GenAI: Creative Text Generation Project Report

Objective:

The goal of this project is to implement generative AI models, focusing on text generation with models like transformers. For practical experience in building a basic transformer model, and fine-tuning the pre-existing models like gpt-2 for creative writing tasks.

Project Structure:

The project will be divided into several stages that gradually build upon each other. Each stage will involve both theoretical and practical components, to understand both the underlying principles of generative AI and the hands-on experience of implementing them.

Project Breakdown:

1. Dataset Preparation (Week 1):

- **Objective:** Prepare a creative dataset for training the transformer model.
- **Dataset Choice:** Select a creative category like poetry, short stories, recipes, etc. A dataset can be created by scraping publicly available resources, such as *poets.org* for poems or using other literary datasets.
- **Tasks:**
    - Scraping text data (e.g., poems, stories) from the chosen website.
    - Preprocessing the data (removing unnecessary punctuation, normalizing the text, etc.).
    - Tokenizing the dataset to convert text into numerical representations (word-level or subword-level tokenization).

2. Theory of Transformers (Week 1 - 2):

- **Objective:** Introduction to the foundational concepts of transformers.
- **Resources:**
  The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time.
  All you need to know about 'Attention' and 'Transformers' — In-depth Understanding — Part 1
  All you need to know about 'Attention' and 'Transformers' — In-depth Understanding — Part 2
  Transformer Neural Networks, ChatGPT's foundation, Clearly Explained!!! ***
  3B1B's series on LLMs

3. Building a Basic Transformer (Week 2):

- **Objective:** Implement a very basic version of a transformer model from scratch.
- **Tasks:**
    - **Breakdown of Components:**
        - **Self-Attention:** Code for the self-attention mechanism, explaining how the model weighs different parts of the input.
        - **Positional Encoding:** Implement the positional encoding to account for the order of words in the sequence.
        - **Encoder-Decoder Layers:** Implement the encoder and decoder layers, showing how information flows between them in the transformer architecture.
    - **Expected Output:**
        - A model that can generate the next word (or token) given a sequence of previous words, for tasks like text completion (with detailed comments on what each code block is doing)
        - **Voiceover:** Create a 10-minute voiceover explaining your code work, including concepts like self-attention, positional encoding, and multi-head attention, in groups of 3 or 4.
    - **Evaluation Metric:** Basic loss (cross-entropy loss) during training. The model's output can be evaluated qualitatively by generating a few text samples and seeing if they make sense contextually.
- **Resources:**
    Build your own Transformer from scratch using Pytorch
    Transformer Model from Scratch using TensorFlow - GeeksforGeeks

4. Fine-Tuning Pre-Trained GPT-2 Model (Week 3):

- **Objective:** Explore state-of-art pre-trained models, pipeline functions and fine-tune a pre-trained GPT-2 model on the creative dataset prepared earlier.
- **Tasks:**
    - **Pre-trained Model:** Use an existing GPT-2 model as a base.
    - **Fine-Tuning:** Train the model on the dataset to adapt it to the specific genre or type of creative writing chosen by the mentees.
    - **Model Evaluation:** Use metrics like BLEU scores or human evaluation to assess the quality of the generated text. The output should be coherent and contextually relevant to the dataset (e.g., a poem, story, or recipe).
    - **Deployment:** Upload the fine-tuned model to a repository like Hugging Face, experimenting with different prompts and generating text.
- **Resources:**
    First 4 chapters from the HuggingFace NLP course
    Fine tuning a gpt2 model for poetry generation

Deliverables:

1. **Cleaned Dataset:** A prepared and tokenized dataset for training.
2. **Transformer Model Code with a voiceover explaining it:** Code for the basic transformer model including the self-attention mechanism, positional encoding, and encoder-decoder layers.
3. **Fine-Tuned GPT-2 Model:** The fine-tuned model, uploaded to Hugging Face or a similar platform.

Expected Outcomes:

- **Learning Outcomes:** Gained hands-on experience with modern generative models like transformers and GPT-2. Understood the underlying mechanisms of attention, encoding, and decoding in transformers.
- **Creativity and Innovation:** Explored the creative potential of AI in generating text, experimenting with prompts and seeing how AI can assist in creative writing tasks.
- **Saving and uploading the trained model:** Uploading the fine-tuned models to platforms like Hugging Face encourages to help the broader AI community.

220215_Task 1.ipynb - Colab

220215_Task 2.ipynb - Colab

Week 1: Dataset Creation and Cleansing

Task Description:

For Week 1, your task is to first create a **raw database** of any creative type. This can include:

- Poems
- Short stories (beware of their length)
- Recipes
- Jokes (but prepare yourself for potentially embarrassing outputs from your GPT-2 model 😅), etc.

Save the dataset in a JSON file. Here's an example structure for your raw dataset:

```
[
  {
```

"poem": "I am taken with the hot animal\nof my skin, grateful to swing my limbs\n\nand have them move as I intend, though\nmy knee, though my shoulder, though something\nis torn or tearing. Today, a dozen squid, dead\n\non the harbor beach: one mostly buried,\none with skin empty as a shell and hollow\n\nfeeling, and, though the tentacles look soft,\nI do not touch them. I imagine they\nwere startled to find themselves in the sun.\n\nI imagine the tide simply went out\nwithout them. I imagine they cannot\n\nfeel the black flies charting the raised hills\nof their eyes. I write my name in the sand:\nDonika Kelly\n. I watch eighteen seagulls\n\nskim the sandbar and lift low in the sky.\nI pick up a pebble that looks like a green egg.\n\nTo the ditch lily I say\nI am in love\n.\nTo the Jeep parked haphazardly on the narrow\nstreet\nI am in love\n. To the roses, white\n\npetals rimmed brown, to the yellow lined\npavement, to the house trimmed in gold\nI am\n\nin love\n. I shout with the rough calculus\nof walking. Just let me find my way back,\nlet me move like a tide come in."

},

{

"poem": "It’s neither red\nnor sweet.\nIt doesn’t melt\nor turn over,\nbreak or harden,\nso it can’t feel\npain,\nyearning,\nregret.\n\nIt doesn’t have\na tip to spin on,\nit isn’t even\nshapely—\njust a thick clutch\nof muscle,\nlopsided,\nmute. Still,\nI feel it inside\nits cage sounding\na dull tattoo:\nI want, I want—\n\nbut I can’t open it:\nthere’s no key.\nI can’t wear it\non my sleeve,\nor tell you from\nthe bottom of it\nhow I feel. Here,\nit’s all yours, now—\nbut you’ll have\nto take me,\ntoo."

}.....

]

You can also include additional metadata for each entry, such as:

- **Tags**
- **Author**
- **Type** (e.g., sonnet, prose, etc.)

Dataset Size:

Make sure the dataset is appropriately sized:

- **Too small**: Risk of underfitting during training.
- **Too large**: Prolonged training time (though it may yield better results).

Dataset Cleansing:

Once the raw dataset is ready, the next step is to create a **cleansed.txt** file. This involves:

1. Removing unnecessary elements, such as special characters (`@~{[<`).
2. Eliminating empty or duplicate entries.
3. Tailoring the cleansing process to your chosen category (e.g., preserving line breaks and punctuation for poems).

Below is a sample Python code snippet for cleansing:

```
import json

# Load raw data from JSON
with open("poems_data.json", "r") as f:
    data = json.load(f)

# Cleansing process
# Example: Remove entries with empty poems or filter based on specific
criteria
data = [poem for poem in data if len(poem["poem"].split())]

# Print the number of entries after cleansing
print(len(data))

# Check the first few cleansed entries
print(data[:5])
```

Next Steps:

Once you've completed this part, you can proceed to:

1. **Data Modeling**: Structuring the data for training.
2. **Dataset Creation**: Preparing the dataset.

Good luck, and happy dataset creation!

I will release the next steps of tokenization and Dataset creation soon.

Tokenization:

I request you to visit videos 13 - 18 from this playlist: Hugging face course
From the above material, you get an idea of the tokenizers used…

Now we explore a very specific kind of tokenizer used in gpt2 models, byte-pair tokenizers.
Byte Pair Encoding (BPE) is a subword tokenization technique that splits text into smaller units (subwords or characters) based on their frequency in the training data. It's widely used in models like GPT-2, where handling unseen words and maintaining efficiency are essential.
Here is a colab notebook link to explore it further:
Tokenizers.ipynb

Also, please go through these to get an idea of transformers
Transformers, explained: Understand the model behind GPT, BERT, and T5
Video 5-8 of Hugging face course
Illustrated Guide to Transformers Neural Network: A step by step explanation

Next steps are Hugging face 'datasets' library, dataloading, and pipeline functions…
Still a long way to go for a fine tuned model :)

Data Loading

Since we are now done with tokenization and have our cleansed dataset ready, we will move to data loading, i.e. making the dataset fit to be fed into the model directly…

Here's a brief explanation of the roles of each component in `torch.utils.data`:

1. **`Dataset`**:
    - A base class for PyTorch datasets.
    - It provides a way to define and customize your dataset by implementing two key methods:
        - `__len__`: Returns the size of the dataset.
        - `__getitem__`: Fetches a data sample by index.
    - Example: Create a custom dataset for specific data processing needs.
2. **`random_split`**:
    - Splits a dataset into non-overlapping subsets of given lengths.
    - Useful for creating training, validation, and test splits.
    - Example: `train_set, val_set = random_split(dataset, [80, 20])`.
3. **`DataLoader`**:

    - Wraps a dataset to provide batch loading, shuffling, and parallel data loading.
    - Handles batching, shuffling, and multiprocessing (for faster data loading).
    - Example: `train_loader = DataLoader(train_set, batch_size=32, shuffle=True)`.
4. **`RandomSampler`**:
    - Samples elements randomly from a dataset without replacement.
    - Useful for shuffling when more control is needed than just using `shuffle=True` in `DataLoader`.
    - Example: `sampler = RandomSampler(dataset)`.
5. **`SequentialSampler`**:
    - Samples elements sequentially, i.e., in the order they appear in the dataset.
    - Useful for inference or when the order of data matters.
    - Example: `sampler = SequentialSampler(dataset)`.

Data Loading.ipynb please go through this colab file to understand how to load the data into a suitable form

Pipeline functions and model configuration

Pipeline functions offer an easy way to use Hugging Face Transformers for a variety of NLP tasks, such as sentiment analysis, text classification, and text generation, with minimal code. In this module, we'll explore common pipeline tasks, show how to customize them with specific models, and dive into GPT-2 model configurations to understand how to fine-tune and optimize them for specific needs. Let's get started!
Pipeline functions.ipynb

A Bit About the `datasets` Library

The `datasets` library from Hugging Face provides an efficient way to access, load, and process a variety of datasets for NLP tasks. With a few simple commands, you can:

- Load large datasets.
- Explore and customize data processing.
- Prepare data for model training or evaluation.

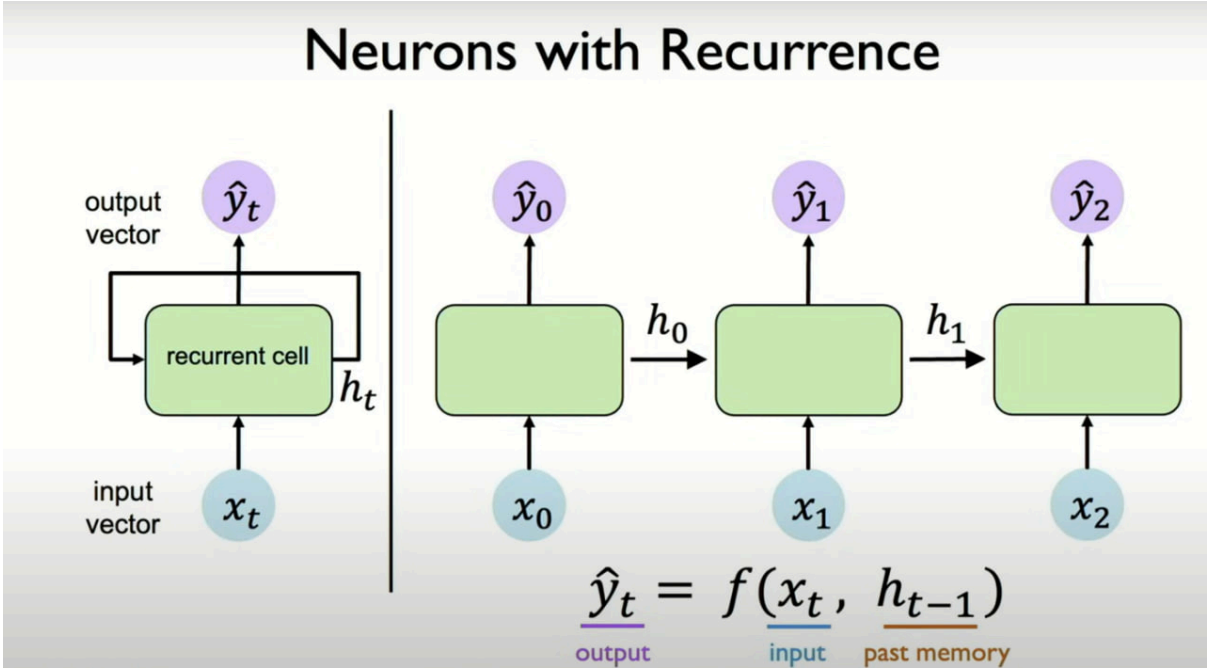
Refer to the following articles for an overview:

Introduction - Hugging Face NLP Course

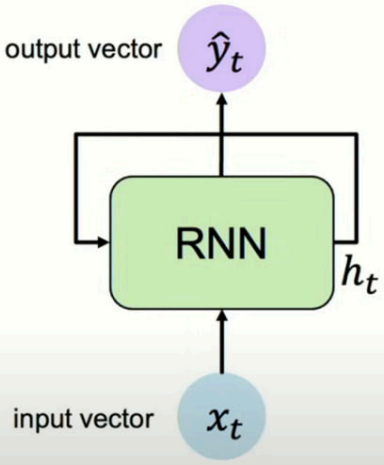
[What if my dataset isn't on the Hub? - Hugging Face NLP Course]
[Time to slice and dice - Hugging Face NLP Course]

1. **Dataset Preparation for Fine-Tuning:**
    - Prepare the dataset up to the data-loading stage.
    - Output relevant intermediate results and include detailed comments about the following:
        - The type of dataset used.
        - Cleansing techniques applied to the dataset.
    - Perform **Exploratory Data Analysis (EDA)** on the cleaned data. Include any insights or analyses that you consider relevant.
    - Print a few sample instances from your cleaned dataset to demonstrate the quality of the preprocessing.
2. **Word Embedding Visualization Using Co-occurrence Matrix:**
    - Construct a co-occurrence matrix for a dataset of sentences. (Suggested dataset size: approximately **5,000–10,000 sentences** for meaningful results.)
    - Apply **Principal Component Analysis (PCA)** to reduce the dimensionality of the matrix to 2D.
    - Visualize the 2D word embeddings for a selected set of words (e.g., **"king," "queen," "man," "woman," "apple," "orange"**).
    - Compare your results with existing 2D embeddings (e.g., from **pre-trained models like Word2Vec, GloVe, or FastText**) and analyze the differences.

RNN



Recurrent Neural Networks (RNNs)



Apply a **recurrence relation** at every time step to process a sequence:

$$h_t = f_W(x_t, h_{t-1})$$

cell state | function with weights W | input | old state

Note: the same function and set of parameters are used at every time step

RNNs have a **state**, h_t, that is updated **at each time step** as a sequence is processed

Output Vector

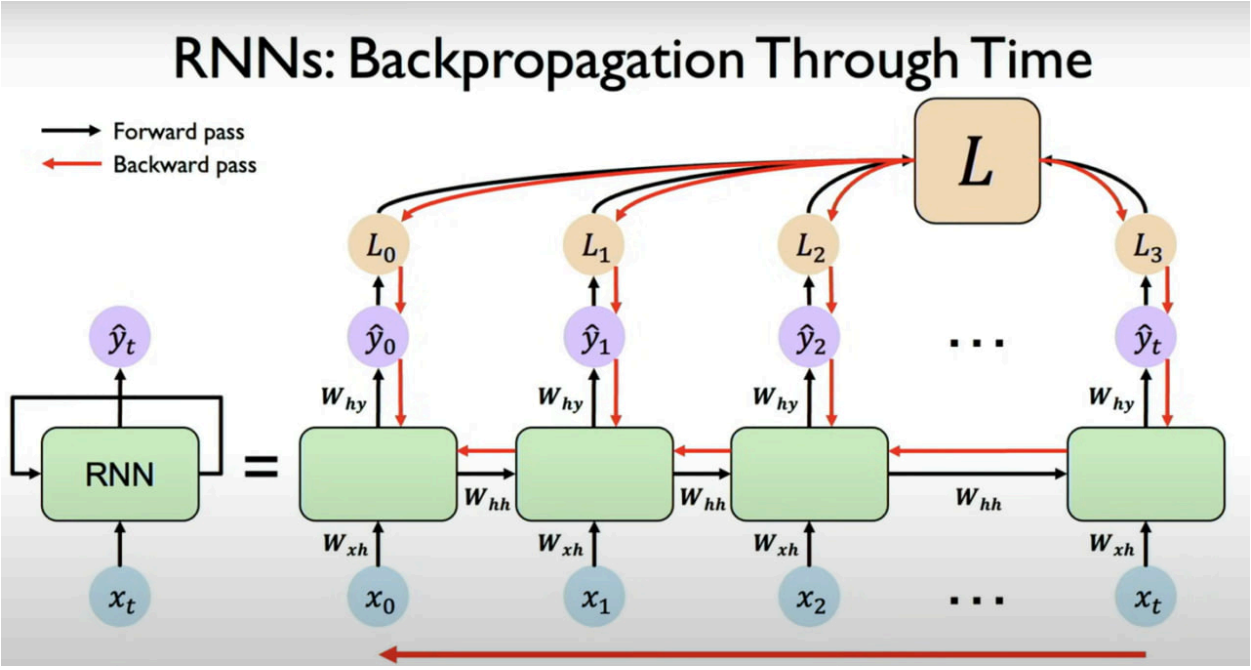
$$\hat{y}_t = \boldsymbol{W}_{\boldsymbol{hy}}^{\boldsymbol{T}} h_t$$

Update Hidden State

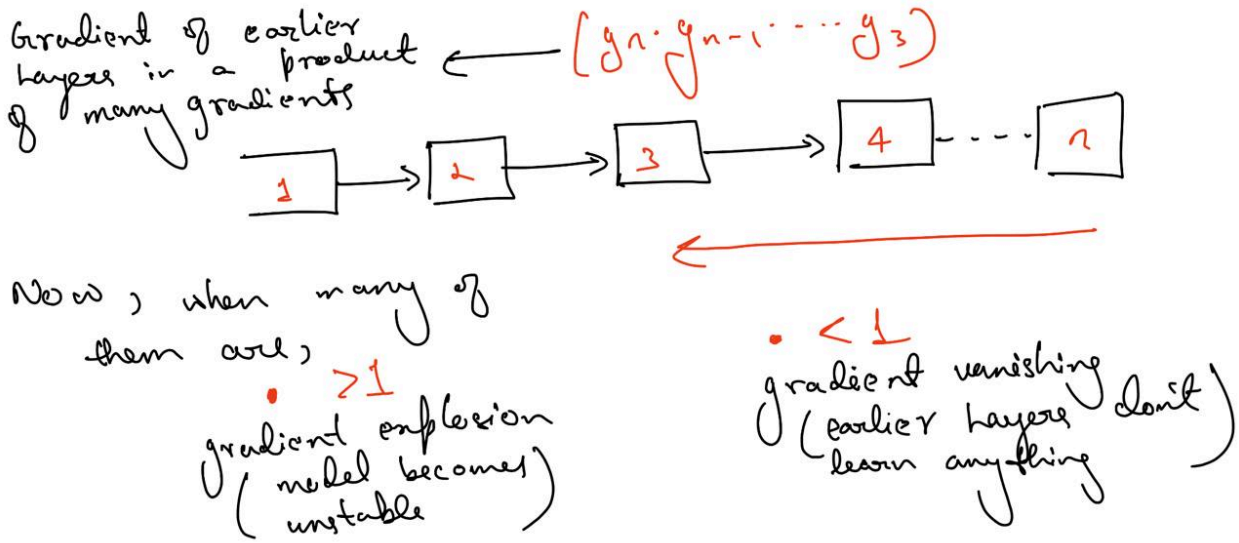
$$h_t = \tanh(\boldsymbol{W}_{\boldsymbol{hh}}^{\boldsymbol{T}} h_{t-1} + \boldsymbol{W}_{\boldsymbol{xh}}^{\boldsymbol{T}} x_t)$$

Input Vector

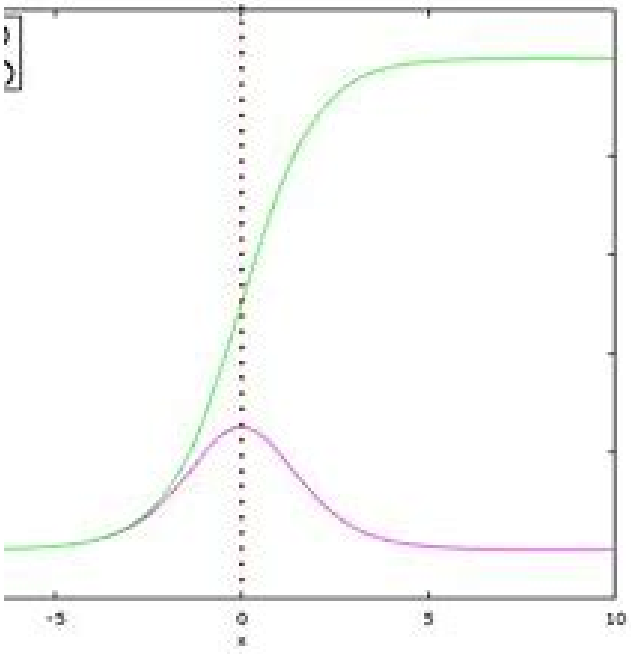
$$x_t$$



Problem of vanishing and exploding gradients:



Tackling of, Vanishing gradients: Use better activation functions like ReLU, instead of sigmoids/tanh, use gradient clipping, better parameters initialisation
Exploding gradients: Use gradient clipping, or L2 weight regularisation to prevent the weights from becoming too large



σ(x) and its derivate σ'(x)

Domain: (−∞, +∞)

Range: (0, +1)

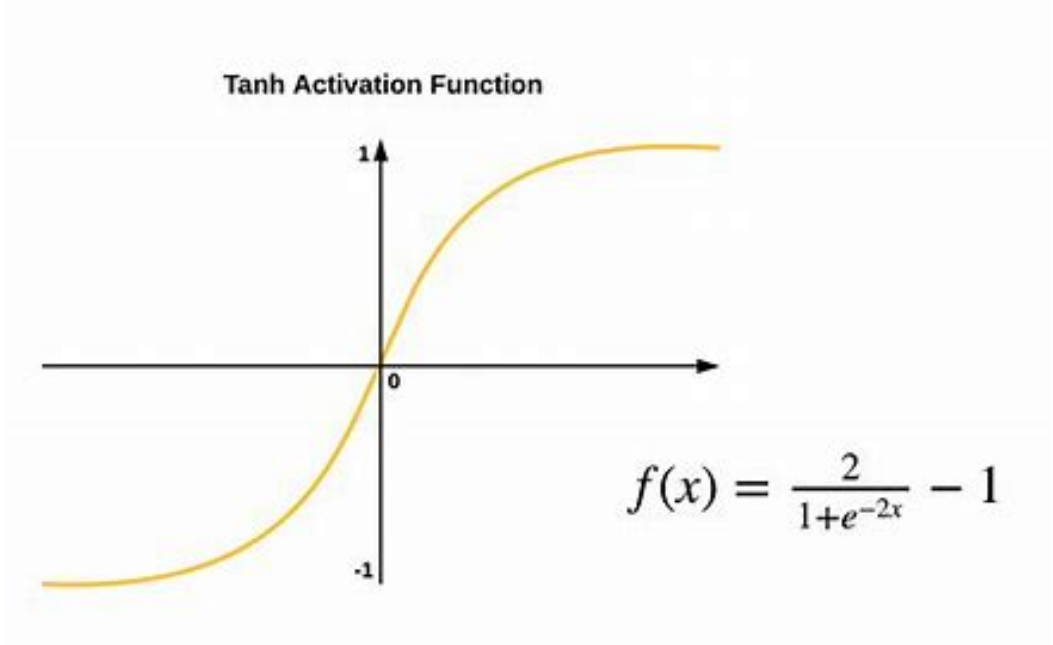
σ(0) = 0.5

Other properties

$$\sigma(x) = 1 - \sigma(-x)$$

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

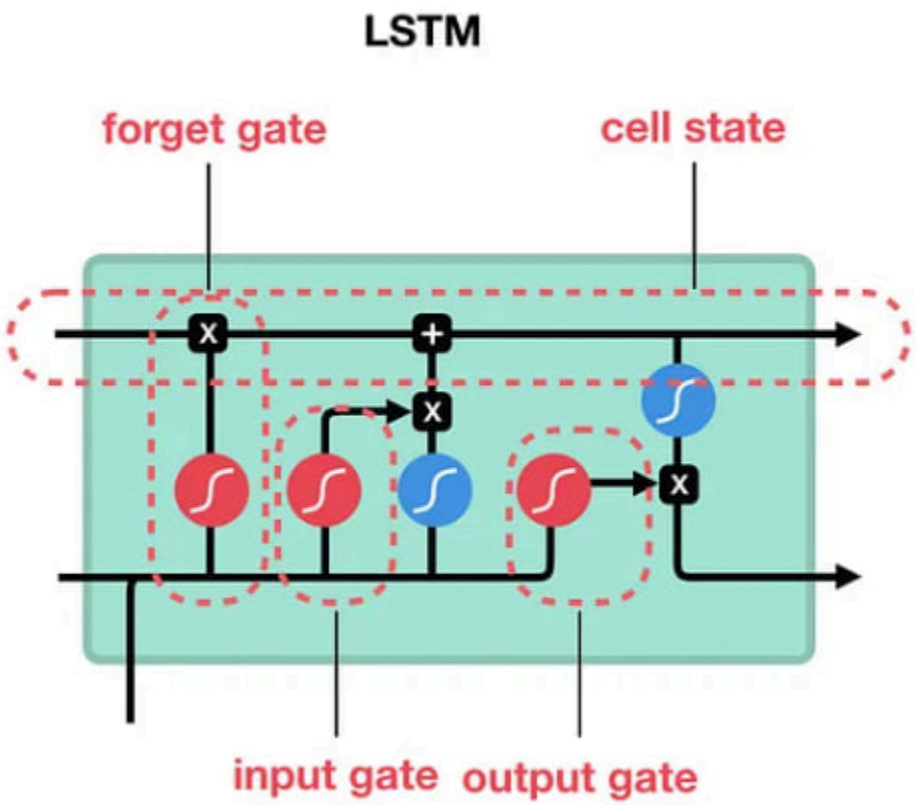


LSTMs and GRUs were introduced to tackle this problem of forgetting previous data.

LSTM and GRU

Long short-term memory, and gated recurrent units

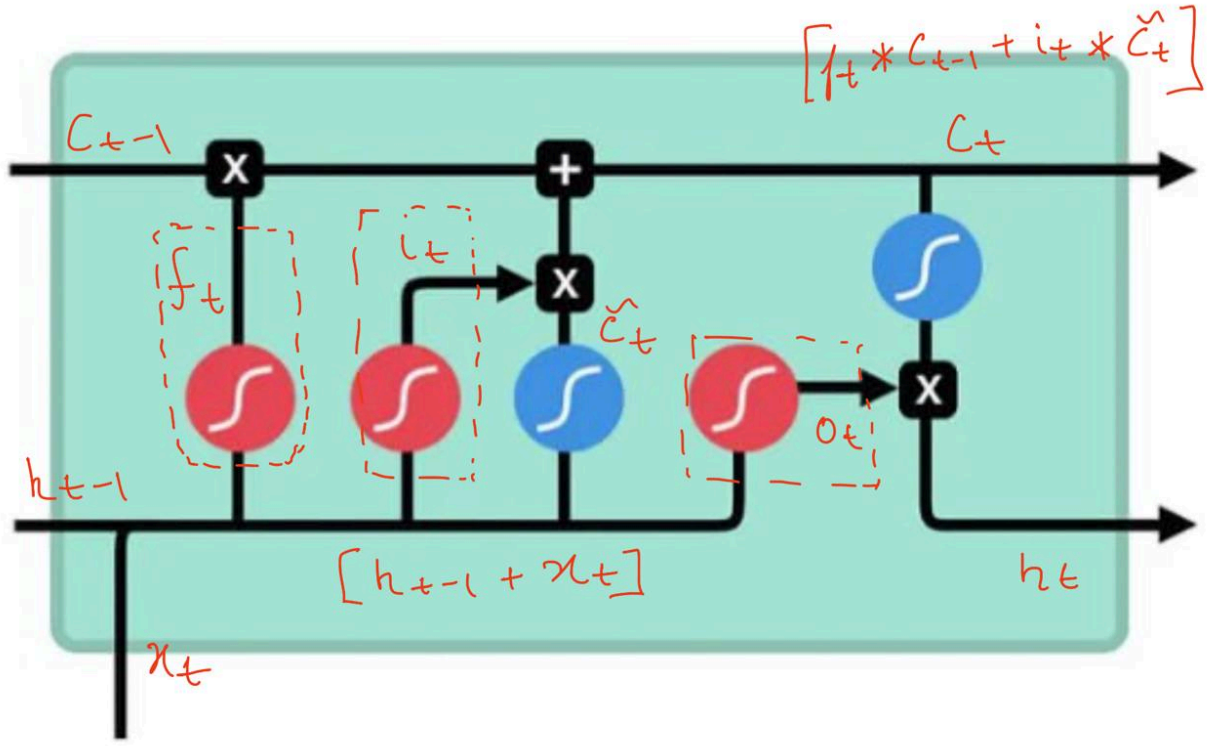
(some glossary for future reference)



(basic architect of LSTM)

Core Concept

The core concept of LSTM is the cell state, and its various gates. The cell state act as a transport highway that transfers relative information all the way down the sequence chain. You can think of it as the “memory” of the network. The cell state, in theory, can carry relevant information throughout the processing of the sequence. So even information from the earlier time steps can make its way to later time steps, reducing the effects of short-term memory. As the cell state goes on its journey, information gets added or removed to the cell state via gates. The gates are different neural networks that decide which information is allowed on the cell state. The gates can learn what information is relevant to keep or forget during training.



(Information flow in a LSTM cell)
Check out hadamard product for vectors

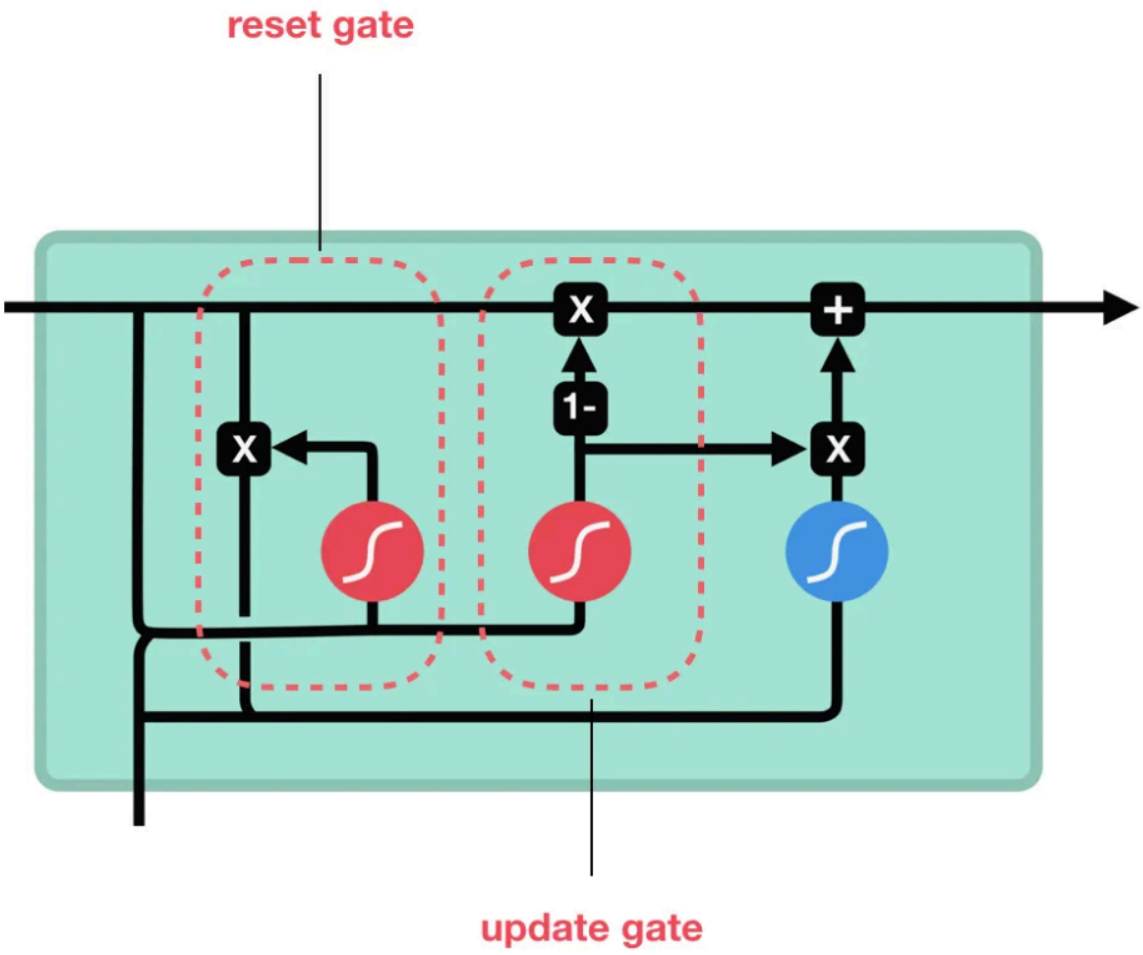
Tanh activation

The tanh activation is used to help regulate the values flowing through the network. The tanh function squishes values to always be between -1 and 1. When vectors are flowing through a neural network, it undergoes many transformations due to various math operations. So imagine a value that continues to be multiplied by let's say ***3***. You can see how some values can explode and become astronomical, causing other values to seem insignificant.

Sigmoid

Gates contains sigmoid activations. A sigmoid activation is similar to the tanh activation. Instead of squishing values between -1 and 1, it squishes values between 0 and 1. That is helpful to update or forget data because any number getting multiplied by 0 is 0, causing values to disappear or be "forgotten." Any number multiplied by 1 is the same value therefore that value stays the same or is "kept." The network can learn which data is not important therefore can be forgotten or which data is important to keep.

GRU



Update Gate

The update gate acts similar to the forget and input gate of an LSTM. It decides what information to throw away and what new information to add.

Reset Gate: The reset gate is another gate is used to decide how much past information to forget. And that's a GRU. GRU's has fewer tensor operations; therefore, they are a little speedier

to train then LSTM's. There isn't a clear winner which one is better. Researchers and engineers usually try both to determine which one works better for their use case.

Limitations of RNNs, LSTMs, and GRUs:

1. **Difficulty with Long-Term Dependencies**:
    - Even though LSTMs and GRUs improve upon basic RNNs by mitigating the vanishing gradient problem, they still struggle with very long-term dependencies. They can remember information over many time steps, but as sequences grow longer, even these models may fail to capture distant dependencies effectively.
2. **Computationally Expensive**:
    - LSTMs and GRUs involve multiple gates and complex operations (like the cell state and forget/update gates in LSTMs), making them more computationally expensive than simpler RNNs. This complexity increases training time and resource consumption.
3. **Sequential Processing**:
    - RNNs, LSTMs, and GRUs process sequences step-by-step, which means they cannot be parallelized effectively for training (unlike transformers).

This makes them slower in practice, especially for long sequences, as each step depends on the previous one.

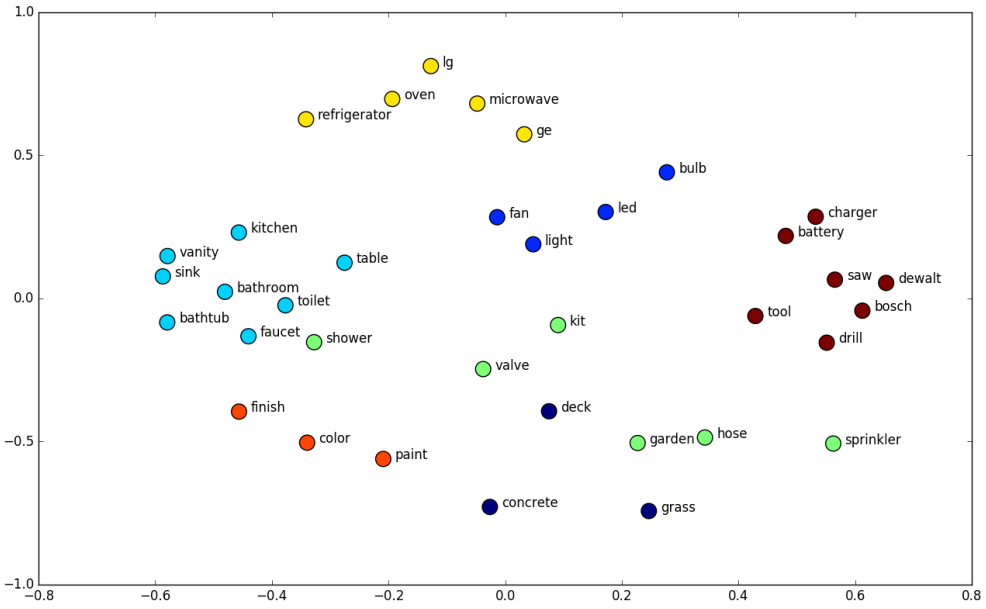
4. **Memory and Storage Requirements**:
    - RNNs, LSTMs, and GRUs require maintaining hidden states for each time step, leading to higher memory usage. This can be problematic when processing very long sequences or using a large batch size.
5. **Sensitivity to Initialization**:
    - The performance of RNNs, LSTMs, and GRUs is sensitive to weight initialization. Poor initialization can exacerbate issues like vanishing or exploding gradients, and finding the right initialization is non-trivial.
6. **Difficulty with Highly Noisy Data**:
    - These models may not perform well when the data is highly noisy or unstructured. RNNs, LSTMs, and GRUs can overfit to noise, especially if regularization methods are not employed effectively.
7. **Limited Parallelism**:
    - Because of their sequential nature, RNNs, LSTMs, and GRUs are not inherently suited for efficient parallel processing, making them slower to train compared to models like transformers that allow for parallel computation.

While LSTMs and GRUs are improvements over vanilla RNNs in handling long-term dependencies and mitigating some of the gradient issues, they still share many of these

limitations, which is why alternative architectures like transformers are becoming

increasingly popular for sequence-based tasks.

Word embeddings are a representation of words in a continuous vector space where similar words are mapped to similar points in that space. They are a type of distributed representation for text, capturing semantic relationships and contextual meaning. Here's how they work:



---

1. Key Idea

Words are represented as dense vectors (real-valued vectors of fixed size) instead of sparse, high-dimensional representations like one-hot encoding. The embeddings encode semantic similarity so that words with similar meanings are close together in the vector space.

For example:

- "king" might be close to "queen" in the vector space.
- The relationship between "man" and "woman" is similar to that between "king" and "queen" (captured mathematically, e.g., via vector arithmetic).

---

2. How Word Embeddings Are Learned

Word embeddings are typically learned through unsupervised training on a large corpus of text. There are two major approaches:

a) Predictive Models:

These models learn embeddings by predicting a word given its context or vice versa. Two well-known examples are:

- **Word2Vec**:
    - *Skip-gram*: Predict surrounding context words given the target word.
    - *CBOW (Continuous Bag of Words)*: Predict the target word given surrounding context words.
- Training involves adjusting the embedding vectors so that predictions minimize the error.
- Example: "The cat sat on the **mat**" → The embedding of "mat" gets adjusted to align with "cat," "sat," and "on."

b) Count-based Models:

These models use co-occurrence statistics to learn embeddings:

- **GloVe (Global Vectors for Word Representation)**:
    - Learns embeddings by factoring a word co-occurrence matrix, capturing the statistical information of words appearing together.
    - Example: Words like "king" and "queen" might co-occur frequently with "royal," leading to similar embeddings.

---

3. Contextual Word Embeddings

Traditional word embeddings like Word2Vec and GloVe assign a single vector to each word, regardless of context. This leads to limitations for polysemy (words with multiple meanings).

Contextual embeddings (used in models like **ELMo**, **BERT**, and **GPT**) address this:

- The vector representation of a word depends on the surrounding words, providing context-sensitive embeddings.
- Example: "bank" in "river bank" vs. "bank" in "money bank" will have different embeddings.

---

4. Mathematical Perspective

The embeddings are typically stored in an **embedding matrix** W:

- Each word in the vocabulary corresponds to a row in W.
- The i-th row of W is the vector representation of the i-th word.

During training:

- The vectors are adjusted to minimize a loss function (e.g., cross-entropy loss for Word2Vec).
- Words appearing in similar contexts are optimized to have similar vectors.

---

5. Applications

- **Similarity Search**: Finding similar words using cosine similarity.
- **Text Classification**: Using embeddings as input features for downstream tasks.
- **Machine Translation**: Embeddings help align similar words across languages.
- **Analogy Tasks**: "King - Man + Woman = Queen" is a classic demonstration of embeddings' arithmetic properties.

Advantages of Word Embeddings

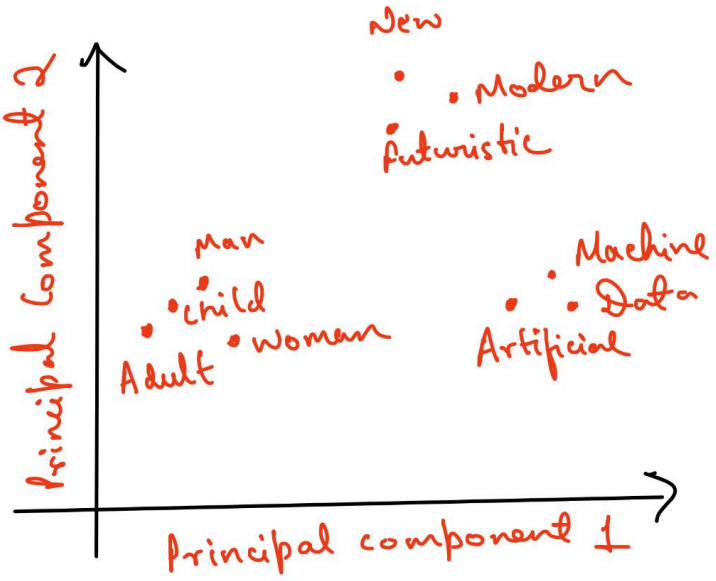
- Compact representation (dense vectors are more memory-efficient than one-hot encodings).
- They generalize well, capturing semantic and syntactic relationships.
- Enable transfer learning for NLP tasks.

Limitations

- Traditional embeddings (e.g., Word2Vec) are static and cannot capture context.
- Pretrained embeddings might not capture domain-specific nuances without fine-tuning.

TASK: mat(about,dataset) += 1

Prepare a dataset of about 400-500 hundred sentences, and then a co-occurance matrix out of its vocab. Normalise the matrix, and apply PCA, to reduce the dimension of the matrix to (vocab_size,2) and plot the embedding vectors of a few relevant words from your vocab like:



Key Differences

Aspect	Co-occurrence Matrix + PCA	Neural Network-Based Embeddings
Construction	Based on word co-occurrence counts and dimensionality reduction.	Learned by optimizing a loss function during training.
Contextuality	Static (word meaning is fixed).	Can be static (Word2Vec/GloVe) or dynamic (BERT/GPT).
Resource Requirements	Low computational cost.	High computational cost, especially for large models.
Representational Power	Limited to global relationships.	Captures complex syntactic and semantic relationships.
Interpretability	More interpretable (principal components).	Less interpretable (neural network parameters).

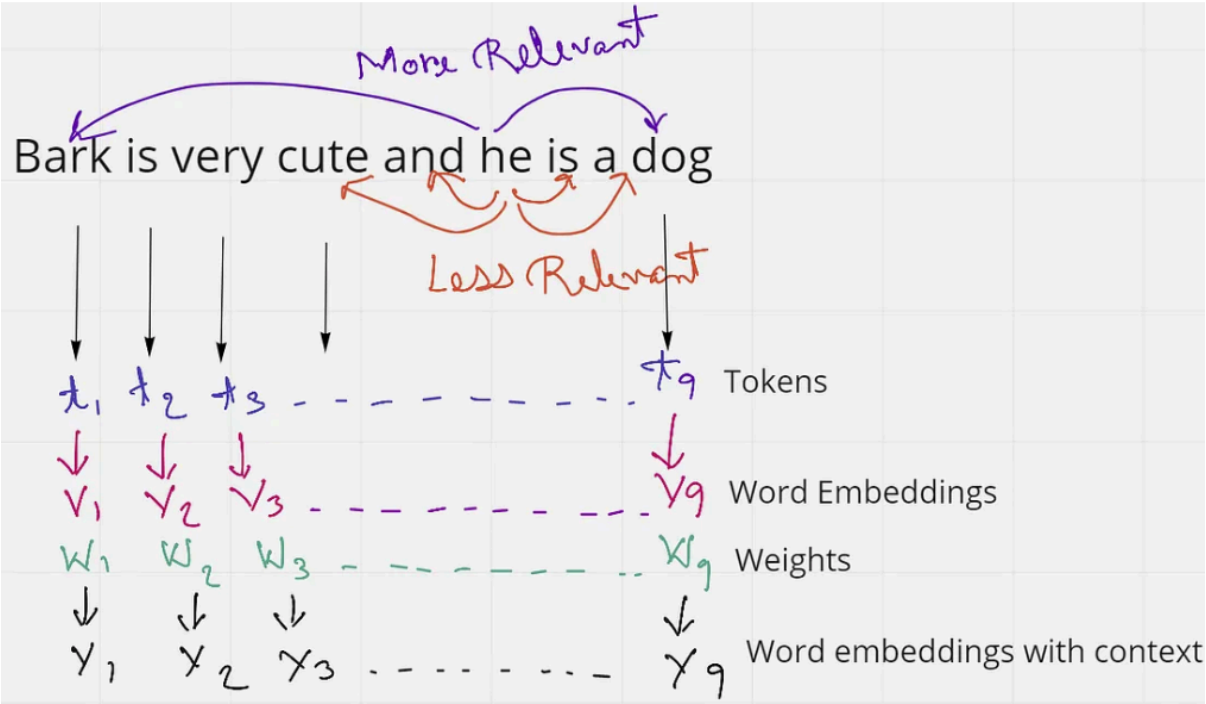
Adaptability to Context	Cannot handle polysemy or context shifts.	Modern embeddings dynamically adjust to context.

When to Use Which?

- **Co-occurrence Matrix + PCA**:
  - Suitable for smaller datasets or when computational resources are limited.
  - Useful for exploring basic word relationships or as a teaching tool for distributional semantics.
- **Neural Network-Based Embeddings**:
  - Ideal for modern NLP applications requiring rich representations
  - Use static embeddings (Word2Vec, GloVe) for simpler tasks.
  - Use contextual embeddings (BERT, GPT) for semantical understanding

Transformers

Suppose we have a statement, “Bark is very cute, and he is a dog.” If we want to check what the word “he” is attending to, one thing we can try to do is to calculate their dot product similarity. But how do we find dot product of two words? By first finding out their embedding vectors….
The complete process of a simple attention mechanism is given below:



1. Finding the Weights

$v_1 v_1 = w_{11}$
$v_1 v_2 = w_{12}$
$v_1 v_3 = w_{13}$
$\vdots$
$v_1 v_9 = w_{19}$

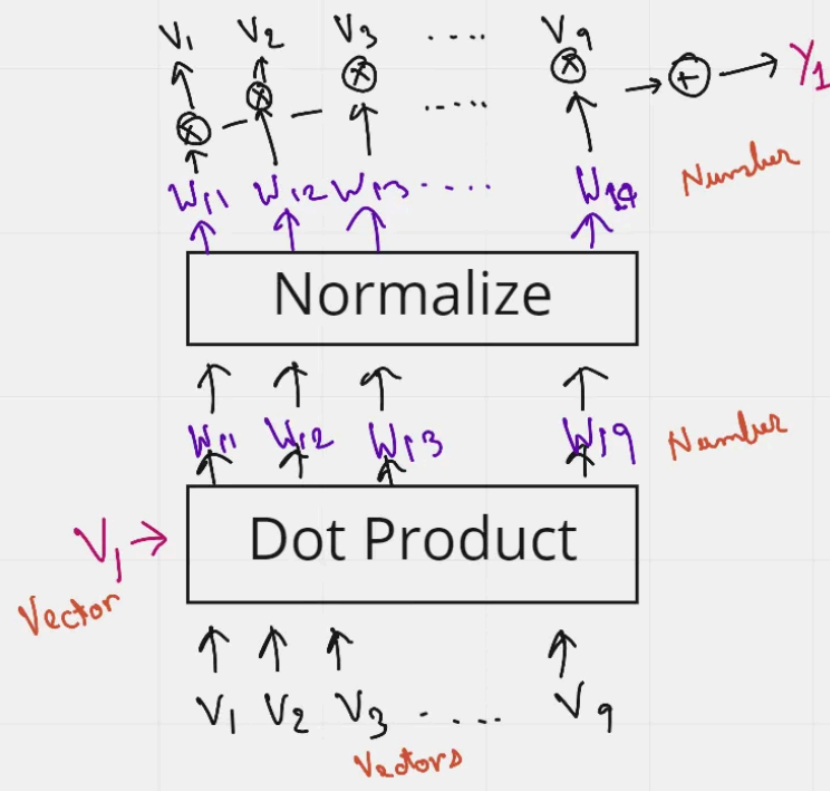
Normalize →

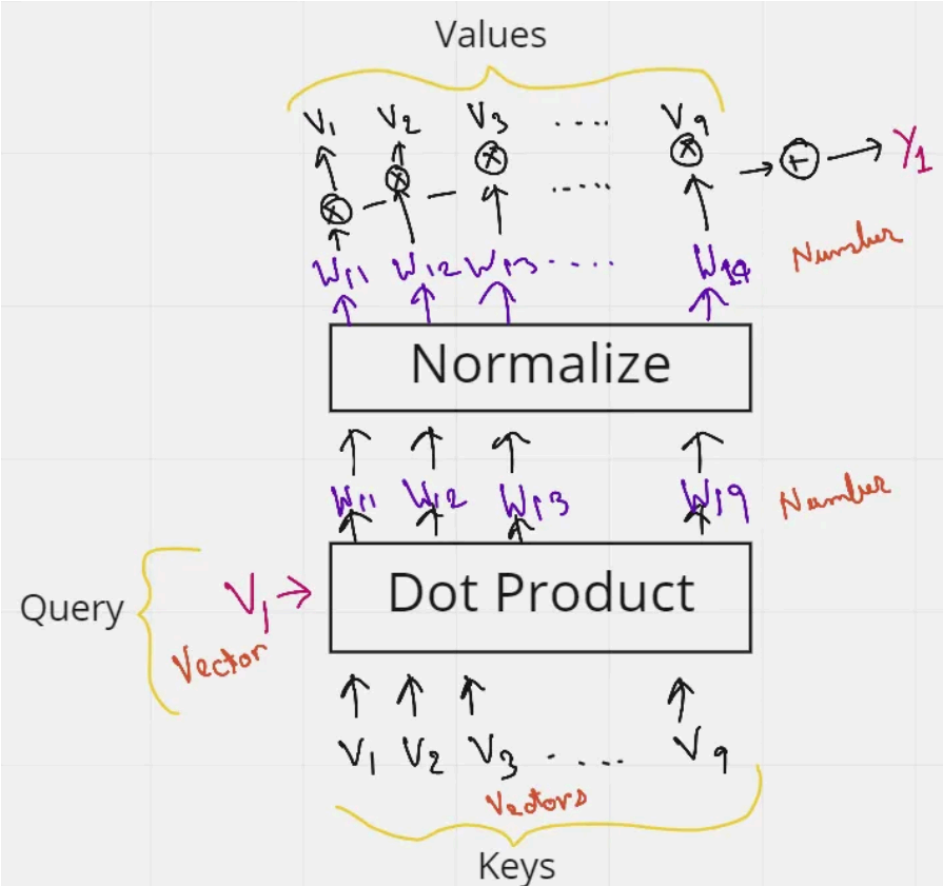
w_{11}
w_{12}
w_{13}
$\vdots$
w_{19}

Weights to re-weigh the first vector

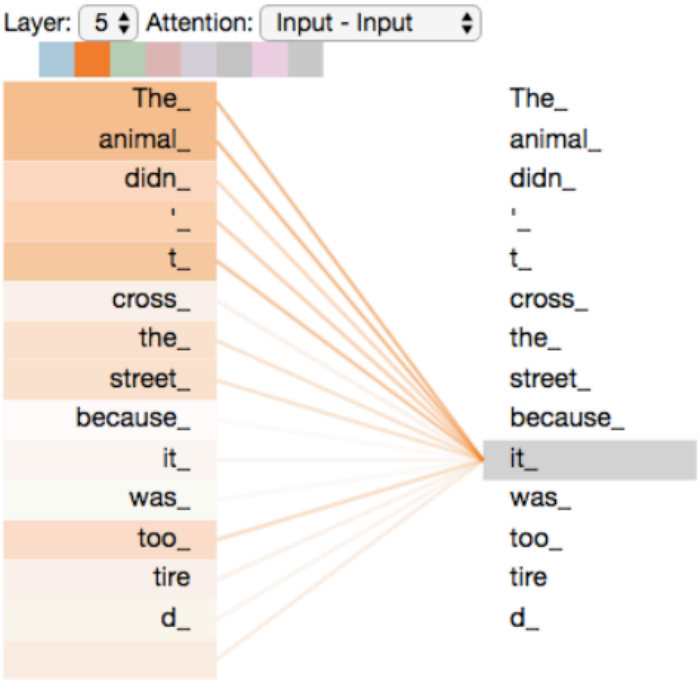
2 Obtaining Embedding with context

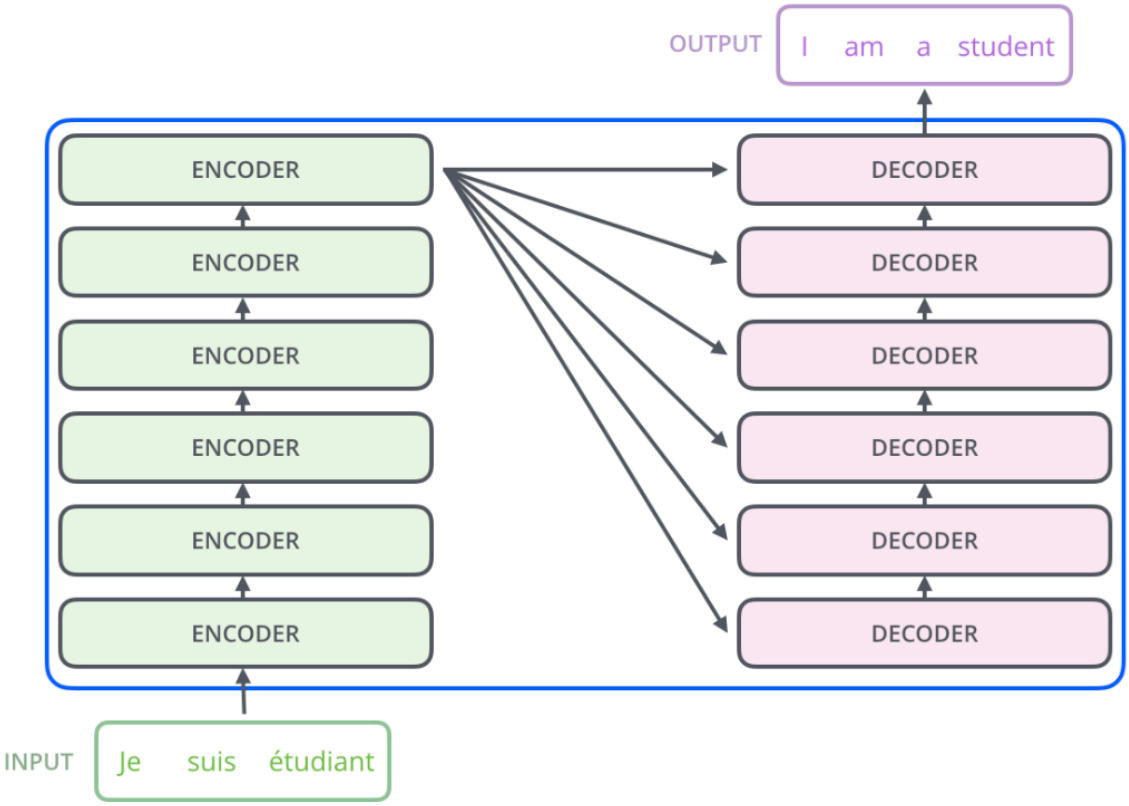
$w_{11} v_1 + w_{12} v_2 + w_{13} v_3 \ldots + w_{19} v_9 = y_1$
$w_{21} v_1 + w_{22} v_2 + w_{23} v_3 \ldots w_{29} v_9 = y_2$
$\vdots$
$w_{91} v_1 + w_{92} v_2 + w_{93} v_3 \ldots w_{99} v_9 = y_9$



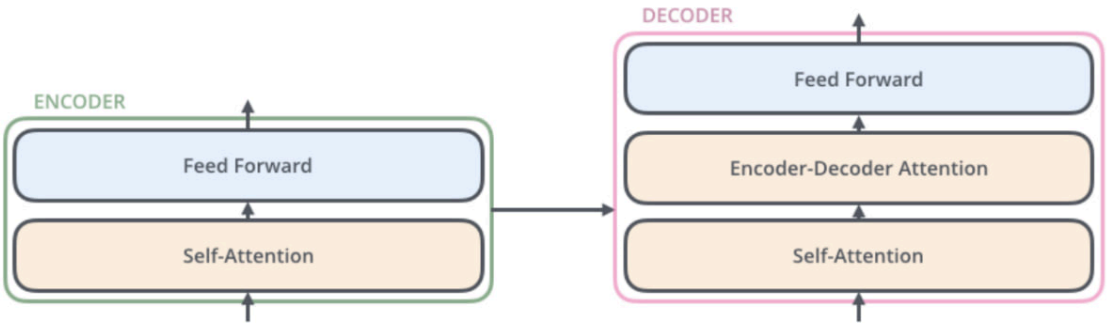


A simple example to visualize self attention:

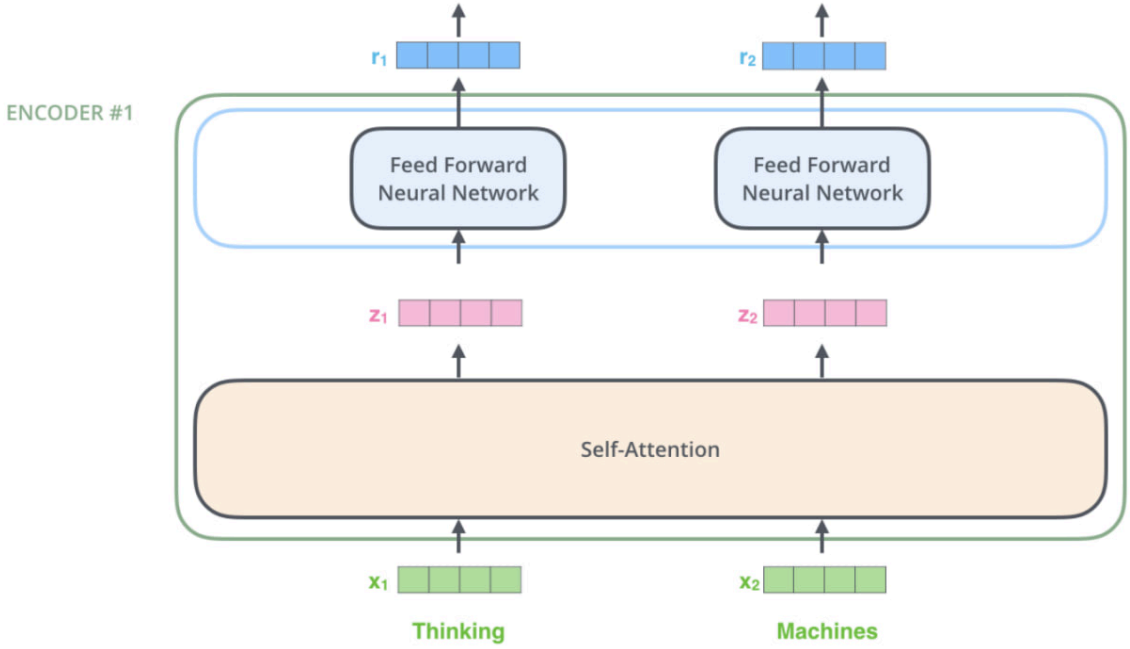




A high level diagram of transformers



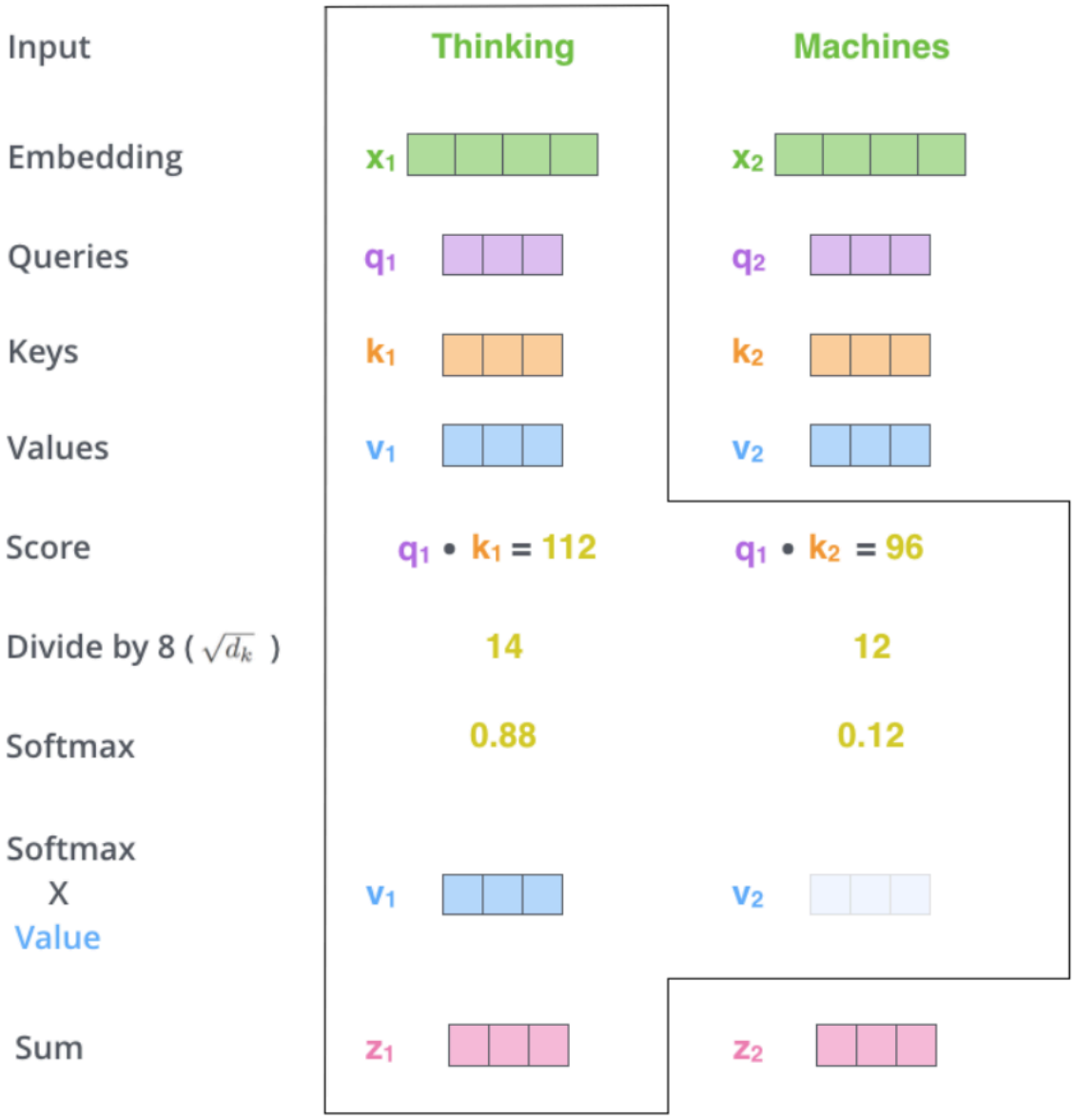
Inside an ecdec



The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network -- the exact same network with each vector flowing through it separately.

For a particular encoder

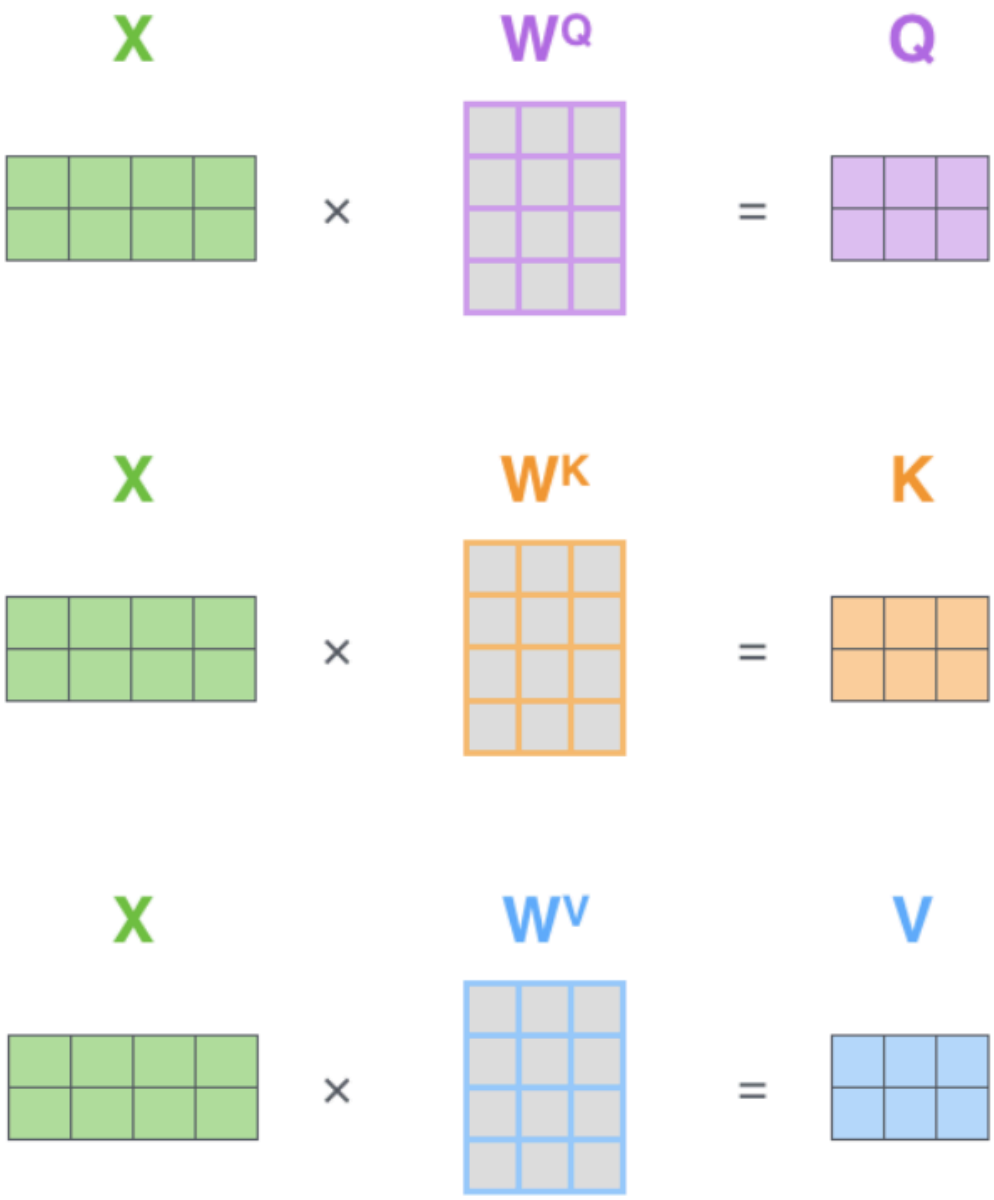
(Note that this is the first encoder, so its input are the words from the input seq, and from the next encoder, their inputs (R) will be the outputs of the previous encoder)

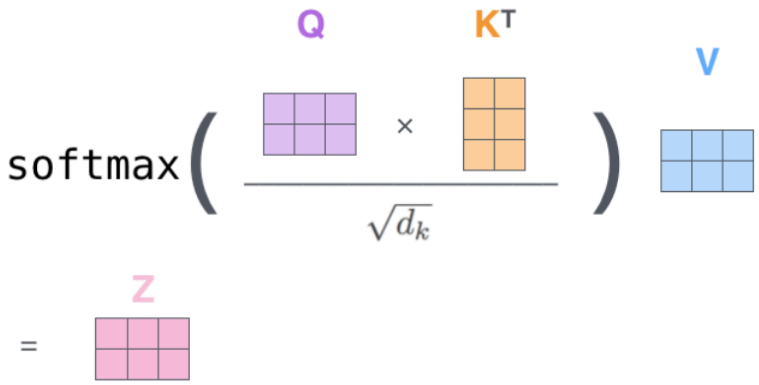


What goes inside the self attention block?
X mul (WQ, WK, WV)

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

We can dictate the entire process in terms of matrices instead of talking about individual vectors


X
W^Q
Q
×
=
X
W^K
K
×
=
X
W^V
V
×
=



Multihead attention

Exactly! You've got the concept right! The idea behind multi-head attention is that each attention head can focus on different parts of the input sequence and capture different kinds of relationships between words.

Let's break it down using example sentence:
"Cat was cute and it sat on a mat."

How Multi-Head Attention Might Work Here:

- **Head 1** might focus on **syntax** (like identifying the subject of the sentence, i.e., "cat" as the noun).
- **Head 2** could focus on **adjective-noun relationships**, associating "cute" with "cat" to capture the semantic connection between the adjective and the noun.
- **Head 3** could focus on **verb-object relationships**, associating "sat" with "mat" (the object of the verb "sat").
- **Head 4** could capture **pronoun-referent relationships**, associating "it" with "cat," understanding that "it" refers back to the "cat" (coreference resolution).

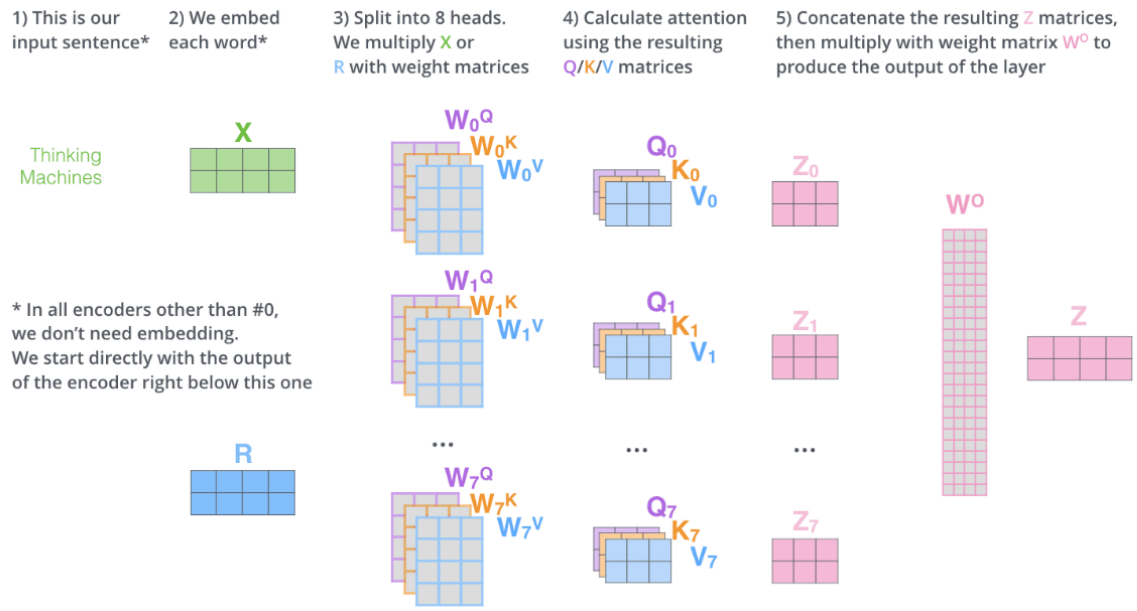
How This Works in Practice:

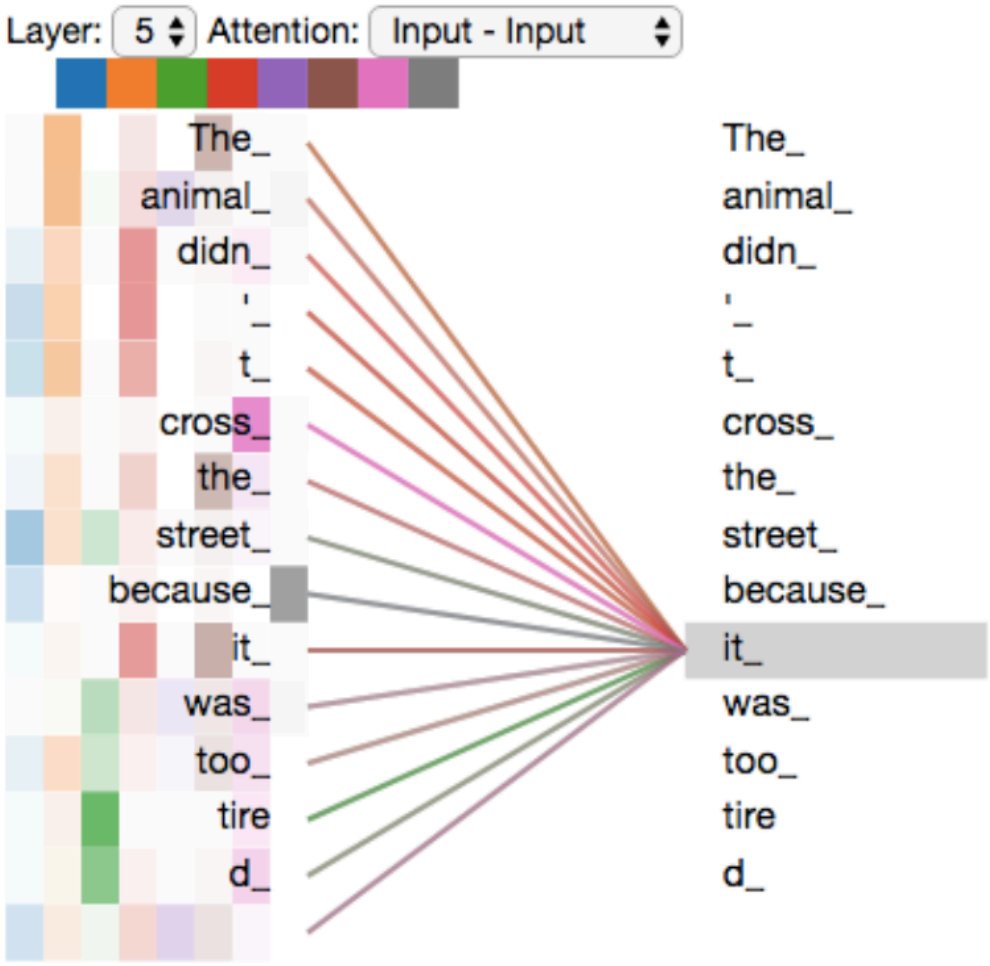
- The model doesn't explicitly "decide" to focus on certain parts like "noun" or "verb" in a fixed manner. Instead, through training, each attention head learns to focus on different patterns and relationships that help it make the most accurate predictions for the task at hand.

- The different heads allow the model to simultaneously capture different aspects of the sentence (e.g., syntactic relationships, semantic relationships, etc.) without explicitly dividing these tasks.

So in a sense, one attention head can focus on nouns, another on verbs, another on adjectives, and so on. This parallel attention mechanism allows the model to understand the sentence from multiple perspectives simultaneously, improving its ability to capture complex linguistic structures.

Final flow chart to explain the process with mutihead attention





In this diagram, pay attention to how different attention heads are attending the word “it”

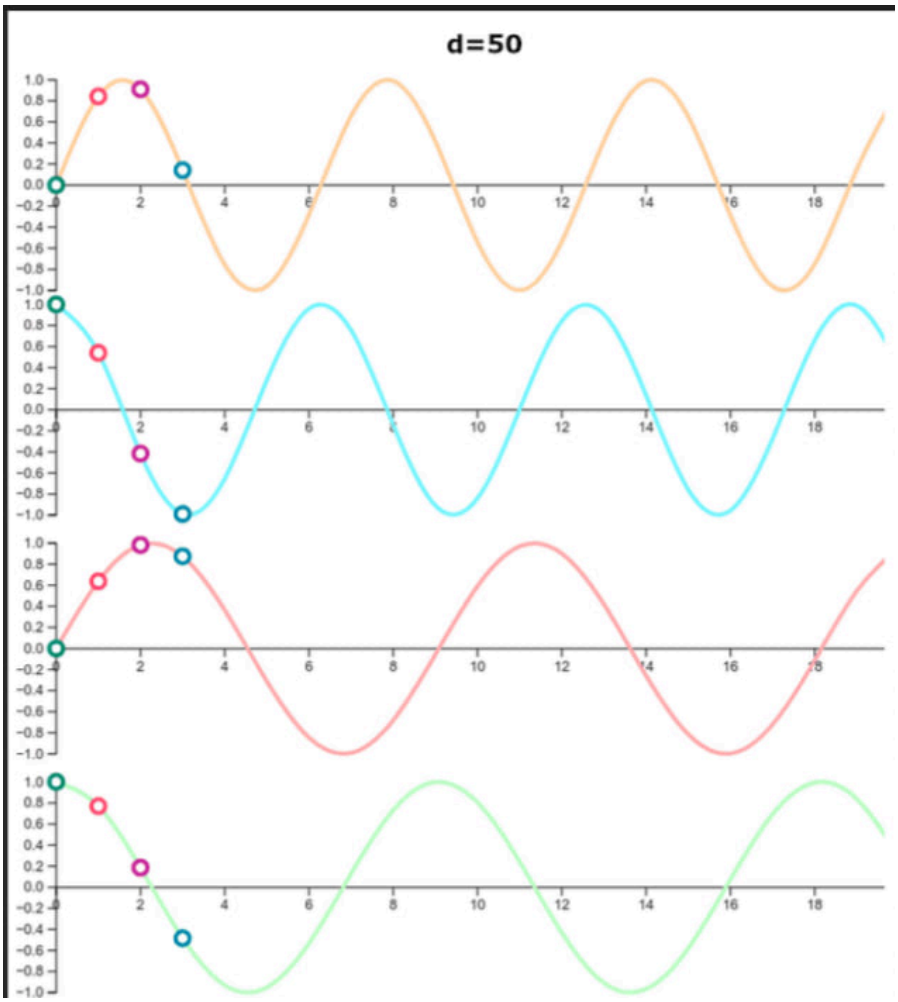
Positional encoding

What is Positional Encoding?

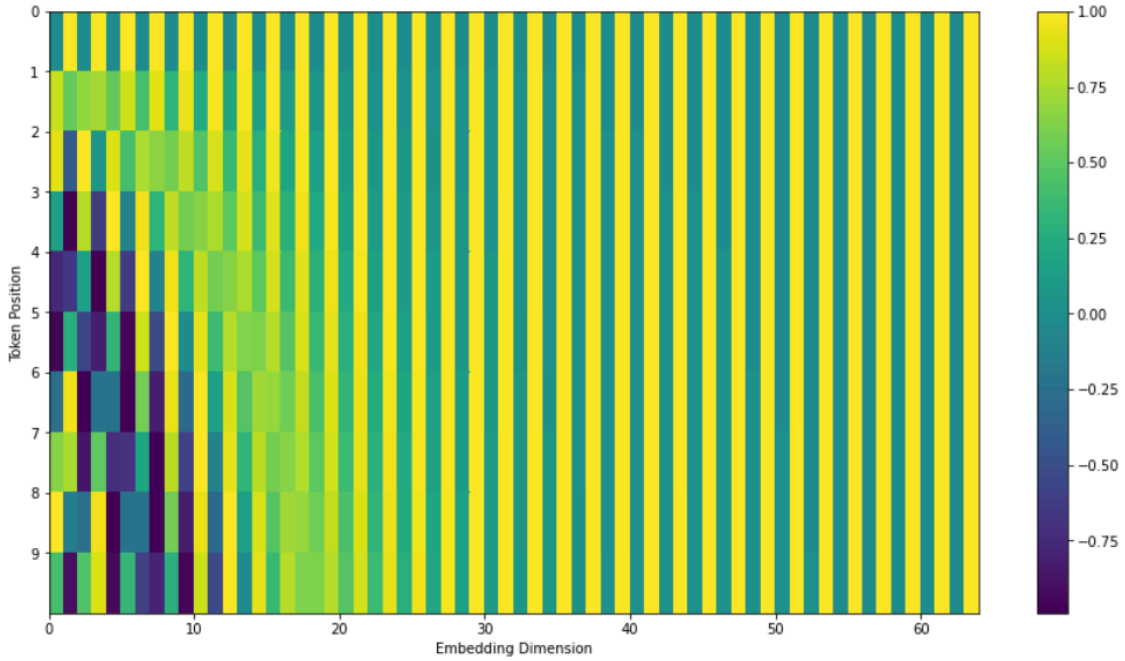
In transformers, **positional encoding** is a mechanism to inject information about the position of each token in the sequence into the model. This is essential because transformer architectures, unlike RNNs, do not inherently understand the order of input tokens—they process input sequences in parallel.

Positional encodings enable transformers to capture the sequence's order, which is critical for tasks like natural language processing (NLP) and audio processing, where the order of tokens or features conveys meaning.

How is Positional Encoding Used?

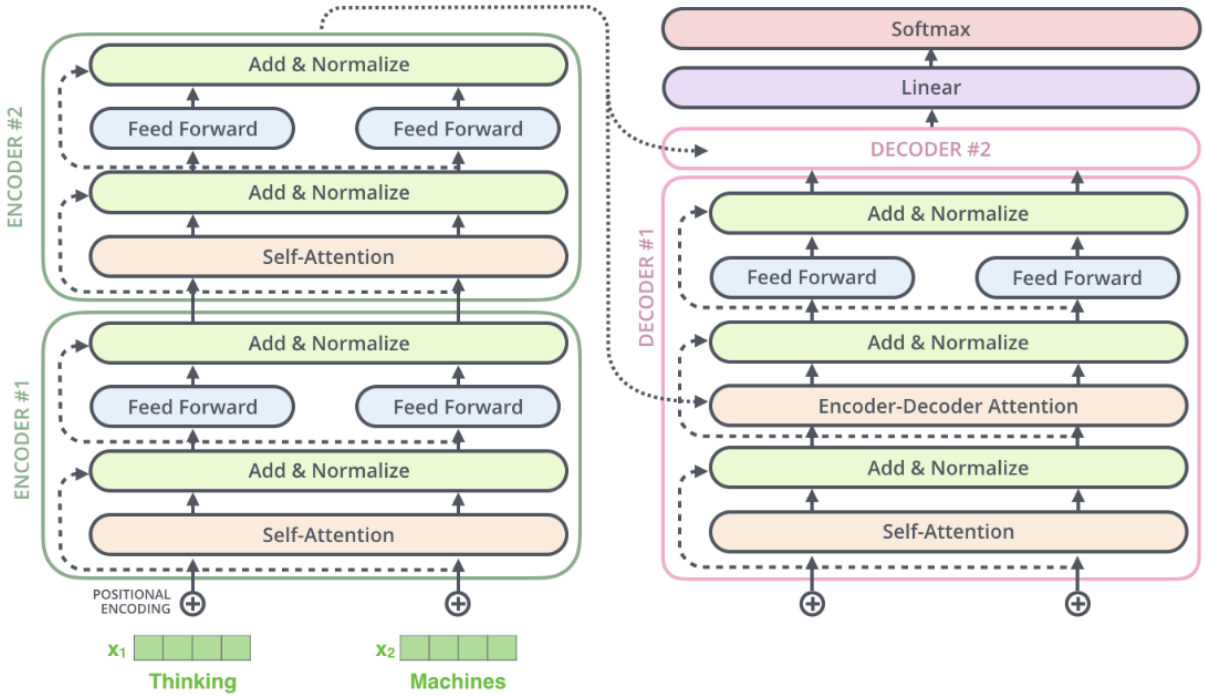
1. **Addition to Input Embeddings**:
    - Positional encodings are added to the input embeddings of tokens before feeding them into the transformer model.
    - If E is the embedding matrix and P is the positional encoding matrix, the input to the first transformer layer is: Z=E+P
2. **Learned or Fixed Encodings**:
    - **Learned Positional Encodings**: Parameters are trained along with the model, similar to word embeddings.
    - **Fixed Positional Encodings**: Use pre-defined mathematical functions (e.g., sine and cosine functions).
    - 

Say for ith time step, wave is sin(2pi*t)
For input[0] —> positional encoding = [sin1(0), sin2(0), …. sini(0)....]
For input[i] —> positional encoding = [sin1(i), sin2(i), …. sini(i)....]



What can you observe from this pic?

Residual Connection



We see that here, a new layer named, “add and normalize” has been added

For vectors x —> z,

[x + z] → new vector → normalise

Fine tuning pre-trained transformers

In case of any discrepancies or references, check out: AutoCompose.ipynb
After loading the datasets, and converting them into the forms suitable to be into the models, we now move towards the final steps:
Imports:

```
from transformers import GPT2Tokenizer, GPT2LMHeadModel, GPT2Config,
AdamW, get_linear_schedule_with_warmup
```

Setting the device:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
device
```

(you would like to train your model on GPU to reduce the training time drastically)

Steps before training loop

1) Pretrained model:

```
# Load model configuration
config = GPT2Config.from_pretrained("gpt2")
```

(as we did before)

2) Creating a model instance

```
# Create model instance and set embedding length
model = GPT2LMHeadModel.from_pretrained("gpt2", config=config)
model.resize_token_embeddings(len(tokenizer))
```

3) Shifting the model to run on GPU

```
# Running the model on GPU
model = model.to(device)
```

4) Setting up the parameters

```
epochs = 4
warmup_steps = 1e2
sample_every = 100
```

epochs: The number of complete passes through the training dataset. Increasing epochs allows the model to learn more but risks overfitting if too high.

`warmup_steps`: A small number of initial training steps where the learning rate gradually increases, helping the model stabilize before using the full learning rate.
`sample_every`: Determines how frequently (in terms of training steps) you generate a sample or evaluate the model to monitor its performance.

5) Scheduling optimizer:

```
# Using AdamW optimizer with default parameters
optimizer = AdamW(model.parameters(), lr=5e-4, eps=1e-8)

# Toatl training steps is the number of data points times the number
of epochs
total_training_steps = len(train_dataloader)*epochs

# Setting a variable learning rate using scheduler
scheduler = get_linear_schedule_with_warmup(optimizer,

num_warmup_steps=warmup_steps,

num_training_steps=total_training_steps)
```

Optimizer setup (`AdamW`):

1) Uses the AdamW optimizer, which combines adaptive learning rates with weight decay for better generalization.
2) Parameters:
    a) `lr=5e-4`: Initial learning rate.
    b) `eps=1e-8`: Small value to prevent division by zero in calculations.

Total training steps:

- Computes the total number of steps for training as `len(train_dataloader) * epochs`, where:
    c) `len(train_dataloader)` is the number of batches in one epoch.
    d) `epochs` is the number of complete passes through the dataset.

Learning rate scheduler:

- Uses `get_linear_schedule_with_warmup` to adjust the learning rate during training:
    e) `num_warmup_steps`: Number of initial steps for gradual learning rate increase.
    f) `num_training_steps`: Total training steps for linear decay after warm-up.

6) This function is typically used to log or display training durations in a clear and readable format.

```
def format_time(elapsed):

  return str(datetime.timedelta(seconds=int(round(elapsed))))
```

Training loop

Initialization

```
total_t0 = time.time()
training_stats = []
model = model.to(device)
```

Explanation:

- Records the total start time for tracking training duration.
- Prepares a list (`training_stats`) to store metrics like loss and time for each epoch.
- Moves the model to the device (CPU/GPU) for computation.

Epoch Start

```
for epoch_i in range(epochs):

  print(f'Beginning epoch {epoch_i+1} of {epochs}')



  t0 = time.time()

  total_train_loss = 0

  model.train()
```

Explanation:

- Begins a new epoch.
- Logs the epoch number, resets the epoch start time, and cumulative training loss.
- Puts the model in training mode.

Processing Training Batches

```
for step, batch in enumerate(train_dataloader):

    b_input_ids = batch[0].to(device)

    b_labels = batch[0].to(device)

    b_masks = batch[1].to(device)



    model.zero_grad()



    outputs = model(b_input_ids,

                    labels=b_labels,

                    attention_mask=b_masks)



    loss = outputs[0]



    batch_loss = loss.item()

    total_train_loss += batch_loss
```

Explanation:

- Loops through each batch in the training data.
- Sends input IDs, labels, and attention masks to the device.
- Resets gradients from the previous step.
- Performs a forward pass to compute the loss for the batch.
- Tracks the loss for the current batch and accumulates it.

Sampling and Monitoring

```
# Sampling every x steps

    if step != 0 and step % sample_every == 0:



      elapsed = format_time(time.time()-t0)

      print(f'Batch {step} of {len(train_dataloader)}. Loss: {batch_loss}.
Time: {elapsed}')

      model.eval()



      sample_outputs = model.generate(

                                    bos_token_id=random.randint(1,30000),

                                    do_sample=True,

                                    top_k=50,

                                    max_length = 200,

                                    top_p=0.95,

                                    num_return_sequences=1

                                )

      for i, sample_output in enumerate(sample_outputs):

        print(f'Example ouput: {tokenizer.decode(sample_output,
skip_special_tokens=True)}')

      print()

      model.train()
```

Explanation:

- At intervals defined by `sample_every`, logs the elapsed time and current loss.

- Temporarily switches to evaluation mode to generate text samples.
- Uses `model.generate` to produce a sample sequence, decodes it, and displays the output.
- Resumes training mode afterward.

Backpropagation and Optimization

```
    loss.backward()

    optimizer.step()

    scheduler.step()
```

Explanation:

- Computes gradients via backpropagation (`loss.backward`).
- Updates the model's weights using the optimizer.
- Adjusts the learning rate using the scheduler.

Epoch Completion

```
  avg_train_loss = total_train_loss / len(train_dataloader)

  training_time = format_time(time.time()-t0)

  print(f'Average Training Loss: {avg_train_loss}. Epoch time:
{training_time}')

  print()
```

Explanation:

- Computes the average loss over all batches in the epoch.
- Calculates and logs the time taken for the epoch.

Validation

```
t0 = time.time()
model.eval()

total_eval_loss = 0
nb_eval_steps = 0

for batch in val_dataloader:
  b_input_ids = batch[0].to(device)
  b_labels = batch[0].to(device)
  b_masks = batch[1].to(device)

  with torch.no_grad():

      outputs  = model(b_input_ids,
                       attention_mask = b_masks,
                       labels=b_labels)

      loss = outputs[0]

  batch_loss = loss.item()
  total_eval_loss += batch_loss

avg_val_loss = total_eval_loss / len(val_dataloader)
val_time = format_time(time.time() - t0)
print(f'Validation loss: {avg_val_loss}. Validation Time: {val_time}')
print()
```

Explanation:

- Switches to evaluation mode.
- Iterates over the validation data to compute the validation loss.
- Uses `torch.no_grad()` to disable gradient calculations, improving efficiency during validation.
- Logs the average validation loss and validation time.

Logging Statistics

```
  training_stats.append(

      {

          'epoch': epoch_i + 1,

          'Training Loss': avg_train_loss,

          'Valid. Loss': avg_val_loss,

          'Training Time': training_time,

          'Validation Time': val_time

      }

  )

  print("------------------------------")
```

Explanation:

- Appends the metrics (training loss, validation loss, training/validation time) for the epoch to `training_stats`.

Total Training Time

```
print(f'Total training took {format_time(time.time()-total_t0)}')
```

Explanation:

- Logs the total time taken for the entire training process.

After training:

Generating poems/content:

```
model.eval()

prompt = "<|BOS|>..." #feel free to experiment with it

generated = torch.tensor(tokenizer.encode(prompt)).unsqueeze(0)
generated = generated.to(device)

sample_outputs = model.generate(
            generated,
            do_sample=True,
            top_k=50,
            max_length=300,
            top_p=0.95,
            num_return_sequences=3,
            temperature=1.0,
            repetition_penalty=1.2,
            no_repeat_ngram_size=2,
            early_stopping=True
      )


for i, sample_output in enumerate(sample_outputs):
  print("{}: {}\n\n".format(i, tokenizer.decode(sample_output,
skip_special_tokens=True)))
```

Parameter Explanations:

1. **`do_sample`**: Enables random sampling instead of deterministic greedy decoding, encouraging more diverse outputs.
2. **`top_k`**: Limits word selection to the top `k` most probable options, balancing diversity and coherence.
3. **`max_length`**: Defines the maximum number of tokens in the generated output.
4. **`top_p`**: Implements nucleus sampling, selecting tokens from the smallest set whose cumulative probability exceeds `p`.
5. **`num_return_sequences`**: Specifies the number of different sequences to generate per input.

6. **`temperature`**: Adjusts randomness in sampling; values >1 increase diversity, and values <1 make outputs more deterministic.
7. **`repetition_penalty`**: Discourages repetitive n-grams in generated text by penalizing them.
8. **`no_repeat_ngram_size`**: Prevents the repetition of n-grams of the specified size, ensuring variety in output.
9. **`early_stopping`**: Halts generation if an end-of-sequence token is produced, avoiding unnecessary extension of output.

This setup ensures diverse and controlled text generation while mitigating redundancy.

Saving the model:

```
import torch
import os

# Specify the directory where the model will be saved
output_dir = "/content/drive/My Drive/……"

# Check if model uses DataParallel and save the correct model
model_to_save = model.module if hasattr(model, 'module') else model
```

```
# Save the trained model, configuration, and tokenizer
model_to_save.save_pretrained(output_dir)
tokenizer.save_pretrained(output_dir)

print(f'Model and tokenizer saved to {output_dir}')
```

Explanation:

1. **`output_dir`**:
   Specifies the path where the model, tokenizer, and training stats will be saved.

2. **`model_to_save`**:
   If you're using multiple GPUs with `DataParallel`, the model is wrapped in `model.module`. This checks if the model is wrapped and extracts the underlying model; otherwise, it saves the model directly.

3. **`save_pretrained(output_dir)`**:
   Saves the model and tokenizer in the specified directory. This function saves all necessary files for the model to be reloaded later using `from_pretrained()`.

4. **`print(f'Model and tokenizer saved to {output_dir}')`**:
   Prints a message confirming the save location for the model and tokenizer.

Some of the poems generated by my model:

The night was dark and the wind was slow
As I lay there singing
The refrain of a bird
Haunting the dead.

I thought my song was lovely,
And I sang in a language
Which the gods spoke
In languages
Of things dead and things alive.

———————————————

A lark, singing in the hollow of night,
Is setting fire to a pane, and burning it
With its flaming breath.

Heaven is a night when the air is fair, a day when
Hissing is not, but is still
As he gathers shade from the dark.

———————————————

I want to love you, dear,
Your voice is sweet and your eyes are kind, and I want
To hear your voice in my dreams.

I dreamed a great world, but I fear
The dark and shadowy corners of it;
You are a voice that I can never hear, except
Through dreams or from the deep dark.

———————————————

The sun has already set, and I am pacing home
from a long-term relationship.
I am not yet ready to say goodbye,
or seek another lover. I want to know
if I'm still here. The world seems so far away.

Between us, there were a thousand sparkling stars...
And circling us now, a hundred little stars.

But to-day I can only see the bright limb of your wrist
From my vision. Only a feeble wish
Had gone along—
That all the stars were pointing to the same place
In that farthest corner of our sky.

The night was starry, and so were her eyes!
And now, a few minutes ago,
They shone with the splendour of love,
In the garden of the south,
In a little garden near
The village;

But now she is gone, and the moon
Deems to fade away, and the air
Is full of darkness, and cold
Till the shadows and the mist
Come drifting back.
But my heart is still
And I feel it.

(My dataset wasn’t that large so it lacks consistency sometimes)