



GenAI  
Tutorial

2025

# LLMs From Scratch

Forough Shirin Abkenar

# Contents

<b>1</b>	<b>Overview</b>	<b>1</b>
<b>2</b>	<b>LLMs Architecture</b>	<b>2</b>
<b>3</b>	<b>Encoder-only Models</b>	<b>3</b>
3.1	Input Embedding . . . . .	3
3.2	Positional Encoding . . . . .	4
3.3	Encoder Layer . . . . .	5
3.3.1	Multi-Head Self-Attention Mechanism . . . . .	5
3.3.2	Add & Norm . . . . .	6
3.3.3	Feed Forward Network . . . . .	6
3.4	Implementation . . . . .	7
3.4.1	Evaluation . . . . .	8
<b>4</b>	<b>Decoder-only Models</b>	<b>9</b>
4.1	Input Embedding . . . . .	9
4.2	Positional Encoding . . . . .	11
4.3	Decoder Layer . . . . .	11
4.3.1	Multi-Head Self-Attention Mechanism . . . . .	11
4.3.2	Add & Norm . . . . .	12
4.3.3	Feed Forward Network . . . . .	13
4.4	Implementation . . . . .	13
4.4.1	Evaluation . . . . .	15
<b>5</b>	<b>LLM Fine-tuning</b>	<b>17</b>
<b>6</b>	<b>PEFT</b>	<b>18</b>
<b>7</b>	<b>RLHF</b>	<b>19</b>
<b>8</b>	<b>RAG</b>	<b>20</b>
<b>9</b>	<b>Agentic AI</b>	<b>21</b>
<b>10</b>	<b>References</b>	<b>22</b>

# List of Figures

2.1	Architecture of an LLM . . . . .	2
3.1	Encoder-only mode architecture . . . . .	3
3.2	Architecture of an attention head . . . . .	5
4.1	Decoder-only mode architecture . . . . .	9
4.2	Architecture of an attention head . . . . .	12

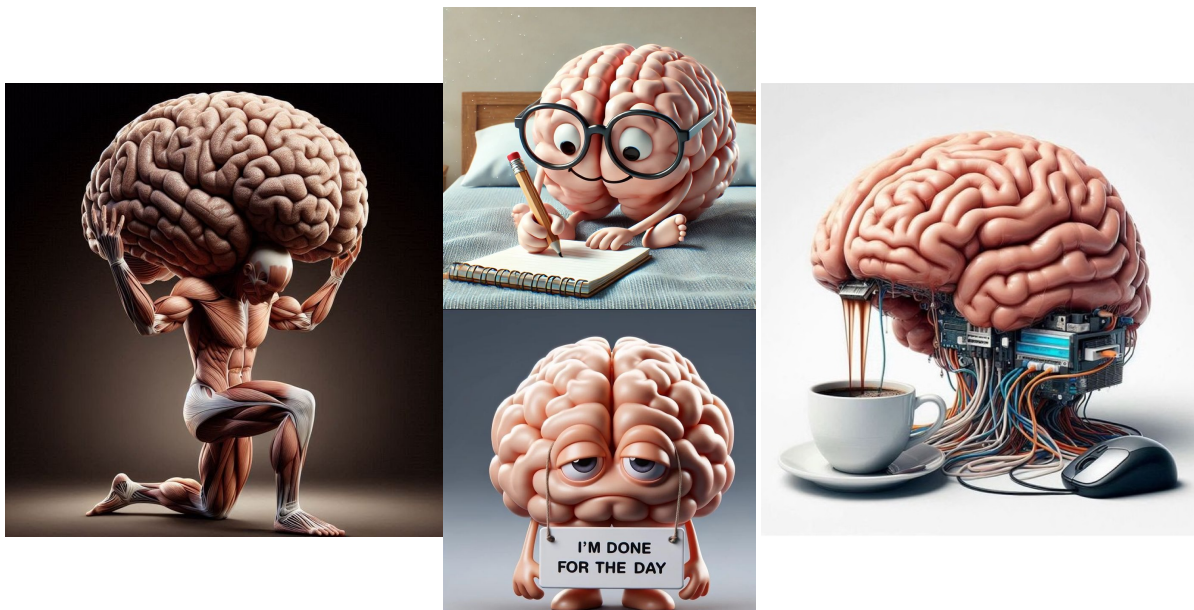
# List of Tables

3.1	Performance evaluation of tinyBERT . . . . .	8
4.1	Performance evaluation of TinyGPT . . . . .	16

# Overview

The current document studies fundamental topics related to generative artificial intelligence (AI) and large language models (LLMs) [1], such as model architectures and optimizations, fine-tuning, and agentic systems, and the corresponding codes are implemented in Python and PyTorch. For implementation purposes, we also use defined tokenizers, models, and agents in HuggingFace [2] and LangChain [3]. It is worth noting

that the cliparts used in this document were downloaded from Pinterest [4].



# LLMs Architecture

Transformers form the foundation of large language model (LLM) architectures. The original LLM architecture consists of two main components, named as an encoder and a decoder (see Fig. 2.1) [1]. Models that follow this design are referred to as *encoder-decoder models*. In these models, the encoder embeds the input, and the resulting representation is passed to the decoder for further processing. Building on the capabilities of these two components, two additional LLM architectures were later introduced, namely *encoder-only* and *decoder-only* models. The following two sections review these architectures in detail. It is worth mentioning that the encoder-decoder architecture is beyond the scope of the current document.

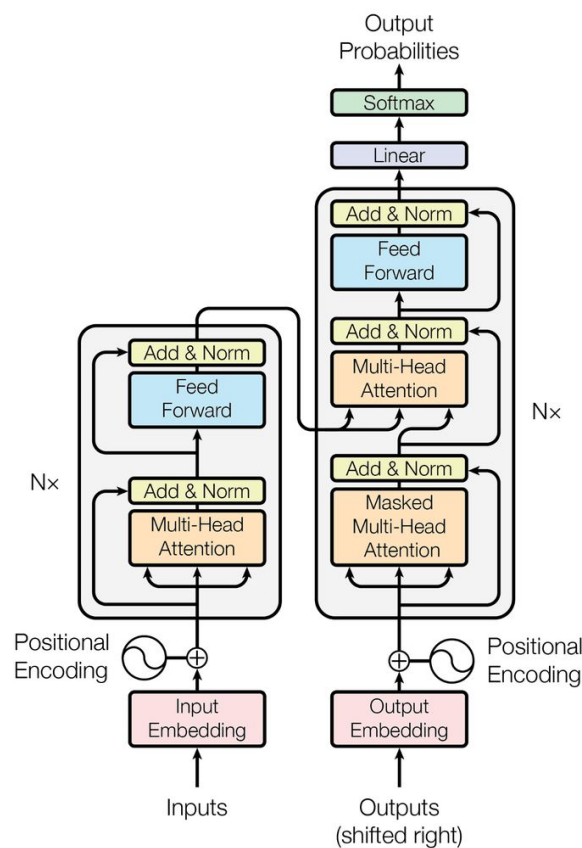


Figure 2.1: Architecture of an LLM including an encoder and a decoder [1].



# Encoder-only Models

The encoder-only models in LLMs own the LLM architecture with only the encoder transformer (see Fig. 3.1). The encoder in the LLM architecture is responsible for receiving the input data (prompts) and embedding (encoding) them into meaningful output (vector representation). Encoder-only models exploit bidirectional processing of data whereby the input tokens are processed using information from both left and right to understand the token's context.

Bidirectional encoder representations from transformers (BERT) [5] and robustly optimized BERT pretraining approach (RoBERTa) [6] models are examples of encoder-only models that are applicable for text classification, sentiment analysis, named entity recognition (NER), and etc.

As seen in Fig. 3.1, the inputs in the encoder-only model are passed through sequential components, i.e., input embedding, positional encoding, and encoder layer (shown as Nx in the figure). In the rest of the current chapter, we review each layer and the corresponding mechanisms.

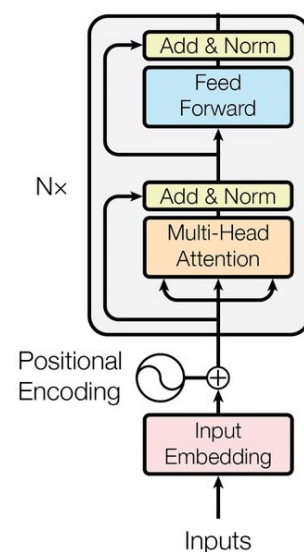


Figure 3.1: Encoder-only mode architecture [1].

## 3.1 Input Embedding

In encoder-only models, the inputs are textual data, represented as sequences of text units such as words, subwords, or characters. However, the encoder block processes numerical representations rather than raw text. To bridge this gap, an *input embedding* layer converts textual inputs into numerical IDs. This process relies on a predefined *vocabulary* in which each text unit, commonly referred to as a *token*, is mapped to a unique identifier, namely token ID. Notably, each token ID is a vector of numbers with a shape of pre-defined embedding dimension. Consequently, the input text undergoes *tokenization*, where it is divided into tokens, and then each token is mapped to the corresponding token ID.

For instance, in a Python programming, we define the input as

```
# Input
text = """
Jalāl al-Dīn Muḥammad Rūmī, or simply Rumi, was a 13th-century poet, Hanafi
faqih, Maturidi theologian, and Sufi mystic born during the Khwarazmian Empire.
"""
```

In the next step, we need to define the tokens unit. Considering the word unit as the tokens, we have

```
# Data preparation for the vocabulary

# remove newlines
text = text.replace('\n', ' ')

# convert text to characters
words = text.split()
print(f"Words: {words}")

# size of vocabulary
vocab = list(set(words))
vocab_size = len(vocab)

Words: ['Jalāl', 'al-Dīn', 'Muḥammad', 'Rūmī,', 'or', 'simply', 'Rumi,', 'was', 'a', '13th-century', 'poet,', 'Hanafi', 'faqih,', 'Maturidi', 'theologian,', 'and', 'Sufi', 'mystic', 'born', 'during', 'the', 'Khwarazmian', 'Empire.']
```

Finally, the tokens are converted into token IDs. To achieve this, the vocabulary (stoi in the code) is defined, and the *encode* function is applied to the input text.

```
# Encoding

# string to integer
stoi = {c: i for i, c in enumerate(vocab)}
results = []
for k, v in stoi.items():
    results.append(f"{k}: {v}")
print(results)

# define the encoder
encode = lambda s: torch.tensor([stoi[c] for c in s.split()], dtype=torch.long)

encoded_text = encode(text)
print(f"\nEncoded text: {encoded_text}")

['poet,: 0', 'al-Dīn: 1', 'faqih,: 2', 'and: 3', 'Muḥammad: 4', 'born: 5', 'Hanafi: 6', 'was: 7', 'Rūmī,: 8', 'Rumi,: 9', 'or: 10', 'simply: 11', 'a: 12', 'Khwarazmian: 13', 'Sufi: 14', 'mystic: 15', 'theologian,: 16', 'Maturidi: 17', 'the: 18', 'Jalāl: 19', '13th-centur y: 20', 'Empire.: 21', 'during: 22']

Encoded text: tensor([19, 1, 4, 8, 10, 11, 9, 7, 12, 20, 0, 6, 2, 17, 16, 3, 14, 15, 5, 22, 18, 13, 21])
```

## 3.2 Positional Encoding

Positional encoding is a mechanism whereby the information about the position of a token is injected to the input data. Hence, the model can learn the meaning and importance of the corresponding token w.r.t. its position in the input (subject, object, verb, adjective, etc.). The traditional positional encoding is calculated using Eq. 3.1, where  $pos$ ,  $i$ , and  $d_{\text{model}}$  are position, the index for the dimension, and the embedding dimension [1].

$$\begin{aligned} PE_{(pos,i)} &= \sin\left(pos/10000^{2i/d_{\text{model}}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(pos/10000^{2i/d_{\text{model}}}\right) \end{aligned} \quad (3.1)$$

It is worth noting that positional encoding can also be learned during the training of the encoder. For example, in the following code snippet, the positional encoding is implemented as a dedicated layer within the model. During the forward pass, the positional information is integrated into the input representations by adding it to the token embeddings.

```

class TinyBERT(nn.Module):
    def __init__(self, vocab_size, block_size, n_embed=128, n_head=4, n_layer=4, dropout=0.1):
        super(TinyBERT, self).__init__()
        self.token_embed = nn.Embedding(vocab_size, n_embed)
        self.pos_embed = nn.Embedding(block_size, n_embed)
        self.blocks = nn.Sequential(
            *[Block(n_embed, n_head, dropout) for _ in range(n_layer)]
        )
        self.ln_f = nn.LayerNorm(n_embed)
        self.mlm_head = nn.Linear(n_embed, vocab_size)

    def forward(self, idx, labels=None):
        B, T = idx.shape
        token = self.token_embed(idx)
        pos = self.pos_embed(torch.arange(T, device=idx.device))
        x = token + pos
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.mlm_head(x)
        loss = None
        if labels is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), labels.view(-1), ignore_index=-100)

        return logits, loss

```

### 3.3 Encoder Layer

The encoder layer (shown  $N \times$  in Fig. 3.1) comprises two sub-layers. The first sub-layer implements the multi-head self-attention mechanism, and the second sub-layer is a fully-connected feed forward neural network. Each sub-layer has a residual connection around itself, and also is succeeded by a normalization layer [1].

#### 3.3.1 Multi-Head Self-Attention Mechanism

The self-attention mechanism is a core component of transformers, enabling the model to learn and capture the relationships between tokens within a sequence. This mechanism is implemented within the attention heads of the transformer architecture [1].

To model the relationships between tokens, the input is first projected into three distinct representations: the *query* (Q), *key* (K), and *value* (V) vectors. Figure 3.2 illustrates the architecture of an attention head within the model. When an embedded and positionally encoded input passes through the attention head, it undergoes the following five processing steps [1]:

- **Step 1:** query (Q), key (K), and value (V) components are passed through linear layers in the attention head model so that these components are learned.
- **Step 2:** the alignment scores are calculated through multiplication of matrices query and key.
- **Step 3:** the alignment scores are scaled by  $1/d_k$ , where  $d_k$  is the dimension of the query (or the key) matrix.
- **Step 4:** *softmax* operation is applied to the scaled scores to obtain the attention weights.
- **Step 5:** the attention weights are multiplied with the value matrix.

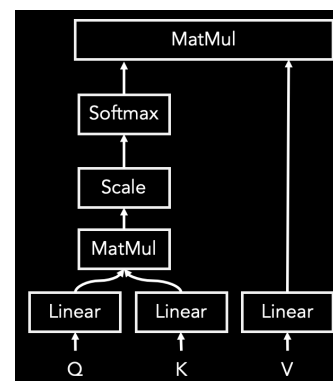


Figure 3.2: Architecture of an attention head [1].

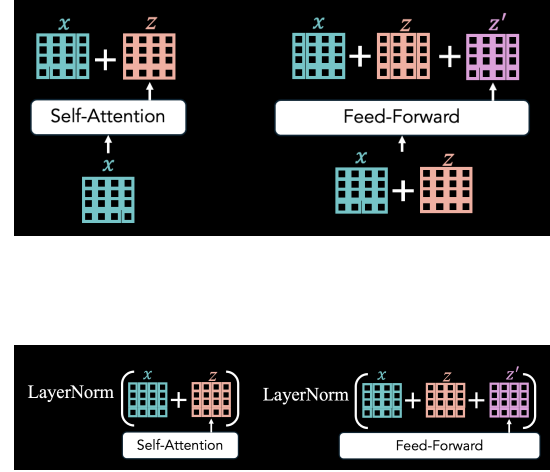


The multi-head self-attention mechanism consists of multiple parallel attention heads, each learning distinct representation subspaces. The outputs from all attention heads are concatenated and then projected through a linear layer to produce the final combined representation [1].

### 3.3.2 Add & Norm

**Residual Connections (Add):** Preserving information from earlier layers helps mitigate the vanishing gradient problem. In transformer encoders, this is achieved through *residual connections*, where the original input of a layer is added to its output. This mechanism enables the network to retain essential information across layers and facilitates more effective gradient flow during training [1].

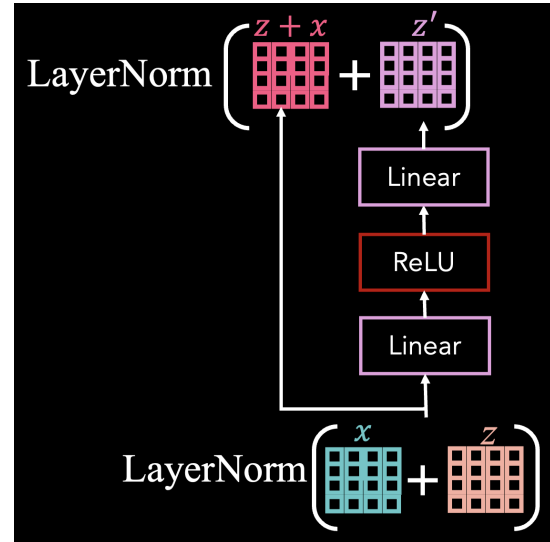
**Layer Normalization (Norm):** During training, the model may experience *internal covariate shift*, where the distribution of activations changes across layers, potentially leading to vanishing or exploding gradients. To address this challenge, *layer normalization* is applied to the outputs of deeper layers that in result, stabilizes training by reducing distributional shifts and ensures more consistent gradient flow [1].



### 3.3.3 Feed Forward Network

The feed-forward sub-layer introduces nonlinearities into the encoder, thereby enhancing the model's capacity to capture complex patterns and non-linear relationships within sequential data. The feed forward network consists of two linear transformations separated by an activation function, typically the rectified linear unit (ReLU) or the Gaussian error linear unit (GELU) [1].

The ReLU activation applies  $\max(0, x)$  to each input  $x$ , effectively setting all negative values to zero. Although computationally efficient, ReLU suffers from the *dying ReLU* problem, where neurons can become permanently inactive during training and consistently output zero due to negative weighted inputs. In contrast, GELU is a smoother activation function that maps a value to  $x \times \Phi(x)$ , where  $\Phi(x)$  denotes the cumulative distribution function (CDF) of the standard normal distribution. This smooth approximation allows GELU to retain small negative values and has been shown to improve performance in transformer-based architectures [1].



### 3.4 Implementation

In this section, we use a small input text to develop a minimal encoder-only model, called **TinyBert**. The goal is to gain familiarity with the operation of encoder-only models. The overall network architecture is shown below, and the complete implementation script is available on [GitHub](#).

```
class TinyBERT(nn.Module):
    def __init__(self, vocab_size, block_size, n_embed=128, n_head=4, n_layer=4, dropout=0.1):
        super(TinyBERT, self).__init__()
        self.token_embed = nn.Embedding(vocab_size, n_embed)
        self.pos_embed = nn.Embedding(block_size, n_embed)
        self.blocks = nn.Sequential(
            *[Block(n_embed, n_head, dropout) for _ in range(n_layer)]
        )
        self.ln_f = nn.LayerNorm(n_embed)
        self.mlm_head = nn.Linear(n_embed, vocab_size)

    def forward(self, idx, labels=None):
        B, T = idx.shape
        token = self.token_embed(idx)
        pos = self.pos_embed(torch.arange(T, device=idx.device))
        x = token + pos
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.mlm_head(x)
        loss = None
        if labels is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), labels.view(-1), ignore_index=-100)

        return logits, loss
```

As defined in the `__init__` function, the model network includes the following layers:

- `self.token_embed`: it creates an embedding layer that converts token indices (integers) into dense vector representations (embeddings).
- `self.pos_embed`: this layer create the positional encoding of the input tokens.
- `self.blocks`: it includes encoder blocks (layers), each with a multi-head self-attention head, a feed forward network, and the corresponding add & norm components.
- `self.ln_f`: this layer defines the final layer normalization applied to the transformer's output before passing it into the language modeling head.
- `self.mlm_head`: it defines the masked language model (MLM) head, i.e., the final layer that maps the hidden representations produced by the transformer into predicted token probabilities.

When an input passes through the *forward* function, it is first converted into token embeddings. Positional embeddings are then added to incorporate information about token order. The resulting representations are sequentially passed through all the transformer blocks defined in the model. The output of the final block is normalized using the last layer normalization and then fed into the language modeling head. At this stage, each token is represented by a hidden vector of size equal to the embedding dimension, and the final layer predicts the probability distribution over the vocabulary for the next token.

```
class Head(nn.Module):
    def __init__(self, n_embed, head_size, dropout):
        super(Head, self).__init__()
        self.key = nn.Linear(n_embed, head_size, bias=False)
        self.query = nn.Linear(n_embed, head_size, bias=False)
        self.value = nn.Linear(n_embed, head_size, bias=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B, T, C = x.shape
        k, q, v = self.key(x), self.query(x), self.value(x)
        att = (q @ k.transpose(-2, -1)) / math.sqrt(k.size(-1))
        att = F.softmax(att, dim=-1)
        att = self.dropout(att)
        out = att @ v

        return out

class MultiHead(nn.Module):
    def __init__(self, n_embed, n_head, dropout):
        super(MultiHead, self).__init__()
        self.head_size = n_embed // n_head
        self.heads = nn.ModuleList(
            [Head(n_embed, self.head_size, dropout) for _ in range(n_head)]
        )
        self.proj = nn.Linear(n_embed, n_embed)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        proj = self.proj(out)
        return self.dropout(proj)

class FeedForward(nn.Module):
    def __init__(self, n_embed, dropout):
        super(FeedForward, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embed, 4*n_embed),
            nn.GELU(),
            nn.Linear(4*n_embed, n_embed),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        return self.net(x)

class Block(nn.Module):
    def __init__(self, n_embed, n_head, dropout):
        super(Block, self).__init__()
        self.mh = MultiHead(n_embed, n_head, dropout)
        self.ff = FeedForward(n_embed, dropout)
        self.ln1 = nn.LayerNorm(n_embed)
        self.ln2 = nn.LayerNorm(n_embed)

    def forward(self, x):
        x_p = self.ln1(x)
        x = x + self.mh(x_p)
        x_p = self.ln2(x)
        x = x + self.ff(x_p)

        return x
```

During training, the model compares these predicted logits with the true token IDs using cross-entropy loss, and updates its weights through backpropagation to minimize this loss.

```
num_epochs = 1000
for epoch in range(num_epochs):
    x_batch, y_batch = get_batch(mask_token_id, vocab_size, batch_size=BATCH_SIZE, block_size=BLOCK_SIZE)
    logits, loss = model(x_batch, y_batch)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    mask = y_batch != -100
    preds = torch.argmax(logits, dim=-1)
    correct = (preds[mask] == y_batch[mask]).sum().item()
    acc = correct / mask.sum().item()

    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item(): .4f}, Accuracy: {acc * 100: .2f}")

Epoch 100/1000, Loss: 4.8240, Accuracy: 5.88
Epoch 200/1000, Loss: 3.9187, Accuracy: 16.67
Epoch 300/1000, Loss: 3.5331, Accuracy: 30.43
Epoch 400/1000, Loss: 2.6440, Accuracy: 46.81
Epoch 500/1000, Loss: 2.0034, Accuracy: 70.00
Epoch 600/1000, Loss: 1.5215, Accuracy: 65.96
Epoch 700/1000, Loss: 1.3033, Accuracy: 74.00
Epoch 800/1000, Loss: 1.0863, Accuracy: 86.67
Epoch 900/1000, Loss: 0.6710, Accuracy: 91.89
Epoch 1000/1000, Loss: 0.6087, Accuracy: 95.83
```

Since the primary objective of the designed TinyBert model is to predict masked tokens in the input text, the `get_batch` function used during training incorporates a `mask_token` function based on the MLM strategy. This function applies an 80/10/10 masking rule: 80% of tokens are replaced with a predefined mask token (e.g., “[MASK]”), 10% are substituted with random tokens from the vocabulary, and the remaining 10% are left unchanged.

```
data = encode(text)
def get_batch(mask_token_id, vocabsize, batch_size=32, block_size=8):
    ix = torch.randint(0, len(data) - block_size - 1, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix]) # (batch_size, block_size)
    y = torch.stack([data[i+1:i+1+block_size] for i in ix]) # (batch_size, block_size)
    x, y = mask_tokens(x.clone(), vocabsize, mask_token_id)
    return x.to(DEVICE), y.to(DEVICE)
```

### 3.4.1 Evaluation

For evaluation, we select a portion of the text, mask certain words, and assess the model’s performance in predicting these masked tokens. Accordingly, we compute the cross-entropy loss, perplexity, and accuracy on the masked words: (1) cross-entropy loss measures the difference between the predicted probability distribution of a model and the true distribution of the target data; lower values indicates better predictions and less uncertainty. (2) perplexity measures the model’s uncertainty; lower perplexity implies that the model assigns higher probability to the actual next word in the sequence, resulting a more confident and accurate model. (3) accuracy measures how well the model predicts the masked tokens, with higher values indicating more correct predictions. Table 3.1 indicates the performance of TinyBert w.r.t. the aforementioned evaluation metrics.

Table 3.1: Performance evaluation of tinyBERT

Cross Entropy Loss	Perplexity	Accuracy (%)
0.2245	1.25	100

# Decoder-only Models

Decoder-only models in LLMs consist solely of the transformer decoder component of the overall architecture (see Fig. 4.1). The decoder receives input data (prompts) and generates coherent and context-aware output. Unlike encoder-only models, which leverage bidirectional context, decoder-only models are **autoregressive** whereby they predict the next token based on the previously generated tokens. Therefore, these models are particularly well-suited for text generation tasks.

Generative pre-trained transformer (GPT) family (e.g., GPT-2, GPT-3, and ChatGPT), pathways language model (PaLM) [7], and large language model Meta AI (LLaMA) [8] models are examples of decoder-only models, commonly used for text generation tasks such as creative writing and conversational agents.

As shown in Fig. 4.1, similar to encoder-only models, the inputs in a decoder-only model are processed through a series of sequential components, including input embedding, positional encoding, and decoder blocks/layers (denoted as Nx in the figure). In the remainder of this chapter, we review each layer and its underlying mechanisms in detail.

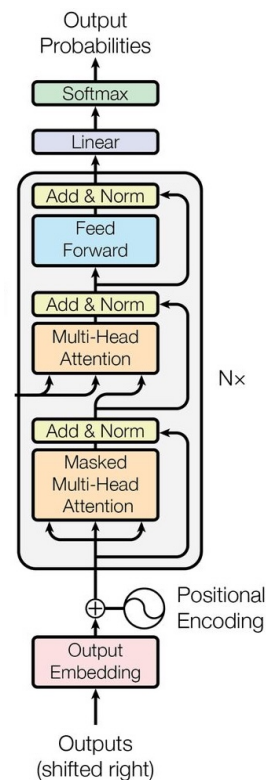


Figure 4.1: Decoder-only mode architecture [1].

## 4.1 Input Embedding

In decoder-only models, the inputs are textual data, represented as sequences of text units such as words, subwords, or characters. However, the decoder block processes numerical representations rather than raw text. To bridge this gap, an *input embedding* layer converts textual inputs into numerical IDs. This process relies on a predefined *vocabulary* in which each text unit, commonly referred to as a *token*, is mapped to a unique identifier, namely token ID. Notably, each token ID is a vector of numbers with a shape of pre-defined embedding dimension. Consequently, the input text undergoes *tokenization*, where it is divided into tokens, and then each token is mapped to the corresponding token ID. For instance, in a Python programming, we define the input as

```
text = """
Jalāl al-Dīn Muḥammad Rūmī, or simply Rumi, was a 13th-century poet, Hanafi
faqih, Maturidi theologian, and Sufi mystic born during the Khwarazmian Empire.
Rumi's works are written in his mother tongue, Persian. He occasionally used the
Arabic language and single Turkish and Greek words in his verse."""
```

In the next step, we need to define the tokens unit. Considering the character unit as the tokens, we have

```
# Data preparation for the vocabulary

# convert text to characters
chars = list(set(text))
```

Finally, the tokens are converted into token IDs. To achieve this, the vocabulary (stoi in the code) is defined, and the *encode* function is applied to the input text.

```
# Encoding

# string to integer
stoi = {c: i for i, c in enumerate(chars)}
results = []
for k, v in stoi.items():
    results.append(f'{k}: {v}')
print(results)

# encode
encode = lambda s: torch.tensor([stoi[c] for c in s], dtype=torch.long)

encoded_text = encode(text)
print(f'\nEncoded text: {encoded_text}')

['z: 0', 'd: 1', 'i: 2', '\n: 3', 'u: 4', 'q: 5', ',: 6', 'k: 7', 'o: 8', '": 9', 'h: 10', 'h: 11', 'P: 12', 'm: 13', ' ': 14, 'ā: 15',
'b: 16', 'l: 17', 'i: 18', 'M: 19', 'R: 20', 'n: 21', 'E: 22', 's: 23', 'a: 24', '–: 25', 'w: 26', 'l: 27', 'p: 28', '.: 29', 'f: 30',
'ü: 31', 'K: 32', 't: 33', 'y: 34', 'A: 35', 'H: 36', 'T: 37', 'e: 38', 'r: 39', 'v: 40', 'c: 41', 'G: 42', '3: 43', 'J: 44', 'D: 45',
'g: 46', 'S: 47']

Encoded text: tensor([ 3, 44, 24, 17, 15, 17, 14, 24, 17, 25, 45,  2, 21, 14, 19,  4, 10, 24,
13, 13, 24,  1, 14, 20, 31, 13,  2,  6, 14,  8, 39, 14, 23, 18, 13, 28,
17, 34, 14, 20,  4, 13, 18,  6, 14, 26, 24, 23, 14, 24, 14, 27, 43, 33,
11, 25, 41, 38, 21, 33,  4, 39, 34, 14, 28,  8, 38, 33,  6, 14, 36, 24,
21, 24, 30, 18, 14,  3, 30, 24,  5, 18, 11,  6, 14, 19, 24, 33,  4, 39,
18,  1, 18, 14, 33, 11, 38,  8, 17,  8, 46, 18, 24, 21,  6, 14, 24, 21,
 1, 14, 47,  4, 30, 18, 14, 13, 34, 23, 33, 18, 41, 14, 16,  8, 39, 21,
14,  1,  4, 39, 18, 21, 46, 14, 33, 11, 38, 14, 32, 11, 26, 24, 39, 24,
 0, 13, 18, 24, 21, 14, 22, 13, 28, 18, 39, 38, 29, 14,  3, 20,  4, 13,
18,  9, 23, 14, 26,  8, 39,  7, 23, 14, 24, 39, 38, 14, 26, 39, 18, 33,
33, 38, 21, 14, 18, 21, 14, 11, 18, 23, 14, 13,  8, 33, 11, 38, 39, 14,
33,  8, 21, 46,  4, 38,  6, 14, 12, 38, 39, 23, 18, 24, 21, 29, 14, 36,
38, 14,  8, 41, 41, 24, 23, 18,  8, 21, 24, 17, 17, 34, 14,  4, 23, 38,
 1, 14, 33, 11, 38, 14,  3, 35, 39, 24, 16, 18, 41, 14, 17, 24, 21, 46,
 4, 24, 46, 38, 14, 24, 21,  1, 14, 23, 18, 21, 46, 17, 38, 14, 37,  4,
39,  7, 18, 23, 11, 14, 24, 21,  1, 14, 42, 39, 38, 38,  7, 14, 26,  8,
39,  1, 23, 14, 18, 21, 14, 11, 18, 23, 14, 40, 38, 39, 23, 38, 29])
```

Moreover, a *decode* function is defined to convert tokens IDs to tokens. To this end, we defined the corresponding dictionary (itos in the code), and the *decode* function is applied to the encoded text.

```
# Decoding

# integer to string
itos = {i: c for c, i in stoi.items()}

# decode
decode = lambda t: ''.join(itos[int(i)] for i in t)

decoded_text = decode(encoded_text)
print(f'Decoded IDs: {decoded_text}')
```

Decoded IDs:  
Jalāl al-Dīn Muḥammad Rūmī, or simply Rumi, was a 13th-century poet, Hanafi faqih, Maturidi theologian, and Sufi mystic born during the Khwarazmian Empire. Rumi's works are written in his mother tongue, Persian. He occasionally used the Arabic language and single Turkish and Greek words in his verse.

## 4.2 Positional Encoding

Positional encoding is a mechanism whereby the information about the position of a token is injected to the input data. Hence, the model can learn the meaning and importance of the corresponding token w.r.t. its position in the input (subject, object, verb, adjective, etc.). The traditional positional encoding is calculated using Eq. 4.1, where  $pos$ ,  $i$ , and  $d_{\text{model}}$  are position, the index for the dimension, and the embedding dimension [1].

$$\begin{aligned} PE_{(pos,i)} &= \sin\left(pos/10000^{2i/d_{\text{model}}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(pos/10000^{2i/d_{\text{model}}}\right) \end{aligned} \quad (4.1)$$

It is worth noting that positional encoding can also be learned during the training of the encoder. For example, in the following code snippet, the positional encoding is implemented as a dedicated layer within the model. During the forward pass, the positional information is integrated into the input representations by adding it to the token embeddings.

```
class TinyGPT(nn.Module):
    def __init__(self, vocab_size, block_size, n_embed=128, n_head=4, n_layer=4, dropout=0.1):
        super(TinyGPT, self).__init__()
        self.token_embed = nn.Embedding(vocab_size, n_embed)
        self.pos_embed = nn.Embedding(block_size, n_embed)
        self.blocks = nn.Sequential(
            *[Block(n_embed, n_head, block_size, dropout) for _ in range(n_layer)]
        )
        self.ln_f = nn.LayerNorm(n_embed)
        self.lm_head = nn.Linear(n_embed, vocab_size)

    def forward(self, idx, targets=None):
        B, T = idx.shape
        token = self.token_embed(idx)
        pos = self.pos_embed(torch.arange(T, device=idx.device))
        x = token + pos
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.lm_head(x)
        loss = None
        if targets is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1))

        return logits, loss
```

## 4.3 Decoder Layer

The decoder layer (shown as  $N_x$  in Fig. 4.1) consists of three sub-layers: (1) a multi-head self-attention mechanism applied to the decoder's inputs, (2) a multi-head cross-attention mechanism over the encoder's outputs, and (3) a fully connected feed forward network. Each sub-layer includes a residual connection and is followed by a normalization layer [1].

### 4.3.1 Multi-Head Self-Attention Mechanism

The self-attention mechanism is a core component of transformers, enabling the model to learn and capture the relationships between tokens within a sequence. This mechanism is implemented within the attention heads of the transformer architecture [1].



To model the relationships between tokens, the input is first projected into three distinct representations: the *query* (Q), *key* (K), and *value* (V) vectors. Figure 4.2 illustrates the architecture of an attention head within the model. When an embedded and positionally encoded input passes through the attention head, it undergoes the following six processing steps [1]:

- **Step 1:** query (Q), key (K), and value (V) components are passed through linear layers in the attention head model so that these components are learned.
- **Step 2:** the alignment scores are calculated through multiplication of matrices query and key.
- **Step 3:** the alignment scores are scaled by  $1/d_k$ , where  $d_k$  is the dimension of the query (or the key) matrix.
- **Step 4:** a causal masking function is applied to prevent the model from attending to future tokens.
- **Step 5:** *softmax* operation is applied to the scaled scores to obtain the attention weights.
- **Step 6:** the attention weights are multiplied with the value matrix.

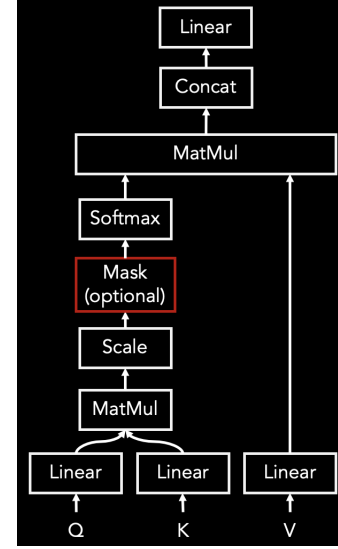


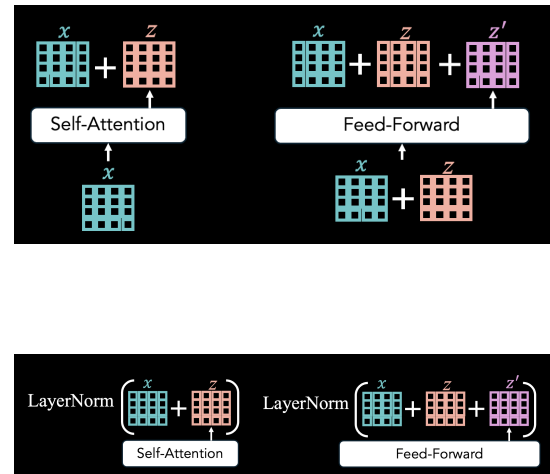
Figure 4.2: Architecture of an attention head [1].

The multi-head self-attention mechanism consists of multiple parallel attention heads, each learning distinct representation subspaces. The outputs from all attention heads are concatenated and then projected through a linear layer to produce the final combined representation [1].

#### 4.3.2 Add & Norm

**Residual Connections (Add):** Preserving information from earlier layers helps mitigate the vanishing gradient problem. In transformer encoders, this is achieved through *residual connections*, where the original input of a layer is added to its output. This mechanism enables the network to retain essential information across layers and facilitates more effective gradient flow during training [1].

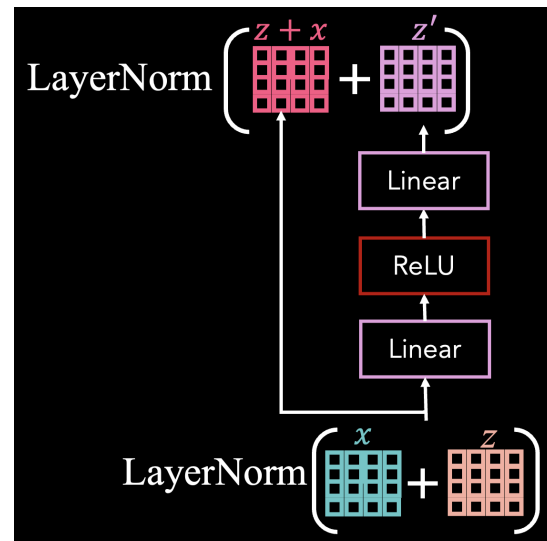
**Layer Normalization (Norm):** During training, the model may experience *internal covariate shift*, where the distribution of activations changes across layers, potentially leading to vanishing or exploding gradients. To address this challenge, *layer normalization* is applied to the outputs of deeper layers that in result, stabilizes training by reducing distributional shifts and ensures more consistent gradient flow [1].



### 4.3.3 Feed Forward Network

The feed-forward sub-layer introduces non-linearities into the encoder, thereby enhancing the model's capacity to capture complex patterns and non-linear relationships within sequential data. The feed forward network consists of two linear transformations separated by an activation function, typically the Rectified Linear Unit (ReLU) or the Gaussian Error Linear Unit (GELU) [1].

The ReLU activation applies  $\max(0, x)$  to each input  $x$ , effectively setting all negative values to zero. Although computationally efficient, ReLU suffers from the *dying ReLU* problem, where neurons can become permanently inactive during training and consistently output zero due to negative weighted inputs. In contrast, GELU is a smoother activation function that maps a value to  $x \times \Phi(x)$ , where  $\Phi(x)$  denotes the cumulative distribution function (CDF) of the standard normal distribution. This smooth approximation allows GELU to retain small negative values and has been shown to improve performance in transformer-based architectures [1].



## 4.4 Implementation

In this section, we use a small input text to develop a minimal decoder-only model, called **TinyGPT**. The goal is to gain familiarity with the operation of decoder-only models. The overall network architecture is shown below, and the complete implementation script is available on [GitHub](#).

```
class TinyGPT(nn.Module):
    def __init__(self, vocab_size, block_size, n_embed=128, n_head=4, n_layer=4, dropout=0.1):
        super(TinyGPT, self).__init__()
        self.token_embed = nn.Embedding(vocab_size, n_embed)
        self.pos_embed = nn.Embedding(block_size, n_embed)
        self.blocks = nn.Sequential(
            *[Block(n_embed, n_head, block_size, dropout) for _ in range(n_layer)]
        )
        self.ln_f = nn.LayerNorm(n_embed)
        self.lm_head = nn.Linear(n_embed, vocab_size)

    def forward(self, idx, targets=None):
        B, T = idx.shape
        token = self.token_embed(idx)
        pos = self.pos_embed(torch.arange(T, device=idx.device))
        x = token + pos
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.lm_head(x)
        loss = None
        if targets is not None:
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1))

        return logits, loss
```

As defined in the `__init__` function, the model network includes the following layers:

- `self.token_embed`: it creates an embedding layer that converts token indices (integers) into dense vector representations (embeddings).
- `self.pos_embed`: this layer create the positional encoding of the input tokens.

- `self.blocks`: it includes encoder blocks (layers), each with a multi-head self-attention head, a feed forward network, and the corresponding add & norm components.
- `self.ln_f`: this layer defines the final layer normalization applied to the transformer's output before passing it into the language modeling head.
- `self.lm_head`: it defines the language modeling head, i.e., the final layer that maps the hidden representations produced by the transformer into predicted token probabilities.

When an input passes through the *forward* function, it is first converted into token embeddings. Positional embeddings are then added to encode the order of tokens. The resulting representations are sequentially passed through all transformer blocks in the model. Within the *attention head*, the *self.register\_buffer* mechanism implements a lower-triangular mask to prevent the model from attending to future tokens during processing.

```
class Head(nn.Module):
    def __init__(self, n_embed, head_size, block_size, dropout):
        super(Head, self).__init__()
        self.key = nn.Linear(n_embed, head_size, bias=False)
        self.query = nn.Linear(n_embed, head_size, bias=False)
        self.value = nn.Linear(n_embed, head_size, bias=False)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer("tril", torch.tril(torch.ones(block_size, block_size)))

    def forward(self, x):
        B, T, C = x.shape
        k, q, v = self.key(x), self.query(x), self.value(x)
        att = (q @ k.transpose(-2, -1)) / math.sqrt(k.size(-1))
        att = att.masked_fill(self.tril[:T, :T]==0, float('-inf'))
        att = F.softmax(att, dim=-1)
        att = self.dropout(att)
        out = att @ v

        return out
```

The output of the final block is normalized using the last layer normalization and then fed into the language modeling head. At this stage, each token is represented by a hidden vector of size equal to the embedding dimension, and the final layer predicts the probability distribution over the vocabulary for the next token.

```
class Block(nn.Module):
    def __init__(self, n_embed, n_head, block_size, dropout):
        super(Block, self).__init__()
        self.mh = MultiHead(n_embed, n_head, block_size, dropout)
        self.ff = FeedForward(n_embed, dropout)
        self.ln1 = nn.LayerNorm(n_embed)
        self.ln2 = nn.LayerNorm(n_embed)

    def forward(self, x):
        x_p = self.ln1(x)
        x = x + self.mh(x_p)
        x_p = self.ln2(x)
        x = x + self.ff(x_p)

        return x

class FeedForward(nn.Module):
    def __init__(self, n_embed, dropout):
        super(FeedForward, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embed, 4*n_embed),
            nn.GELU(),
            nn.Linear(4*n_embed, n_embed),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        return self.net(x)
```

```
class MultiHead(nn.Module):
    def __init__(self, n_embed, n_head, block_size, dropout):
        super(MultiHead, self).__init__()
        head_size = n_embed // n_head
        self.heads = nn.ModuleList([
            Head(n_embed, head_size, block_size, dropout) for _ in range(n_head)
        ])
        self.proj = nn.Linear(n_embed, n_embed)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        proj = self.proj(out)
        return self.dropout(proj)
```

During training, the model compares these predicted logits with the true token IDs using cross-entropy loss, and updates its weights through backpropagation to minimize this loss.

```
num_epochs = 1000
for epoch in range(num_epochs):
    x_batch, y_batch = get_batch(block_size=BLOCK_SIZE, batch_size=BATCH_SIZE)
    _, loss = model(x_batch, y_batch)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item(): .4f}")
```

```
Epoch 100/1000, Loss: 1.6484
Epoch 200/1000, Loss: 0.4358
Epoch 300/1000, Loss: 0.1481
Epoch 400/1000, Loss: 0.0896
Epoch 500/1000, Loss: 0.0778
Epoch 600/1000, Loss: 0.0650
Epoch 700/1000, Loss: 0.0572
Epoch 800/1000, Loss: 0.0520
Epoch 900/1000, Loss: 0.0503
Epoch 1000/1000, Loss: 0.0469
```

#### 4.4.1 Evaluation

For evaluation, we provide the model with a starting prompt and ask it to generate the remaining text. Considering the small size of the training data, the generated outputs are reasonably accurate.

```
def generate(prompt="Jalāl ", block_size=16, max_new_tokens=400, temperature=0.8, top_k=50):
    idx = encode(prompt).unsqueeze(0).to(device)
    with torch.no_grad():
        for _ in range(max_new_tokens):
            logits, _ = model(idx[:, -block_size:])
            logits = logits[:, -1, :] / temperature
            if top_k:
                v, _ = torch.topk(logits, min(top_k, logits.size(-1)))
                logits[logits < v[:, [-1]]] = float('-inf')

            probs = F.softmax(logits, dim=-1)
            next_id = torch.multinomial(probs, num_samples=1)
            idx = torch.cat([idx, next_id], dim=1)

    return decode(idx[0].cpu())
```

```
print(generate(block_size=BLOCK_SIZE))
```

```
Jalāl al-Dīn Muḥammad Rūmī, or simply Rumi, was a 13th-century poet, Hanafi
faqīh, Maturīdī theologian, and Sufi mystic born during the Khwarazmian Empire.
Rumi's works are written in his mother tongue, Persian. He occasionally used the
Arabic language and single Turkish and Greek words in his versersersdsiothersianguue, Persian. He occasionally used the
Arabic language and single Turkish and Greek wo
```

Also, we compute the cross-entropy loss, perplexity, accuracy, bit per char (BPC), and distinct scores.

- **Cross entropy loss:** this metric measures the difference between the predicted probability distribution of a model and the true distribution of the target data. It quantifies how well the model's predicted probabilities match the actual outcomes, with lower values indicating better predictions and less uncertainty.
- **Perplexity:** it measures the model's uncertainty; lower perplexity implies that the model assigns higher probability to the actual next word in the sequence, resulting a more confident and accurate model.
- **Accuracy:** this metric is defined in terms of correct predictions over ground-truth data. However, since the tokens are in terms of characters, accuracy is not a reliable metric in the current evaluations.
- **Bit per char:** BPC measures the average number of bits a model needs to encode or predict each character in the text. Lower BPC indicates better predictive performance and less uncertainty, and it is equivalent to cross-entropy expressed in bits rather than nats.

- **Distinct scores:** these scores quantify diversity in generated text. To compute them, all n-grams of length  $n$  in the text are first extracted. The metric then calculates the proportion of unique n-grams relative to the total number of n-grams. A higher score indicates greater diversity, meaning the text is less repetitive.

Table 4.1 indicates the performance of TinyGPT w.r.t. the aforementioned evaluation metrics.

Table 4.1: Performance evaluation of TinyGPT

<b>Cross Entropy Loss</b>	<b>Perplexity</b>	<b>Accuracy (%)</b>	<b>BPC (bits)</b>	<b>Distinct-1</b>	<b>Distinct-2</b>
0.0399	1.04	98.48	0.058	0.113	0.432

# LLM Fine-tuning



# Parameter-efficient Fine-tuning

# Reinforcement Learning from Human Feedback

# Retrieval-Augmented Generation

# Agentic AI

# References

1. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in Neural Information Processing Systems*, vol. 30, 2017. [Online]. Available: <https://arxiv.org/abs/1706.03762>
2. Hugging Face, <http://huggingface.co/>.
3. LangChain, <https://www.langchain.com/>.
4. Pinterest, <https://www.pinterest.com/>.
5. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *North American Chapter of the Association for Computational Linguistics*, 2019. [Online]. Available: <https://api.semanticscholar.org/CorpusID:52967399>
6. Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” 2019. [Online]. Available: <https://api.semanticscholar.org/CorpusID:198953378>
7. A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, and *et al.*, “Palm: Scaling language modeling with pathways,” 2022. [Online]. Available: <https://arxiv.org/abs/2204.02311>
8. H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, “Llama: Open and efficient foundation language models,” 2023. [Online]. Available: <https://arxiv.org/abs/2302.13971>