"
- Target (correct next token): "cats"
- Model predicts probabilities for the next token (cats/dogs/pizza/...)
- **Loss** is low if it gives high probability to "cats", high if it doesn't.

That's what "right vs wrong" means: *did it predict the correct next token(s) from the dataset*, not "did it reproduce the prompt".

## What are gradients? (simple)

A **gradient** tells you:

> "If I change this weight a tiny bit, will the loss go up or down, and by how much?"

So gradients are like **direction + strength** for weight updates:

- **Positive gradient**: increasing the weight increases loss → move weight down
- **Negative gradient**: increasing the weight decreases loss → move weight up
- **Big gradient**: weight matters a lot
- **Small gradient**: weight barely affects loss

Backpropagation is just the method to compute these gradients efficiently for all weights.

## Cross-entropy (simple)

Cross-entropy loss basically says:

> "Punish the model when it gives low probability to the correct next token."

If the correct token is "cats":

- Model says P(cats)=0.90 → small penalty
- Model says P(cats)=0.01 → huge penalty

## Perplexity (simple)

Perplexity is a human-friendly version of cross-entropy:

> "On average, how many tokens is the model 'confused between' each step?"

- Perplexity **2** ≈ like choosing between ~2 plausible tokens
- Perplexity **50** ≈ like being unsure among ~50 tokens

  Lower perplexity = less uncertainty = better next-token prediction.

5.2 Training an LLM

- Train_model simple function
  - The train_model_simple function implements a basic training loop for the LLM. It iterates over epochs, processes batches, calculates loss, updates weights, and evaluates the model's performance using validation data.
  - What are epochs, processes batches? What are validation data
- Evaluate_model function
  - The evaluate_model function calculates the loss on both the training and validation sets to assess the model's performance. It ensures the model is in evaluation mode, disabling gradient tracking and dropout for accurate evaluation.
  - What are validation sets, evaluation mode?
- Generate_and_print_sample
  - The generate_and_print_sample function generates text samples from the model to visually assess its progress during training.
  - It takes a text snippet as input, converts it to token IDs, and uses the generate_text_simple function to generate new text.
- AdamW
  - AdamW is a variant of the Adam optimizer that improves weight decay, a technique that penalizes larger weights to prevent overfitting. This makes AdamW more effective for regularizing and generalizing LLMs.
  - What is the adam optimizer

- Curves
    - The curves show that the model initially learns well, but the training loss continues to decrease while the validation loss stagnates. This indicates overfitting, where the model memorizes the training data instead of generalizing to new data
    - Show me curves so that i know what a different types of training loss and stuff means. Like what is a good curve, what is a bad curve

In training an LLM, `train_model_simple` runs the core training loop: it repeats over multiple **epochs** (each epoch is one full pass through the entire training dataset), and within each epoch it processes the data in **batches** (small chunks of examples processed together for efficiency and memory limits).

For each batch, the model does a forward pass to produce predictions, computes the **loss**, then uses backpropagation to update weights, and periodically checks performance on **validation data**—a held-out portion of data the model does not train on, used to measure how well it generalizes to unseen text.

The `evaluate_model` function measures loss on both the training set and the **validation set** (the dataset split reserved for evaluation) while placing the model in **evaluation mode** (`model.eval()`), which disables training-time randomness like **dropout** and typically uses `no_grad()` to stop gradient tracking so evaluation is faster and reflects true inference behavior.

To qualitatively monitor progress, `generate_and_print_sample` generates a short text continuation from a prompt by converting the prompt into token IDs and calling the text generation function, letting you “see” whether outputs improve over time.

Training commonly uses **AdamW**, which is based on the **Adam optimizer** (an adaptive gradient method that uses running averages of gradients to make stable, efficient updates) but applies **weight decay** more correctly to discourage overly large weights and reduce overfitting.

Finally, loss **curves** help diagnose learning: a “good” curve shows both training and validation loss decreasing and staying close, while a “bad” curve often shows training loss decreasing but validation loss flattening or rising—an overfitting pattern where the model memorizes training data instead of generalizing; other bad patterns include both losses staying high (underfitting) or losses oscillating wildly (unstable training).

---

EASY MODE EXPLANATION — CHAPTER 5.2 TRAINING AN LLM

1. train_model_simple (training loop)

- This is the model’s “practice routine.”

- It repeats: show examples → let the model predict → measure how wrong it was (loss) → update the model's weights → repeat.

2. Epoch

- An epoch = the model goes through the entire training dataset once.

- Example: if you have 10,000 training samples, 1 epoch means it has seen all 10,000 once.

3. Batch (processing batches)

- A batch = a small chunk of training examples processed together (e.g., 32 or 128 examples).

- Processing batches means looping through the dataset in chunks instead of one example at a time.

- Why batches are used: faster training and better use of GPU/memory.

4. Loss

- Loss = a number that measures "how wrong the model was."

- Smaller loss = better predictions; bigger loss = worse predictions.

- In text generation, the model predicts the next token, and loss checks if the model gave high probability to the correct next token.

5. Updating weights (learning)

- Weights are the model's internal "knobs."

- Updating weights means adjusting these knobs so the model makes better predictions next time.

6. Validation data / validation set

- Validation data = a held-out dataset the model does NOT train on.

- Training data is like practice questions; validation data is like a quiz the model didn't memorize.

- If training performance improves but validation does not, the model is likely memorizing (overfitting).

7. evaluate_model

- This function “tests” the model by computing loss on:
  (1) the training set, and
  (2) the validation set.

- It helps you see if the model is learning general patterns or just memorizing.

8. Evaluation mode (model.eval())

- Evaluation mode tells the model: “Stop training behavior.”

- Most important effect: dropout turns OFF so evaluation is stable and fair.

- If you forget eval mode, validation loss might be noisy or inaccurate.

9. Gradient tracking (torch.no_grad())

- During training, the model tracks operations so it can compute gradients and learn.

- During evaluation, we don’t update weights, so we disable gradient tracking using no_grad() to:

    - save memory

    - run faster

10. Dropout

- Dropout randomly “turns off” some connections during training.

- This prevents the model from relying too heavily on a few shortcuts.

- Purpose: reduce overfitting and improve generalization.

- Dropout is typically OFF during evaluation.

11. generate_and_print_sample

- This generates a short text sample from a prompt so you can visually check progress.

- Early training: output may look random.

- Later training: output becomes more coherent.

- It’s a qualitative check (human judgment), not a formal metric.

12. Adam optimizer (simple meaning)

- Adam is a method for updating weights efficiently.
- Compared to basic gradient descent, Adam:
    - adapts the step size automatically
    - uses momentum-like averaging for more stable learning
- Result: usually faster and more stable training.

13. AdamW

- AdamW = Adam + improved weight decay.
- Weight decay penalizes overly large weights, which helps reduce overfitting.
- AdamW is commonly used for transformers/LLMs because it often generalizes better.

14. Loss curves (how to interpret good vs bad)

A) Good learning curve

- Training loss decreases
- Validation loss decreases
- The gap between them stays small
- Meaning: model is learning and generalizing.

B) Overfitting curve

- Training loss keeps decreasing
- Validation loss stops improving or increases
- Meaning: model memorizes training data and fails to generalize.

C) Underfitting curve

- Training loss remains high
- Validation loss remains high
- Meaning: model is too weak, not trained enough, or training setup is poor.

D) Unstable training curve

- Loss jumps up and down a lot (noisy/oscillating)
- Meaning: often caused by too high learning rate, unstable optimization, or exploding gradients.

---

**Definitions**

- **Text generation loss**
  - **A measure used alongside cross entropy loss to evaluate the performance of models in tasks like language modeling**
  - **Provides a more interpretable way of understanding the uncertainty of a model in predicting the next token in a sequence**
- **Cross entropy loss**
  - **A common measure in machine learning that quantifies the difference between 2 probability distributions, typically the true distribution of labels and the predicted distribution from a model**
- **Perplexity**
  - **A numerical measure used to assess the quality of text generated during training, indicating how well the model's predictions align with the desired target text**
- **Backpropagation**
  - **A standard technique for training deep neural networks that involve updating the model's weights to minimize the difference between the model's predicted output and the actual desired output**

5.3 Decoding strategies to control randomness

- Temperature scaling
  - Temperature scaling adjusts the probability distribution of the next token by dividing the logits by a temperature value.
  - Higher temperatures create a more uniform distribution, leading to more diverse outputs, while lower temperatures sharpen the distribution, favoring the most likely tokens
  - What is a temperature value,
- Top-k sampling
  - Top-k sampling selects only the top-k most likely tokens for the next token prediction. This helps to reduce the generation of nonsensical or grammatically incorrect text by focusing on the most probable options.
- Greedy encoding
  - Greedy decoding always selects the token with the highest probability, while probabilistic sampling chooses tokens based on their probability distribution. Probabilistic sampling introduces randomness and can lead to more diverse outputs
- Generate function

    - The generate function allows for the specification of temperature and top-k values.
    - If a temperature is provided, the logits are scaled accordingly.
    - If a top-k value is given, the logits are masked to exclude tokens outside the top-k most likely options.

Decoding strategies control how “random” an LLM’s text generation is when choosing the next token. The model produces **logits** (raw scores) for all possible next tokens, and decoding decides how to turn those into a single choice.

**Temperature scaling** changes how sharp or flat the probability distribution is by dividing logits by a temperature value: a **low temperature** makes the model more confident and repetitive (more likely to pick the top token), while a **high temperature** spreads probability more evenly across many tokens, producing more variety but sometimes more mistakes.

**Top-k sampling** limits the model’s choices to only the **k most likely tokens**, which helps reduce weird or nonsensical outputs by preventing very low-probability tokens from being chosen.

**Greedy decoding** is the simplest strategy: it always picks the single most likely token every step, making output stable but often repetitive; **probabilistic sampling** instead chooses randomly according to token probabilities, which increases creativity and diversity.

A typical `generate` function combines these options: it can apply temperature scaling to logits, then optionally apply top-k masking (set logits outside top-k to negative infinity so they can’t be selected), and finally sample (or pick the max) to produce the next token.

5.4 Loading and saving model weights in PyTorch

- Saving the model weights
    - Saving the model weights allows you to reuse the trained model without retraining it from scratch, saving significant computational time and resources. This is especially important for large language models, which can take a long time to train.
- State_dict, torch.save
    - The recommended way is to save the model's state_dict, which is a dictionary mapping each layer to its parameters, using the torch.save function.
    - This allows you to easily load the weights into a new model instance later.
- model.eval()
    - The model.eval() method switches the model to evaluation mode, disabling dropout layers.
    - This is important for inference, as we don't want to randomly drop out information during prediction.
- Saving optimizer states
    - Saving the optimizer state allows you to resume training from where you left off. This is important for adaptive optimizers like AdamW, which store historical data to adjust learning rates dynamically.

    - Without the optimizer state, the model may learn suboptimally or fail to converge properly.

**Definitions**

- **Adam W**
    - **A variant of the adam optimizer that improves the weight decay approach**
    - **Aims to minimize model complexity and prevent overfitting by penalizing larger weights**
- **Training loop**
    - **The process of iterating over the training data multiple times, calculating the loss for each batch, and updating the model weights to minimize loss**
- **Over fitting**
    - **A phenomenon that occurs when a model learns the training data too well and performs poorly on unseen data**
- **Validation loss**
    - **Loss calculated on a separate dataset that is not used for training, which provides an estimate of the model's performance on unseen data.**

In PyTorch, saving and loading model weights lets you reuse a trained LLM without retraining, which saves huge time and compute.

The standard practice is to save the model's **state_dict**, a dictionary that stores each layer's parameters (weights and biases), using `torch.save`, and later load it into a new model instance with `load_state_dict`.

When you use a trained model for prediction (inference), you typically switch it to **evaluation mode** with `model.eval()`, which turns off training behaviors like **dropout** so the output is stable and not randomly altered.

If you want to resume training exactly from where you stopped, you should also save the **optimizer state**, because optimizers like **AdamW** keep internal running statistics (like momentum/variance estimates) that affect how updates happen; without that saved optimizer state, continuing training can behave differently and may converge more slowly or less reliably.

5.5 Loading pretrained weights from OpenAI

- Using pretrained weights
    - Using pretrained weights from OpenAI eliminates the need for extensive training on a large corpus, saving significant time and computational resources.
- Settings dictionary, params dictionary
    - The settings dictionary contains the LLM architecture settings, while the params dictionary holds the actual weight tensors for the model's layers.
- load_weights_into_gpt

        - The load_weights_into_gpt function carefully matches the weights from OpenAI's implementation with the corresponding layers in the custom GPTModel instance, ensuring consistency and functionality.
- Model_configs dictionary
    - The model_configs dictionary provides the specific architectural settings for different GPT-2 model sizes, allowing for the selection and loading of weights for the desired model.

The pretrained GPT-2 models from OpenAI were trained with a different context_length and used bias vectors in the attention module, requiring these settings to be updated for compatibility

Loading pretrained GPT-2 weights from OpenAI lets you start from a model that already learned general language patterns, so you avoid training from scratch on a massive corpus and save huge compute and time.

In practice, you keep two kinds of information: a **settings dictionary** that describes the model architecture (e.g., number of layers, embedding size, number of attention heads, context length), and a **params dictionary** that contains the actual learned weight tensors for each layer.

A helper like `load_weights_into_gpt` works like a careful “translator”: it maps each weight from OpenAI’s parameter naming/structure to the matching layer in your custom `GPTModel`, ensuring shapes and layer correspondences line up so the model behaves correctly.

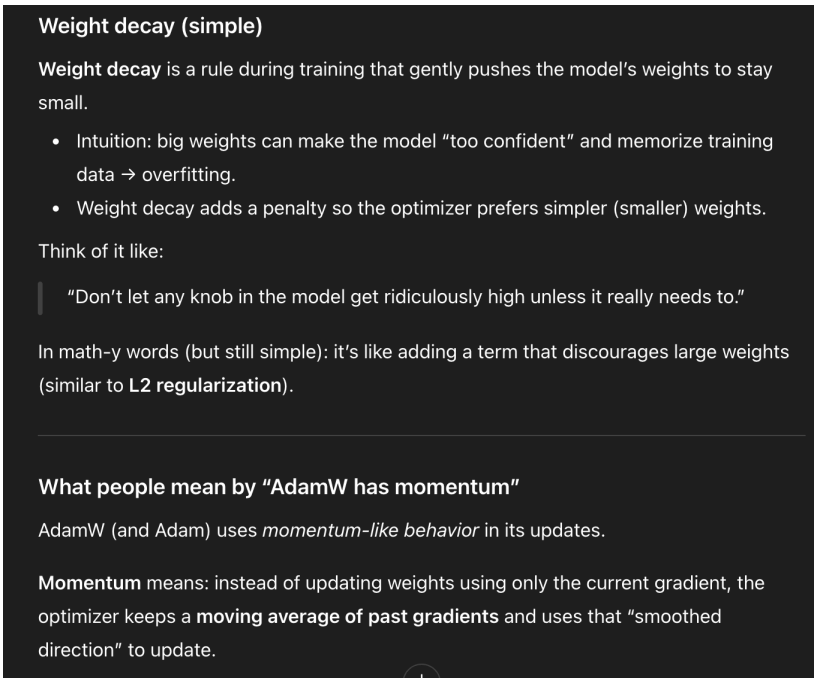
A **model_configs dictionary** stores the architecture settings for different GPT-2 sizes (small/medium/large/xl), making it easy to pick the configuration that matches the pretrained checkpoint you want to load.

One common compatibility issue is that OpenAI’s pretrained GPT-2 checkpoints may assume a different **context_length** and may include **bias vectors** in attention layers, so your custom model must enable the same bias settings and adjust context length (or handle resizing) to load the weights successfully.

**Definitions**

- **State dict**
    - **A dictionary containing the parameters of each layer in a pytorch model**
- **Model weights**
    - **The parameters that are learned during the training process and are used to make predictions**
- **Optimizer state**
    - **A dictionary containing the internal states of the optimizer, such as the learning rate and momentum**
- **Evaluation mode**

- **A mode in which the model is used for inference, and dropout layers are disabled.**



**Weight decay (simple)**

**Weight decay** is a rule during training that gently pushes the model's weights to stay small.

- Intuition: big weights can make the model "too confident" and memorize training data → overfitting.
- Weight decay adds a penalty so the optimizer prefers simpler (smaller) weights.

Think of it like:

> "Don't let any knob in the model get ridiculously high unless it really needs to."

In math-y words (but still simple): it's like adding a term that discourages large weights (similar to **L2 regularization**).

---

**What people mean by "AdamW has momentum"**

AdamW (and Adam) uses *momentum-like behavior* in its updates.

**Momentum** means: instead of updating weights using only the current gradient, the optimizer keeps a **moving average of past gradients** and uses that "smoothed direction" to update.

# 6.1 Different categories of fine-tuning

- 2 ways
  - The two most common ways to fine-tune language models are instruction fine-tuning and classification fine-tuning.
  - Instruction fine-tuning focuses on training the model to understand and execute tasks based on natural language prompts, while
  - classification fine-tuning trains the model to recognize specific class labels.
- Instruction
  - Instruction fine-tuning aims to improve a model's ability to understand and execute tasks based on natural language instructions.
  - For example, training a model to translate English sentences into German using specific instructions would be an instance of instruction fine-tuning.
- Classification
  - Classification fine-tuning involves training a model to recognize specific class labels.
  - For instance, training a model to identify whether an email is spam or not spam is an example of classification fine-tuning.
  - This approach is also used in tasks like image classification and sentiment analysis.
  - Limitations
    - A classification fine-tuned model is restricted to predicting classes it has encountered during training.
    - It can only classify data into the predefined categories it was trained on, limiting its ability to handle tasks outside its training scope.

- Instruction vs classification
    - Instruction fine-tuned models are more flexible and can handle a broader range of tasks based on user instructions.
    - Classification fine-tuned models are highly specialized and excel at categorizing data into predefined classes, but they lack the flexibility of instruction fine-tuned models.
- When to use instruction
    - Instruction fine-tuning is best suited for models that need to handle a variety of tasks based on complex user instructions, improving flexibility and interaction quality. It is ideal for applications requiring adaptability and the ability to respond to diverse user requests.

There are two common ways to fine-tune language models: instruction fine-tuning and classification fine-tuning. Instruction fine-tuning trains a model to understand and follow natural language instructions, allowing it to perform many different tasks such as translation, summarization, or question answering based on user prompts. In contrast, classification fine-tuning trains a model to assign inputs to specific predefined class labels, such as identifying whether an email is spam or determining sentiment. While classification fine-tuned models are highly accurate for the categories they are trained on, they are limited to those classes and cannot generalize beyond them. Instruction fine-tuned models are more flexible and adaptable, making them better suited for applications that require handling diverse tasks and complex user instructions.

6.2 Preparing the dataset

- Data set used: spam or non spam
    - The dataset used is a collection of text messages classified as either 'spam' or 'non-spam' (also known as 'ham').
    - This dataset is used to demonstrate the process of fine-tuning a large language model for **classification tasks.**
- Undersampling
    - The original dataset has a significant imbalance between 'spam' and 'nonspam' messages.
    - **Undersampling creates a balanced dataset**, which is beneficial for training a classification model as it prevents the model from being biased towards the majority class.
- String convert to integer
    - The 'string' class labels are converted into integer class labels (0 and 1) using a mapping dictionary.
    - This process is similar to converting text into token IDs, but instead of using the GPT vocabulary, it uses only two token IDs.
- Random_split function
    - The random_split function divides the dataset into three parts: **training, validation, and testing.**
    - The training set is used to train the model,

    - the validation set is used to adjust hyperparameters and prevent overfitting, and the
    - testing set is used to evaluate the model's performance on unseen data.
- Saving as CSV
    - Saving the dataset as CSV files allows for easy reuse of the data in future steps. This ensures that the prepared dataset can be readily accessed for further analysis and model training.

The dataset used consists of text messages labeled as either spam or non-spam (ham), and it is used to demonstrate how a large language model can be fine-tuned for a classification task. Because the original dataset is highly imbalanced, undersampling is applied to create a balanced number of spam and non-spam messages, which helps prevent the model from becoming biased toward the majority class. The text-based class labels are then converted into integer labels (0 and 1) so the model can process them numerically. Next, the dataset is split into training, validation, and testing sets using a random split, allowing the model to be trained, tuned, and evaluated on unseen data. Finally, the processed datasets are saved as CSV files so they can be easily reused in later stages of model training and analysis.

**Definitions**

- **Instruction fine tuning**
    - **Process of training a language model on a set of tasks using specific instructions to improve its ability to understand and execute tasks described in natural language prompts**
- **Classification fine-tuning**
    - **A method of training a model to recognize a specific set of class labels, such as spam and not spam**
- **Generalist model**
    - **A model that can perform well across a variety of tasks**
- **Specialized model**
    - **A model that is highly trained to perform a specific task**

6.3 Creating data loaders

- Truncating vs padding
    - The two options are truncating all messages to the length of the shortest message or padding all messages to the length of the longest message.
    - Truncation is computationally cheaper but can lead to information loss, while padding preserves the entire content of all messages.
- Padding token
    - The padding token is used to ensure that all text messages in a batch have the same length.
    - In the SpamDataset class, shorter messages are padded with the padding token ID (50256) to match the length of the longest message.
- Spamdata set class

        - The SpamDataset class handles several key tasks: loading data from CSV files, tokenizing text messages using the GPT-2 tokenizer, and padding or truncating sequences to a uniform length. It also provides methods to access individual data samples and the overall dataset length.
- Validation and test sets padding
    - The validation and test sets are padded to match the length of the longest training sequence. Any samples exceeding this length are truncated. This ensures consistency in input lengths across all datasets
    - What are validation and test sets again
- Single training batch
    - A single training batch consists of eight text messages represented as token IDs, each with 120 tokens. The corresponding class labels for each message are stored in a separate tensor. This structure allows for efficient processing of multiple training examples simultaneously
    - What is a tensor? Why is this structure efficient
- data loader class
    - The DataLoader class is used to create data loaders that efficiently load and process data in batches.
    - It takes a dataset as input and allows for customization of parameters like batch size, shuffling, and number of workers.
    - Separate data loaders are created for training, validation, and testing, each with specific configurations.
    - What is this even talking about

To train the model efficiently, the text data must be converted into batches of equal-length numerical sequences. This is done by either truncating long messages or padding shorter ones so that all messages have the same length. Padding uses a special padding token to fill shorter sequences, ensuring uniform input size. A custom SpamDataset class is used to load the data from CSV files, tokenize the text, and apply padding or truncation. The dataset is then split into training, validation, and test sets, where the validation set is used to tune the model and prevent overfitting, and the test set is used to evaluate final performance. Data loaders are created to group data into batches and feed them efficiently into the model during training and evaluation.

‘

6.4 Initializing a model with pretrained weights

- Initializing a pretrained model
    - Initializing a pretrained model prepares it for classification fine-tuning by loading the weights learned during pretraining on unlabeled data.
    - This allows the model to leverage its existing knowledge and accelerate the learning process for the specific classification task.
- Processes involving initialising a pretrained model
    - The process involves using the download_and_load_gpt2 function to retrieve the pretrained weights based on the chosen model size.

        - These weights are then loaded into the GPTModel using the load_weights_into_gpt function, ensuring the model is ready for fine tuning.
- Checking for initialization
    - By providing a prompt and generating text using the generate_text_simple function, we can assess whether the model produces coherent and meaningful output. If the generated text is sensible, it indicates that the pretrained weights have been loaded successfully.
- Classification checking
    - Prompting the model with a spam message allows us to evaluate its initial ability to classify spam. This provides a baseline understanding of the model's performance before fine-tuning and helps identify areas for improvement.

Initializing a model with pretrained weights prepares it for classification fine-tuning by loading knowledge learned from large amounts of unlabeled data during pretraining. This allows the model to start with a strong understanding of language and learn the classification task more quickly and effectively. The process involves downloading pretrained GPT-2 weights and loading them into the model so it is ready for fine-tuning. To check whether the initialization was successful, the model is given a prompt and asked to generate text; coherent output indicates that the pretrained weights were loaded correctly. Additionally, testing the model with a spam-related prompt provides a baseline view of its classification behavior before fine-tuning, helping assess how much improvement is needed.

6.5 Adding a classification head

- The original output layer of a pretrained LLM is designed for language generation, mapping hidden representations to a large vocabulary of tokens. For classification, we need a smaller output layer that maps to the specific classes we want to predict.
    - What are you talking about here
- Using separate output nodes for each class
    - Using a separate output node for each class allows for a more general approach to classification, as **it avoids the need to modify the loss function for binary tasks. This approach is easily adaptable to multi-class problems.**
        - **What is the need to modify the loss function for binary tasks**
- Lower levels vs upper levels
    - The lower layers of a pretrained LLM typically capture general language structures and semantics, while the upper layers learn task-specific features.
    - Fine-tuning only the last layers allows for efficient adaptation to new tasks without disrupting the learned general language knowledge
- Freezing layers
    - Freezing layers prevents their weights from being updated during training. This is done by setting the requires_grad attribute of their parameters to False.

        - Unfreezing layers allows their weights to be updated by setting requires_grad to True.
- Casual attention mask and classification
    - The causal attention mask restricts each token's attention to itself and preceding tokens.
    - As a result, the last token accumulates information from all previous tokens, making it the most comprehensive representation of the input sequence.
    - Since the last token in a sequence has access to information from all preceding tokens due to the causal attention mask, it provides the most relevant context for classification. Therefore, fine-tuning based on the last token's output allows for more accurate predictions.

A pretrained language model is originally designed for text generation, meaning its output layer predicts the next word from a very large vocabulary. For classification tasks such as spam detection, this output layer is replaced with a smaller classification head that predicts only the target classes. This is usually done by adding separate output nodes for each class, which makes the model easy to extend to multi-class problems. During fine-tuning, the lower layers of the model are often frozen because they contain general language knowledge, while only the upper layers and the new classification head are trained to adapt the model to the specific task. Because causal attention allows the last token in a sequence to attend to all previous tokens, its representation contains the most complete information about the input, making it well suited for classification.

6.6 Calculating the classification loss and accuracy

- Model’s output
    - The model's output for the last token is a 2-dimensional tensor representing the probability scores for each class (spam or not spam).
    - The class label is determined by finding the index of the highest probability score using the argmax function.
- Softmax and output
    - The softmax function converts the model's output into probabilities that sum to 1. It is optional because the largest output values directly correspond to the highest probability scores, so argmax can be applied directly to the output tensor
    - What is the argmax function
- Calc_accuracy_loader
    - The calc_accuracy_loader function iterates through a data loader, applies the argmax prediction to each input, and calculates the proportion of correct predictions.
    - It returns the classification accuracy as a percentage
- Classification accuracy -> cross-entropy loss
    - Classification accuracy is not a differentiable function, making it unsuitable for optimization.

        - Cross-entropy loss is a differentiable function that can be used to optimize the model's parameters to indirectly maximize classification accuracy.
- Cacl_loss_batch
    - The classification calc_loss_batch function focuses on optimizing only the last token's output, while the language modeling version optimizes the output of all tokens.
    - This difference reflects the focus on predicting the class label for the last token in classification.

In classification fine-tuning, the model produces a two-dimensional output for the last token, representing the scores for the spam and non-spam classes, and the predicted class is obtained using the argmax function. The `calc_accuracy_loader` function evaluates model performance by iterating over a data loader, applying argmax to each batch of outputs, and computing the proportion of correct predictions as classification accuracy.

However, because classification accuracy is not differentiable, it cannot be used for optimization during training. Instead, the `calc_loss_batch` function computes the cross-entropy loss, which is differentiable and suitable for gradient-based learning. Unlike language modeling, where loss is calculated across all tokens in a sequence, `calc_loss_batch` for classification focuses only on the output of the last token, since it contains the most complete contextual representation of the input due to causal attention.

6.7 Fine-tuning the model on supervised data

- The primary difference is that during fine-tuning, we calculate the classification accuracy instead of generating sample text to evaluate the model's performance
- The train_classifier_simple function tracks the number of training examples seen instead of tokens and calculates accuracy after each epoch. It also omits the step of printing a sample text. 3. The evaluate_model function calculates the loss for both the training and validation sets, providing insights into the model's performance on both seen and unseen data. 4. The loss curves in Figure 6.16 show a sharp decline in both training and validation loss during the initial epochs, indicating effective learning. The close proximity of the curves suggests that the model is generalizing well to unseen data and not overfitting. 5. The number of epochs depends on the dataset's complexity and the task's difficulty. Overfitting may necessitate reducing the number of epochs, while insufficient training might require increasing them.

The main difference during fine-tuning is that model performance is evaluated using classification accuracy rather than by generating sample text. The `train_classifier_simple` function tracks the number of training examples

instead of tokens, computes accuracy after each epoch, and removes the step of printing generated text.

The `evaluate_model` function calculates the loss on both the training and validation datasets, allowing comparison between seen and unseen data. As shown in the loss curves, both training and validation loss decrease rapidly in the early epochs, indicating that the model is learning effectively. The close alignment of these curves suggests good generalization and little overfitting. The appropriate number of training epochs depends on task difficulty and dataset complexity, as too many epochs may cause overfitting while too few may lead to undertraining.

6.8 Using the LLM as a spam classifier

- The process involves using a pre-trained LLM, fine-tuned for classification, to predict the class label (spam or not spam) for a given text message. This is achieved by converting the text into token IDs, feeding it to the model, and obtaining the predicted class label based on the model's output. 2. The classify_review function takes a text message as input, preprocesses it, converts it into token IDs, feeds it to the fine-tuned model, and predicts the class label (spam or not spam) based on the model's output. It then returns the corresponding class name.
- The classification accuracy is evaluated by calculating the fraction or percentage of correct predictions made by the model on a test dataset. This metric indicates how well the model is able to correctly classify spam and non-spam messages. 4. Saving the model allows for its reuse later without the need for retraining. This is useful for deploying the model for real-time spam detection or for further experimentation and analysis.
- While both processes involve converting text into token IDs and using a cross-entropy loss function, classification fine-tuning focuses on training the model to output a correct class label, whereas pretraining aims to predict the next token in a sequence. Classification fine-tuning also involves replacing the output layer of the LLM with a smaller classification layer.

In the classification stage, a pretrained and fine-tuned language model is used to predict whether a given text message is spam or not spam. The text is first converted into token IDs and then fed into the model, which outputs a predicted class label. The `classify_review` function handles this process by preprocessing the input text, converting it into tokens, passing it through the fine-tuned model, and returning the corresponding class name. Model performance is evaluated using classification accuracy, which measures the percentage of correct predictions on a test dataset. Saving the trained model allows it to be reused later without retraining, making it suitable for deployment or further analysis. Although both pretraining and classification fine-tuning involve tokenizing text and using cross-entropy loss, pretraining focuses on predicting the next token in a sequence, while classification fine-tuning trains the model to output a correct class label using a smaller classification head.

## `Download_and_load_gpt2`: loads pretrained GPT weights, reuse knowledge

Loads pretrained GPT-2 weights based on the selected model size so the model can reuse knowledge learned during pretraining.

## `Load_weights_into_gpt`: loads the pretrained weights

Loads the pretrained weights into the GPT model architecture, preparing it for fine-tuning.

## `Generate_text_simple`: check whether pretrained were loaded correctly

Generates sample text from the model to check whether pretrained weights were loaded correctly and the model is functioning properly.

## `SpamDataset` (class): loads and prepares data

A custom dataset class that:

- Loads data from CSV files
- Tokenizes text using the GPT-2 tokenizer
- Pads or truncates sequences to a fixed length
- Provides access to individual samples and dataset size

## `DataLoader`:data prep: batches, shuffling

Creates iterable data loaders that:

- Group data into batches
- Shuffle data (for training)
- Efficiently feed tensors into the model during training, validation, and testing

## `Calc_accuracy_loader`:evaluate model performance: classification accuracy

Evaluates model performance by:

- Iterating through a data loader
- Applying `argmax` to model outputs
- Computing the percentage of correct predictions (classification accuracy)

## `Calc_loss_batch`: cross entropy loss of last token

Computes the **cross-entropy loss** for classification by:

- Using only the output of the **last token**
- Comparing predicted class scores with true class labels

## `Evaluate_model`: loss for both training + eval data

Calculates and returns the loss for both:

- Training data
- Validation data

This helps assess learning progress and detect overfitting.

### `Train_classifier_simple:loop of computing loss and accuracy after each epoch`

Handles the full fine-tuning loop by:

- Training the classification head
- Tracking the number of training examples
- Computing loss and accuracy after each epoch
- Omitting text generation (unlike language-model training)

### `Classify_review:new text message and classifies it`

Takes a new text message and:

- Tokenizes it
- Feeds it into the fine-tuned model
- Uses `argmax` to predict the class
- Returns the class name (spam or not spam)

## 7.1 Introduction to instruction fine-tuning

Pretrained LLMs are mainly designed for text completion, meaning they can continue a sentence or generate text based on a given prompt. However, they often struggle to correctly follow explicit instructions such as grammar correction, rewriting text, or changing tone. Instruction fine-tuning addresses this limitation by training the model to understand and follow specific instructions. This is done by providing examples that pair instructions with desired responses. Preparing a high-quality instruction dataset is therefore a crucial step in instruction fine-tuning.

7.2 Preparing a dataset for supervised instruction fine-tuning

Instruction fine-tuning uses a dataset made up of instruction–response pairs, which teach the model how to respond appropriately to different tasks. The dataset is stored in a JSON file, where each entry contains an instruction, an optional input, and the expected output.

Two prompt styles are introduced: Alpaca and Phi-3. Alpaca uses a structured format with clear sections, while Phi-3 uses special user and assistant tokens. The `format_input` function converts dataset entries into the Alpaca-style prompt format. The dataset is then split into training, validation, and test sets so the model can be trained, tuned, and evaluated properly.

7.3 Organizing data into training batches

A custom collate function is used to prepare instruction data for training by padding all examples in a batch to the same length. Padding is done using the `<|endoftext|>` token so that batches are as compact as possible. Target token IDs are created by shifting the input tokens one position to the right, allowing the model to learn next-token prediction. Padding tokens in the target sequence are replaced with `-100` so the loss function ignores them during training. Keeping a single end-of-text token helps the model learn when a response should stop, which is important for generating complete answers.

7.4 Creating data loaders for an instruction dataset

The `custom_collate_fn` function is used together with the DataLoader to batch instruction data and move input and target tensors to the correct device, such as CPU or GPU. Performing device transfer inside the collate function improves efficiency by avoiding delays during training.

Batch instruction data and move input and target tensors to the correct device

The `partial` function is used to preset the device argument so the collate function always uses the correct hardware. Input sequences are truncated using the `allowed_max_length` parameter to ensure they fit within the LLM's context window. Separate DataLoaders are created for training, validation, and test sets, with configurable batch size and shuffling behavior.

Partial function: preset the device so collate function always uses the correct hardware

Input sequences are truncated using the allowed max_length parameter to ensure they fit in the context window.

7.5 Loading a pretrained LLM

Instruction fine-tuning benefits from using a larger pretrained model, as smaller models often lack the capacity to learn complex instruction-following behavior. Loading a pretrained LLM provides a strong foundation because the model already understands general language patterns.

In this chapter, `gpt2-medium` is used instead of `gpt2-small` to increase model capacity. Evaluating the pretrained model before fine-tuning provides a baseline for comparison. During evaluation, the input instruction is removed from the generated text so that only the model's response is analyzed.

7.6 Fine-tuning the LLM on instruction data

Fine-tuning trains the pretrained LLM on instruction–response data so it becomes better at following user instructions. The process involves loading the model, preparing the dataset, and training it using a loss function and optimizer that minimize prediction error. Hardware limitations such as memory constraints may require smaller batch sizes or shorter input sequences. Model progress is monitored using training and validation loss curves, where decreasing loss indicates improvement. Qualitative inspection of generated responses also helps assess how well the model is learning. Datasets such as Alpaca are especially useful because they contain diverse instructions and responses.

What is a loss function

7.7 Extracting and saving responses

After fine-tuning, the model's responses on a test set are extracted for evaluation. These responses can be manually reviewed or evaluated using benchmarks, human judgment, or automated methods. Conversational performance is particularly important for chat-based applications, as it reflects how naturally and coherently the model responds. An automated evaluation approach similar to AlpacaEval can be used, where another LLM judges the responses. The generated responses are added to the test dataset and saved as a JSON file, making them easy to reuse for further analysis.

alpacaeval

7.8 Evaluating the fine-tuned LLM

Evaluation can be automated by using a larger LLM, such as GPT-4 or Llama 3, to score the responses produced by the fine-tuned model. Tools like Ollama allow these evaluation models to run locally. The `query_model` function sends evaluation prompts to the LLM, while the `generate_model_scores` function aggregates scores across the test dataset to produce an overall performance metric. Different evaluation models can be chosen depending on available computing resources. Model performance can be further improved by tuning hyperparameters, expanding the dataset, refining prompts, or using a larger pretrained model.

Score responses, ollama run locally

Query mode: evaluation prompts to llm,

Generate model scores: aggregate scores across the test dataset to produce an overall performance metric