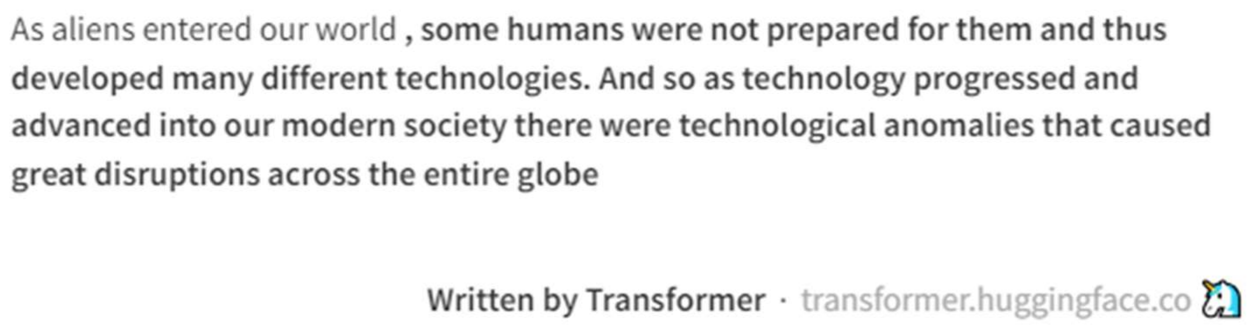
BERT & GPT

Homework – SE Digital Organization

by David Cloos

universität innsbruck

GPT2 - Transformer



As aliens entered our world , **some humans were not prepared for them and thus developed many different technologies. And so as technology progressed and advanced into our modern society there were technological anomalies that caused great disruptions across the entire globe**

Written by Transformer · transformer.huggingface.co 🦄

[1]https://transformer.huggingface.co/doc/gpt2-large

Model architecture – "The Transformer"

Attention Is All You Need

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

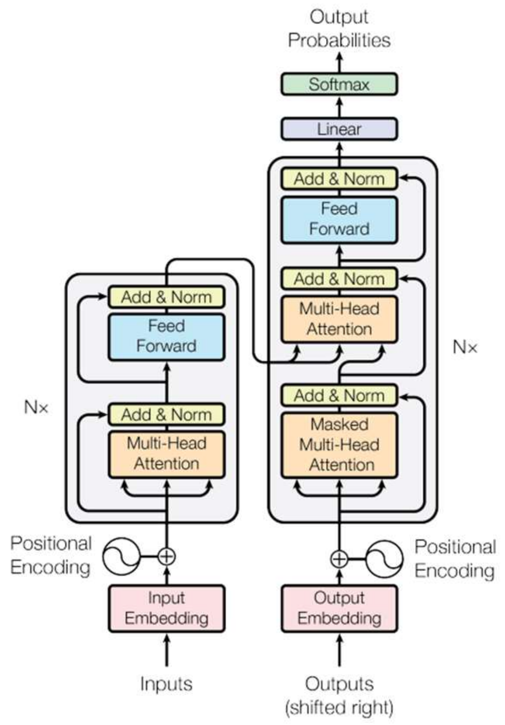


Figure 1: The Transformer - model architecture.

„Attention is all you need"
  - Infinite reference window → references from (almost) unlimited amount of sequences before
  - Identify all relevant information within a sequence → Attention

- Encoder
  - N=6 identical layers → repeated 6 times
  - Contains two sublayers
    - multi head self attention mechanism
    - Feed forward network
  - Surrounded by residual connection and followed by layer normalization
- Decoder
  - N=6 identical layers
    - Contains a third sublayer
      - Multi head attention on the input of the Encoder (cross attention)

[1]Vaswani et al., (2017)

Encoder

1. Input Embedding:
   - Sequence of tokens (words, sentences), represented as a vector (embedding) of dimension 512 for each token
   - Embeddings do not account for position of which tokens might appear within a sequence
2. Positional Encoding
   - Encode relative position of given token in a sequence (using sine and cosine functions)
3. Multi-Self-Attention
   - Compute attention scores for each tokens in the sequence → Attention scores used to compute weighted sums of the embedding
   - E.g., "I love cats" → "Cats" in combination with "love" in a sequence might get higher weights
   - Result:
     - Score Matrix: measure of importance of the tokens within a sequence

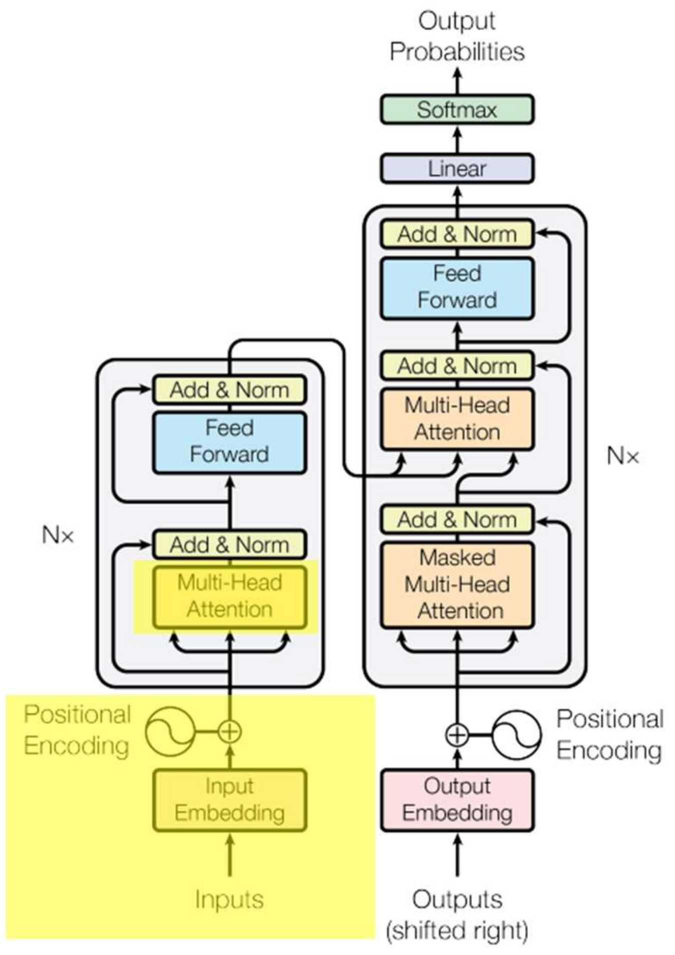


Figure 1: The Transformer - model architecture.

[1]Vaswani et al., (2017)

Encoder

4. Residual connection (training part)

- Auto regressive approach → take inputs from multi head attention vector and add it to the original input
- Purpose is to effectively propagate information through deep neural networks and enable "backpropagation"

4.1 Feed Forward Network

- Non-linear activation function between fully connected layers (ReLu) → capture more complex relationships
- E.g.: learns that "love cats" corresponds to a positive sentiment

4.2 Layer normalization (handles varying length of sequences)

- All input values in all neurons in the same layer are normalized (same mean and variance)

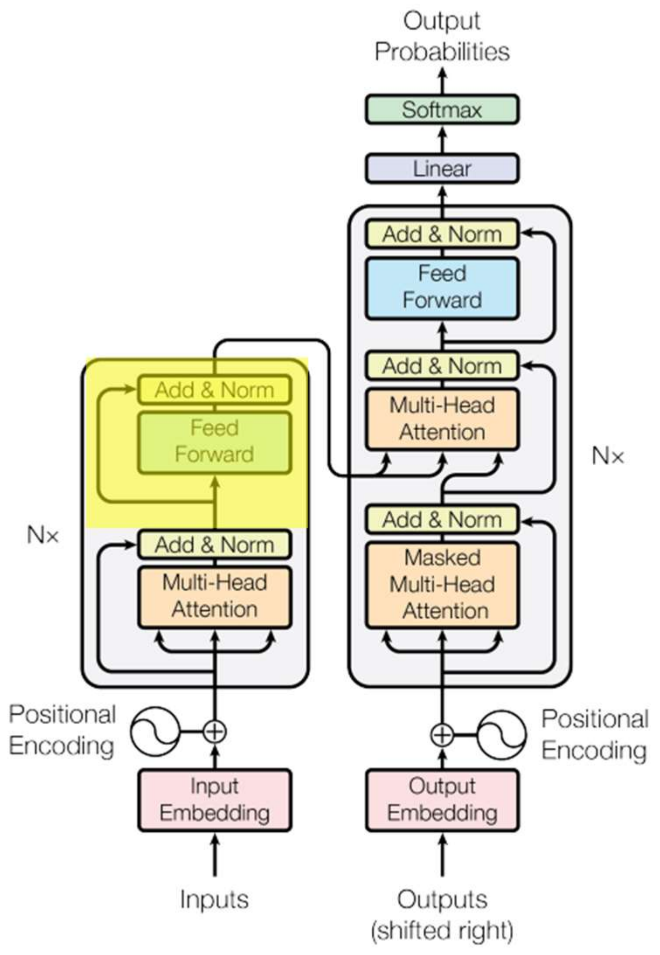
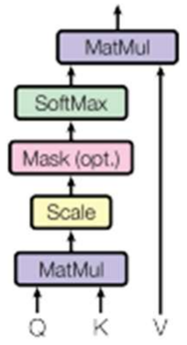


Figure 1: The Transformer - model architecture.

[1]Vaswani et al., (2017)

Attention



- Attention:
  - Input:
    - An attention function is a mathematical operation that takes a question (query) and a collection of related information (key-value pairs) as inputs → represented as vectors
  - Output
    - taking a weighted sum of the information vectors (values), where each weight is determined by how well the question vector (query) matches the corresponding information vector (key)

Short Example

Query:

- Query on YouTube

Key-value pairs:

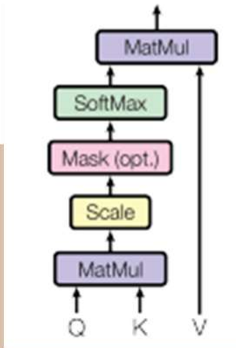
- Keys: Video title, Description etc. will be mapped against the query
- Value: Suggested videos from the database

[1]Vaswani et al., (2017)

Attention

- Attention:
  - Input:
    - An attention function i
      (key-value pairs) as inp
  - Output
    - taking a weighted sum
      vector (query) matches

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$



Example
Word Vectors(Embeddings + positional encoding):
•"The": [0.2, 0.5, 0.3]
•"cat": [0.7, 0.1, 0.4]
•"sat": [0.6, 0.9, 0.2]
•"on": [0.3, 0.4, 0.7]
•"the": [0.1, 0.8, 0.5]
•"mat": [0.9, 0.3, 0.6]

Now, let's say we want to compute the attention for the word "cat" (the query) in the sentence.
1.Query (Q): [0.7, 0.1, 0.4]
2.Keys (K): [[0.2, 0.5, 0.3], [0.7, 0.1, 0.4], [0.6, 0.9, 0.2], [0.3, 0.4, 0.7], [0.1, 0.8, 0.5], [0.9, 0.3, 0.6]]
3.Values (V): [[0.2, 0.5, 0.3], [0.7, 0.1, 0.4], [0.6, 0.9, 0.2], [0.3, 0.4, 0.7], [0.1, 0.8, 0.5], [0.9, 0.3, 0.6]]

Now, we compute the attention scores between the query and each key. This is done by taking the dot product between the query and each key vector:
•Attention scores: [0.14, 1.0, 0.43, 0.59, 0.39, 0.64]

Next, we scale the attention scores by dividing them by the square root of the dimensionality of the key vectors. Assuming the dimensionality is 3:
•Scaled attention scores: [0.14 / sqrt(3), 1.0 / sqrt(3), 0.43 / sqrt(3), 0.59 / sqrt(3), 0.39 / sqrt(3), 0.64 / sqrt(3)]

We then apply a softmax function to obtain the attention weights:
•Attention weights: [0.08, 0.21, 0.11, 0.15, 0.10, 0.16]

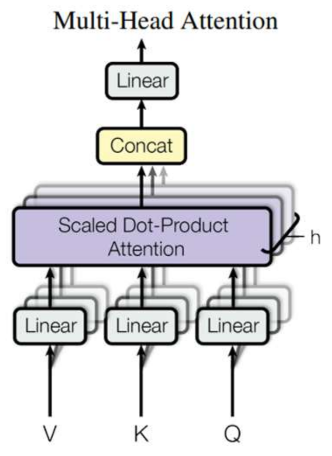
Finally, we compute the weighted sum by multiplying each attention weight by its corresponding value vector:
•Weighted sum: [0.08 * [0.2, 0.5, 0.3] + 0.21 * [0.7, 0.1, 0.4] + 0.11 * [0.6, 0.9, 0.2] + 0.15 * [0.3, 0.4, 0.7] + 0.10 * [0.1, 0.8, 0.5] + 0.16 * [0.9, 0.3, 0.6]]

The resulting weighted sum represents the attended information or context associated with the word "cat" in the sentence.

[1]Vaswani et al., (2017) | [2]ChatGPT

Multi head self attention

- Stack of parallel Attention layers
  - Helps us to understand different aspects of a sentence
  - Receive generalized & different perspectives of a sequence
- Output of all heads is concetenated back into the dimension of n x 512



Example: He sat on the chair & it broke'. Here, one of the attention heads may associate 'it' with chair & other may associate it with 'he'.

Advantage over traditional recurrent or convolutional NN's:
- Wider range of dependencies and relationships from the input can be captured
  - Attends different information with different representations simultaneously
- Parallel computation across multiple heads or attention layers

Decoder

- Similar to encoder, but purpose is to generates text sequences
- Includes one more layer that takes the output from the encoder (cross attention) and performs multi head self attention on input Queries
  - But keys and values are coming from the encoder
- E.g.: Translation German-English
  - Because of cross attention the decoder gets the representation of attention information by the encoder and can focus to express the semantic and contextual information of the german sentence into english
  - Softmax makes sure to predict the probability for next best fitting word

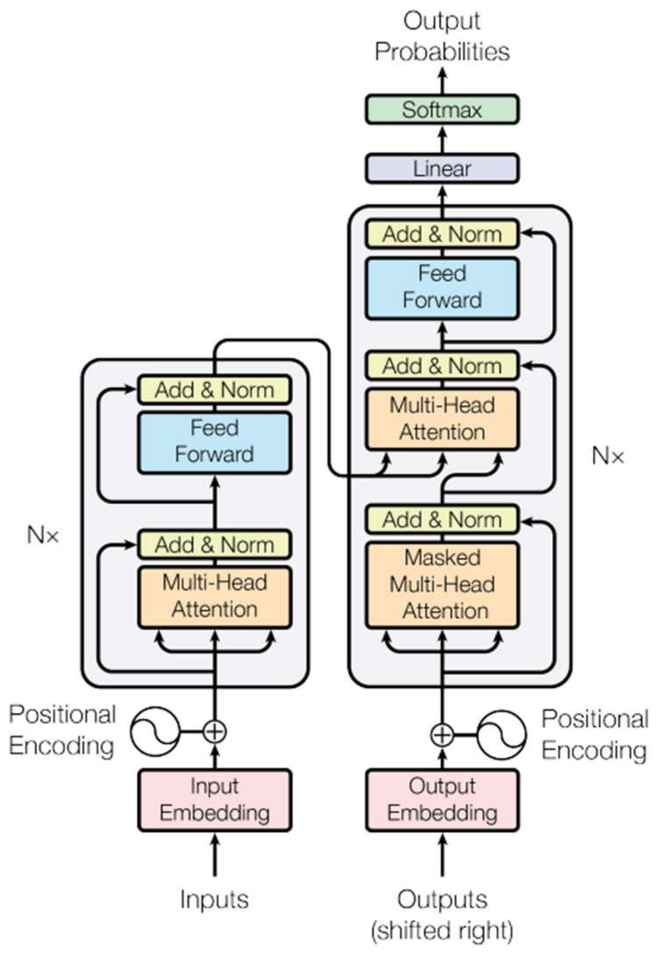
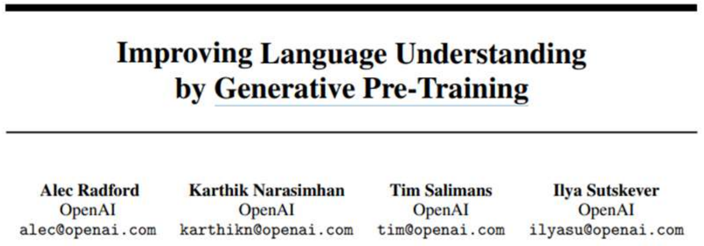


Figure 1: The Transformer - model architecture.

[1]Vaswani et al., (2017)

Generative Pre-Training (GPT)



Improving Language Understanding by Generative Pre-Training

Alec Radford
OpenAI
alec@openai.com

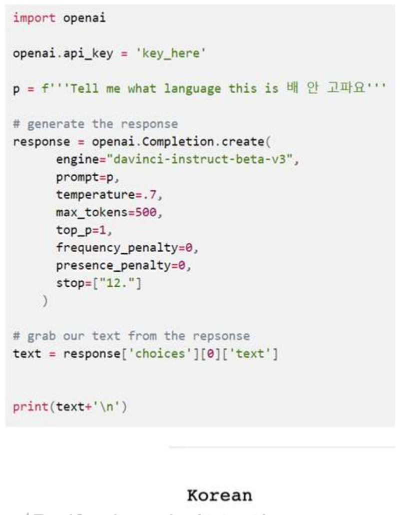
Karthik Narasimhan
OpenAI
karthikn@openai.com

Tim Salimans
OpenAI
tim@openai.com

Ilya Sutskever
OpenAI
ilyasu@openai.com

- *“For our model architecture, we use the **Transformer** presented by Vaswani et al., (2017)” [..] “This model choice provides us with a more structured memory for handling long-term dependencies [..] compared to alternatives like recurrent networks”*

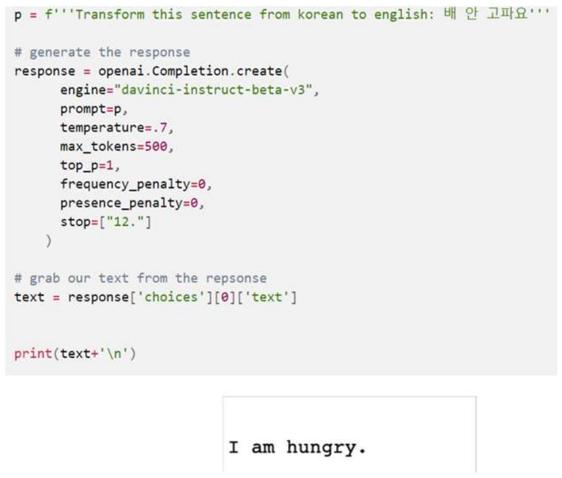
Language detection



```python
import openai

openai.api_key = 'key_here'

p = f'''Tell me what language this is 배 안 고파요'''

# generate the response
response = openai.Completion.create(
      engine="davinci-instruct-beta-v3",
      prompt=p,
      temperature=.7,
      max_tokens=500,
      top_p=1,
      frequency_penalty=0,
      presence_penalty=0,
      stop=["12."]
    )

# grab our text from the repsonse
text = response['choices'][0]['text']


print(text+'\n')
```

```
Korean
```

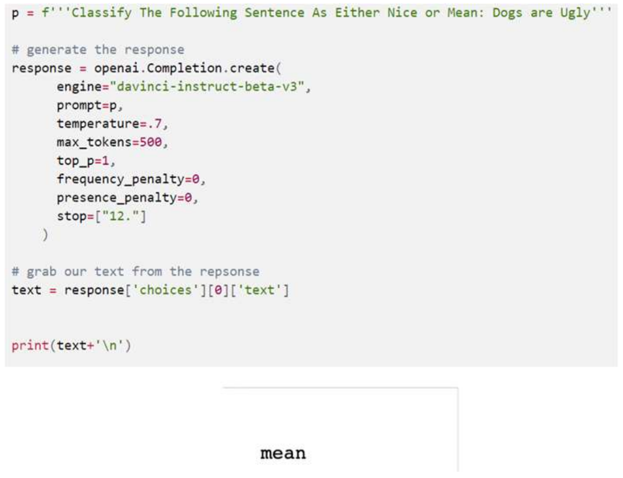
Translation



```python
p = f'''Transform this sentence from korean to english: 배 안 고파요'''

# generate the response
response = openai.Completion.create(
      engine="davinci-instruct-beta-v3",
      prompt=p,
      temperature=.7,
      max_tokens=500,
      top_p=1,
      frequency_penalty=0,
      presence_penalty=0,
      stop=["12."]
    )

# grab our text from the repsonse
text = response['choices'][0]['text']


print(text+'\n')
```

```
I am hungry.
```

Classification



```python
p = f'''Classify The Following Sentence As Either Nice or Mean: Dogs are Ugly'''

# generate the response
response = openai.Completion.create(
      engine="davinci-instruct-beta-v3",
      prompt=p,
      temperature=.7,
      max_tokens=500,
      top_p=1,
      frequency_penalty=0,
      presence_penalty=0,
      stop=["12."]
    )

# grab our text from the repsonse
text = response['choices'][0]['text']


print(text+'\n')
```

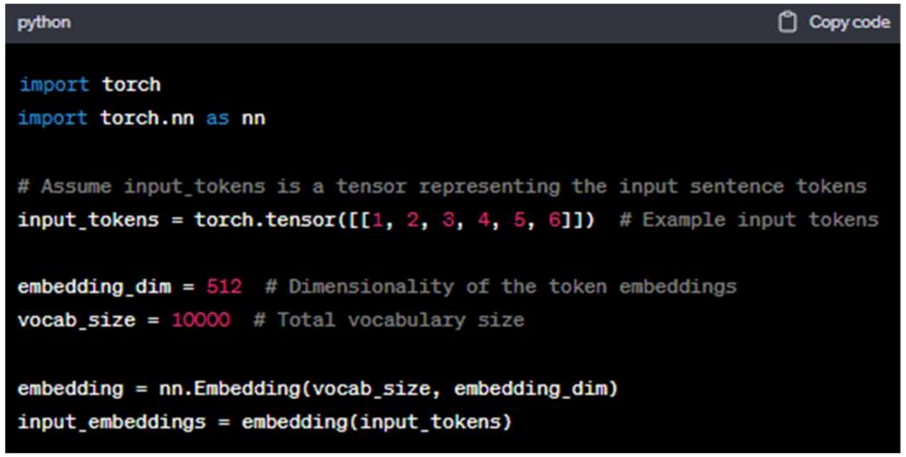
```
mean
```

[1]Radford et al., (2018)

Architecture implemented in Code

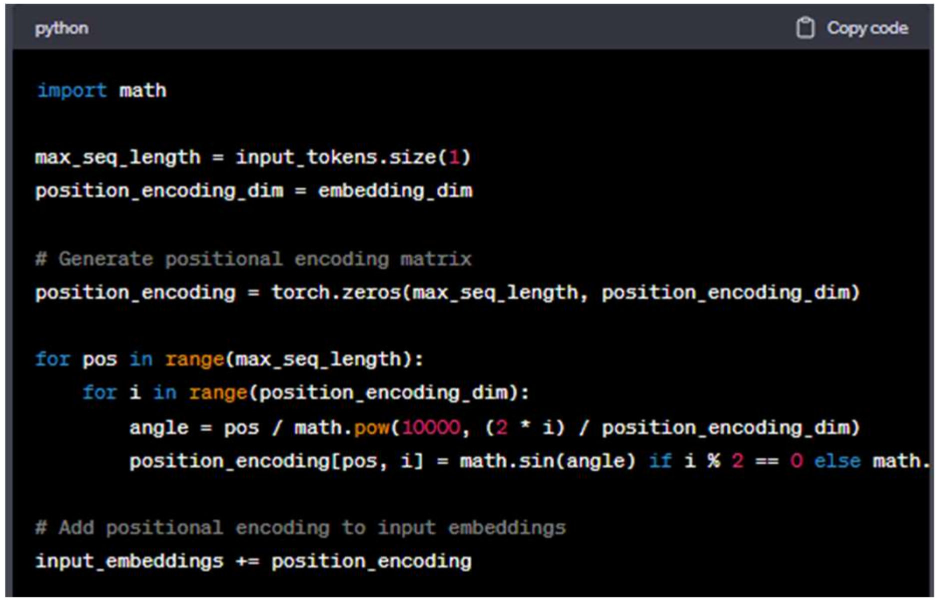
- https://www.youtube.com/watch?v=kCc8FmEb1nY&t=1333s

Input Embedding - Code according to ChatGPT



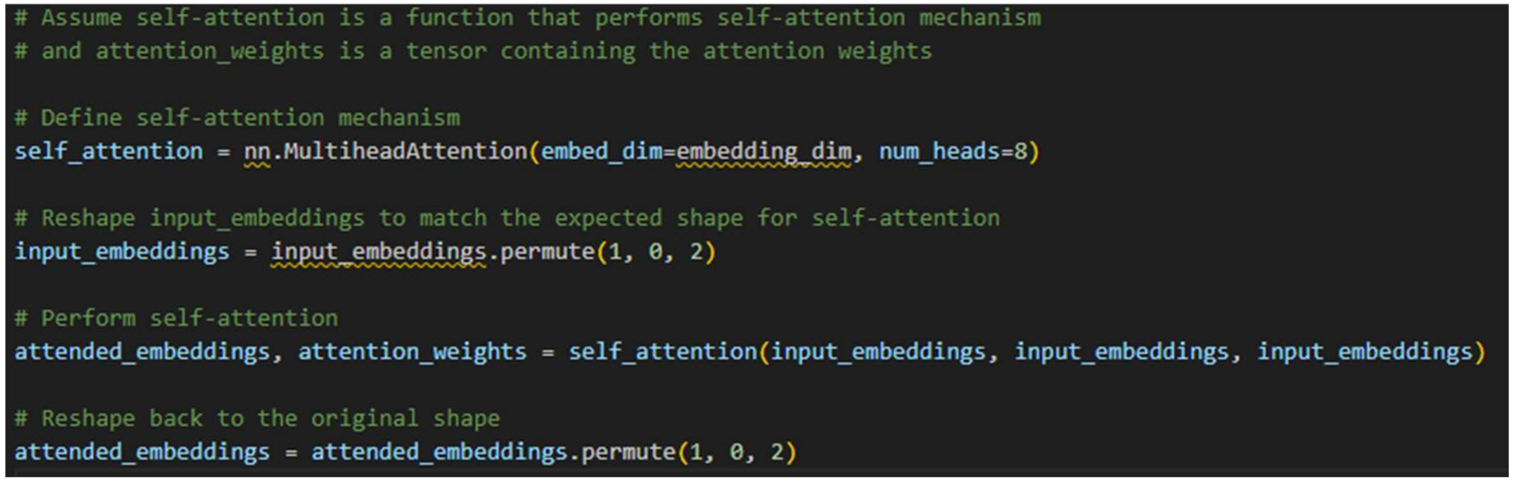
```python
import torch
import torch.nn as nn

# Assume input_tokens is a tensor representing the input sentence tokens
input_tokens = torch.tensor([[1, 2, 3, 4, 5, 6]])  # Example input tokens

embedding_dim = 512  # Dimensionality of the token embeddings
vocab_size = 10000  # Total vocabulary size

embedding = nn.Embedding(vocab_size, embedding_dim)
input_embeddings = embedding(input_tokens)
```

Positional Encoding



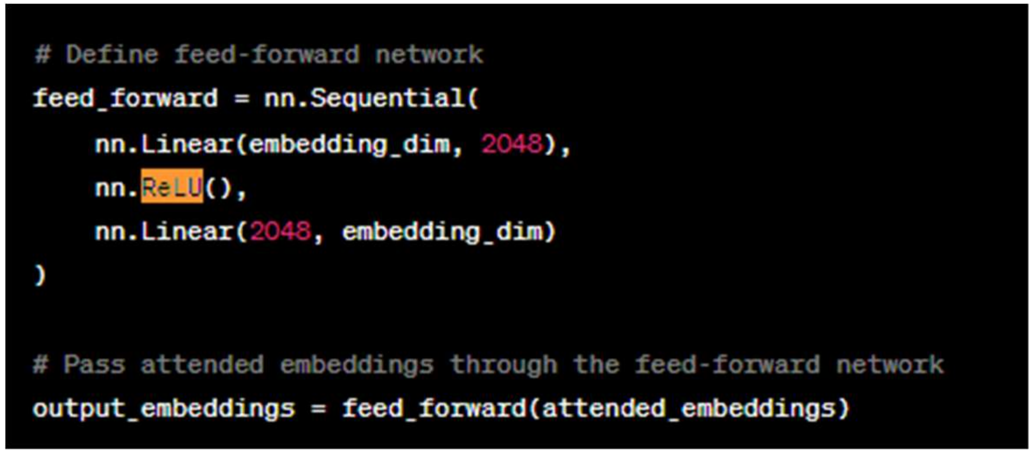
```python
import math

max_seq_length = input_tokens.size(1)
position_encoding_dim = embedding_dim

# Generate positional encoding matrix
position_encoding = torch.zeros(max_seq_length, position_encoding_dim)

for pos in range(max_seq_length):
    for i in range(position_encoding_dim):
        angle = pos / math.pow(10000, (2 * i) / position_encoding_dim)
        position_encoding[pos, i] = math.sin(angle) if i % 2 == 0 else math.

# Add positional encoding to input embeddings
input_embeddings += position_encoding
```

Self attention



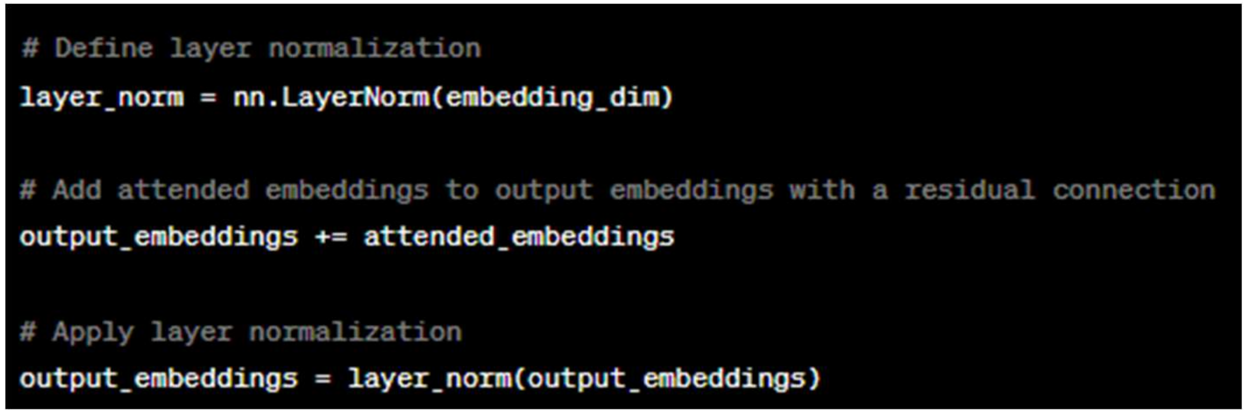
```
# Assume self-attention is a function that performs self-attention mechanism
# and attention_weights is a tensor containing the attention weights

# Define self-attention mechanism
self_attention = nn.MultiheadAttention(embed_dim=embedding_dim, num_heads=8)

# Reshape input_embeddings to match the expected shape for self-attention
input_embeddings = input_embeddings.permute(1, 0, 2)

# Perform self-attention
attended_embeddings, attention_weights = self_attention(input_embeddings, input_embeddings, input_embeddings)

# Reshape back to the original shape
attended_embeddings = attended_embeddings.permute(1, 0, 2)
```

Feed forward network



```
# Define feed-forward network
feed_forward = nn.Sequential(
    nn.Linear(embedding_dim, 2048),
    nn.ReLU(),
    nn.Linear(2048, embedding_dim)
)

# Pass attended embeddings through the feed-forward network
output_embeddings = feed_forward(attended_embeddings)
```

Residual connection and Layer normalization



```
# Define layer normalization
layer_norm = nn.LayerNorm(embedding_dim)

# Add attended embeddings to output embeddings with a residual connection
output_embeddings += attended_embeddings

# Apply layer normalization
output_embeddings = layer_norm(output_embeddings)
```

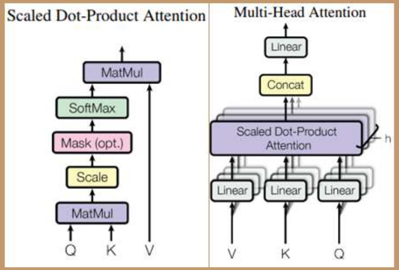
Main mathematical functions[1]

- Positional encoding using sine and cosine functions
  - Keep same dimensions as the embeddings so they can be added up

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

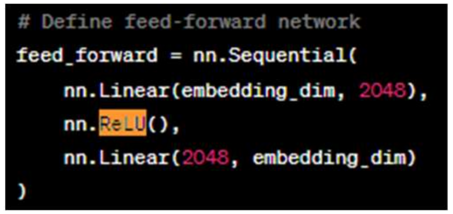
- Scaled Dot-Product Attention
  - Computes the attention scores between the vectors of Query, Key and Value
  - Taking the dot product of the query with all keys, divide each by the dimension of queries and keys and apply softmax to obtain the weights on the values
  - $\sqrt{d_k}$ → square root of dimension of keys

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$



- Feed Forward Network

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$



```
# Define feed-forward network
feed_forward = nn.Sequential(
    nn.Linear(embedding_dim, 2048),
    nn.ReLU(),
    nn.Linear(2048, embedding_dim)
)
```

[1]Vaswani et al., (2017)