GenAI Tutorial | 2025

# LLMs From Scratch

Forough Shirin Abkenar

# Contents

# List of Figures

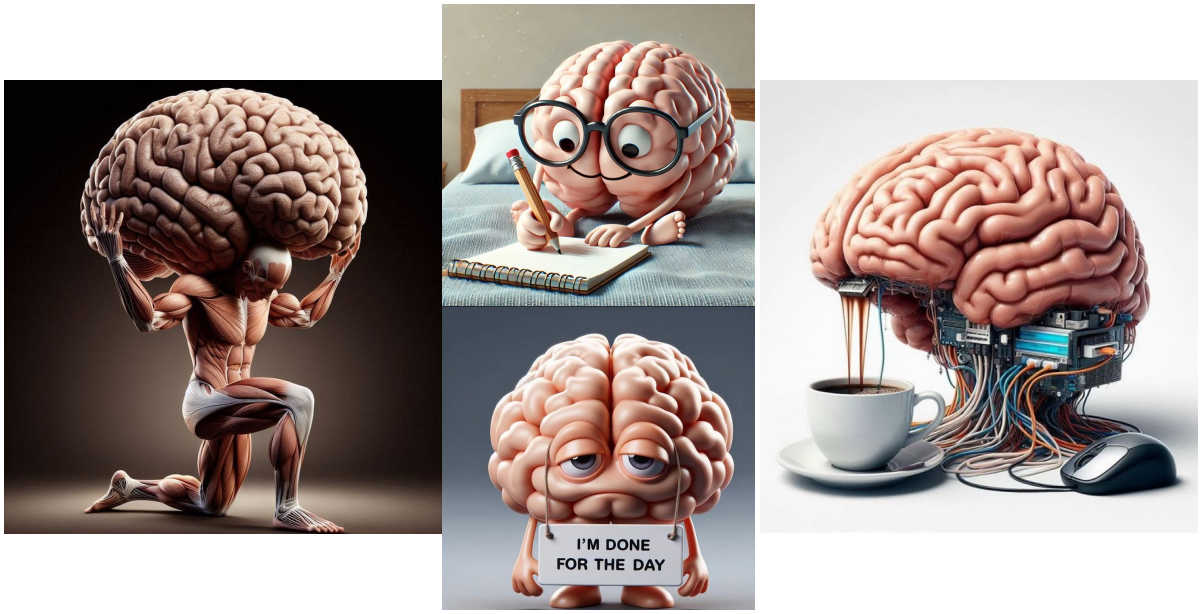


# List of Tables

# 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.

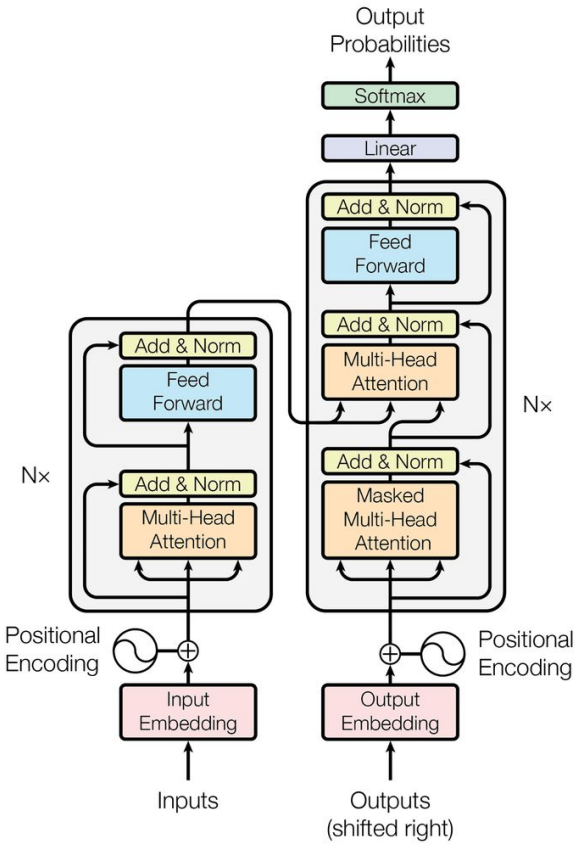



Figure 2.1: Architecture of an LLM including an encoder and a decoder [1].

# Encoder-only Models

The encoder-only models in LLMs owns 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. 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 types 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.

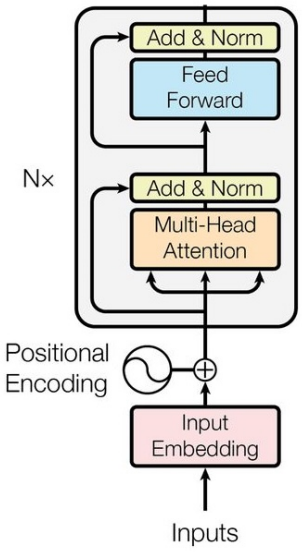



Figure 3.1: Encoder-only mode architecture [1].

## 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 prdefined embedding dimensiion. 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,', 'Matur
idi', 'theologian,', 'and', 'Sufi', 'mystic', 'born', 'during', 'the', 'Khwarazmian', 'Empire.']
```

Finally, tokens must be converted to token IDs. Therefore, the vocabulary (stoi in the codes) is defined an then, the encode function is employed 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', 'sim
ply: 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].

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

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.

```python
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.lm_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.lm_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 Nx 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.

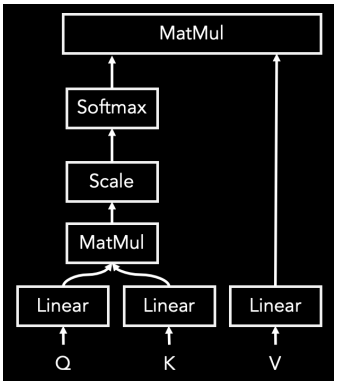
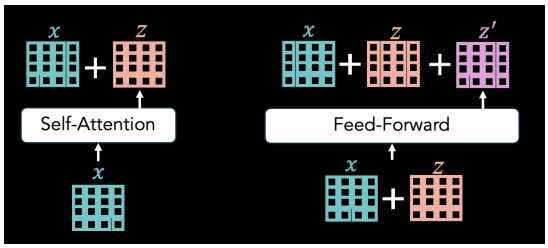



Figure 3.2: Architecture of an attention head [1].

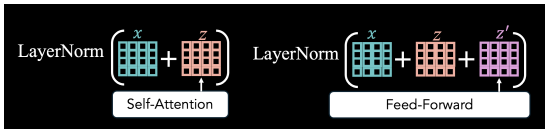
The multi-head self-attention mechanism consists of multiple parallel attention heads, each learning distinct representation subspaces. The outputs from all attention heads are concatenated and then projected through a linear layer to produce the final combined representation [1].

### 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].

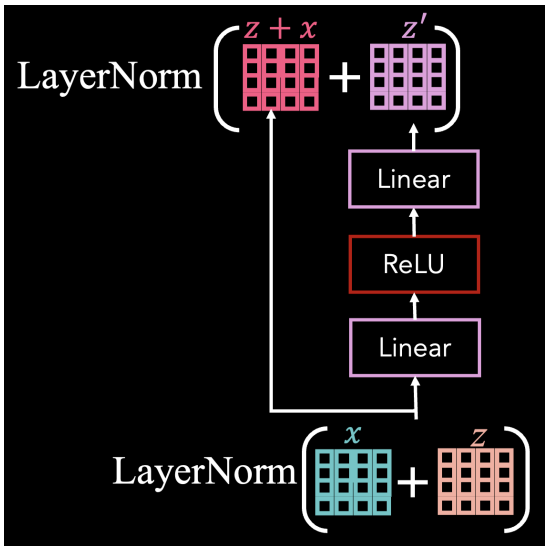



**Layer Normalization (Norm):** During training, the model may experience *internal covariate shift*, where the distribution of activations changes across layers, potentially leading to vanishing or exploding gradients. To address this challenge, *layer normalization* is applied to the outputs of deeper 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 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].

## 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.lm_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.lm_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.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 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__()
        head_size = n_embed // n_head
        self.heads = nn.ModuleList(
            [Head(n_embed, 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
```

### 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. Notably, 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. 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

# LLM Fine-tuning

# Parameter-efficinet 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," *ArXiv*, vol. abs/1907.11692, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:198953378