CH4 텍스트 생성을 위한 GPT 모델을 처음부터 구현하기

Build a Large Language Model (From Scratch)

Sebastian Raschka 저

2025-10-18

태영

어디를 지나고 있나

- 1장: LLM 개념
- 2장: 텍스트 처리 / 토크나이징
- 3장: 어텐션 메커니즘
- 4장: GPT 구현 및 생성
- 5장: Pretraining
- 6장: Fine-tuning (분류)
- 7장: Instruction 튜닝

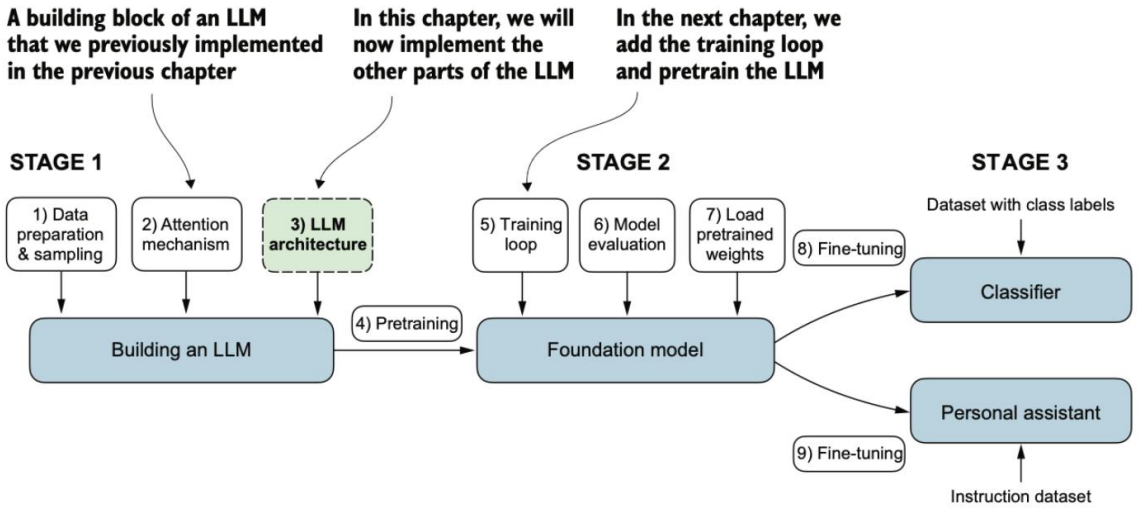

Figure 4.1 The three main stages of coding an LLM. This chapter focuses on step 3 of stage 1: implementing the LLM architecture.

GPT 아키텍처 개요

- GPT = **토큰 단위 생성 모델 (Generative Pretrained Transformer)**
- 내부는 반복적인 **트랜스포머 블록**으로 구성되어 복잡하지 않음
- 이 장에서는 GPT-2 Small (124M 파라미터) 기준으로 구현
- 최종 목표: **입력 토큰 → 문맥 벡터 → 다음 토큰 예측**

$$\text{output} = LN(TransformerBlock_N(\ldots(Embedding(x))))$$

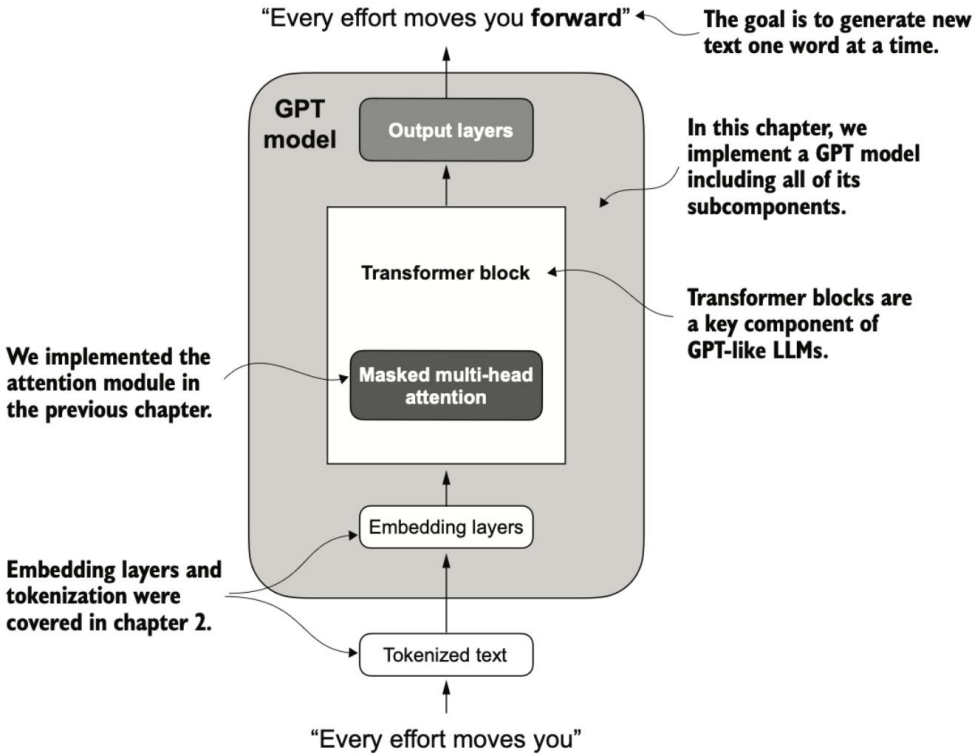

Figure 4.2 A GPT model. In addition to the embedding layers, it consists of one or more transformer blocks containing the masked multi-head attention module we previously implemented.

GPT-2 Small 구성 파라미터

```python
GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 1024,
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1,
    "qkv_bias": False
}
```

항목	값	설명
vocab_size	50 257	모델이 인식할 수 있는 **토큰의 총 개수** (단어 + 기호 + 서브워드 단위)
context_len	1 024	한 번의 입력에서 처리할 수 있는 **최대 토큰 길이** (문맥 창 크기)
emb_dim	768	각 토큰을 임베딩할 **벡터 차원 수**, 즉 표현 공간의 크기
layers	12	트랜스포머 **블록의 깊이** (모델의 층 수)
heads	12	멀티헤드 어텐션에서 병렬로 사용하는 **헤드의 수**
dropout	0.1	과적합 방지를 위한 **드롭아웃 비율** (훈련 시 뉴런 무작위 비활성화 확률)

DummyGPTModel 구조

- GPT 전체 구조의 **골격(skeleton)** 을 구현
- 핵심 모듈: **토큰/위치 임베딩 → 드롭아웃 → 트랜스포머 블록(n회) → 정규화 → 출력층**
- 실제 연산 대신 플레이스홀더(DummyTransformerBlock, DummyLayerNorm)로 구성

```
class DummyGPTModel(nn.Module):
    def __init__(self, cfg):
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])
        self.trf_blocks = nn.Sequential(
            *[DummyTransformerBlock(cfg) for _ in range(cfg["n_layers"])]
        )
        self.final_norm = DummyLayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(cfg["emb_dim"], cfg["vocab_size"], bias=False)
```

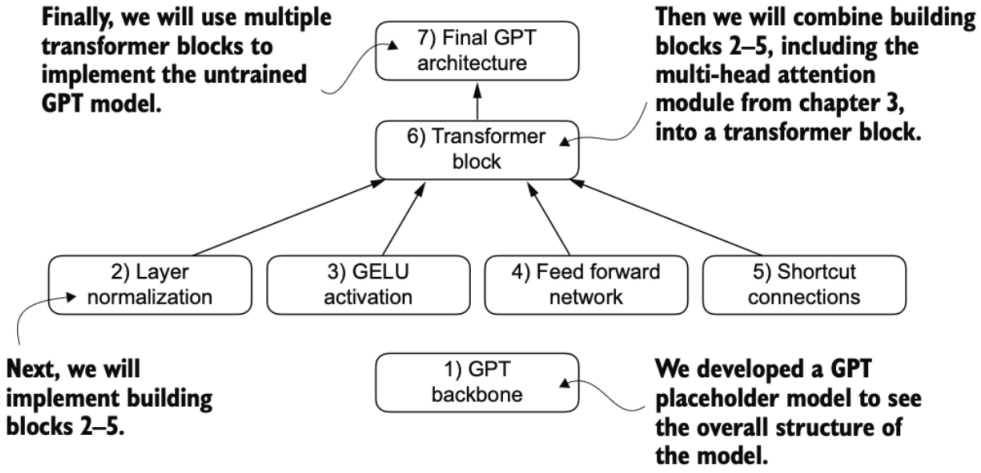

Figure 4.3 The order in which we code the GPT architecture. We start with the GPT backbone, a placeholder architecture, before getting to the individual core pieces and eventually assembling them in a transformer block for the final GPT architecture.

토크나이징과 출력 확인

- 입력 문장을 **토크나이즈 → 임베딩 → 모델 통과 → 로짓 출력**
- 출력 텐서 모양: [batch=2, tokens=4, vocab=50257]
- 각 토큰은 50,257차원의 확률 벡터로 표현됨

```
tensor([[6109, 3626, 6100, 345],
        [6109, 1110, 6622, 257]])
Output shape: torch.Size([2, 4, 50257])
```

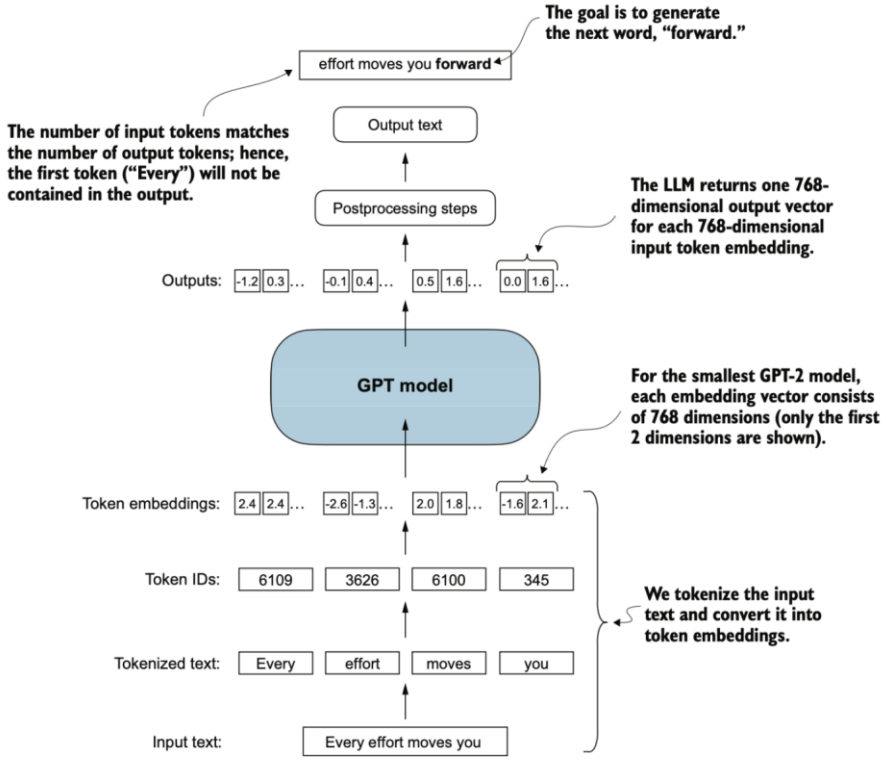

Figure 4.4 A big-picture overview showing how the input data is tokenized, embedded, and fed to the GPT model. Note that in our `DummyGPTClass` coded earlier, the token embedding is handled inside the GPT model. In LLMs, the embedded input token dimension typically matches the output dimension. The output embeddings here represent the context vectors (see chapter 3).

LayerNorm (정규화)

- 깊은 신경망의 **기울기 소실/폭발** → 학습 불안정
- 활성값을 **평균 0, 분산 1**로 표준화 → 안정·빠른 수렴
- GPT-2/현대 트랜스포머: **MHA 앞·뒤, 최종 출력 전**에 적용 (Pre-LN)

$$LN(x) = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta$$

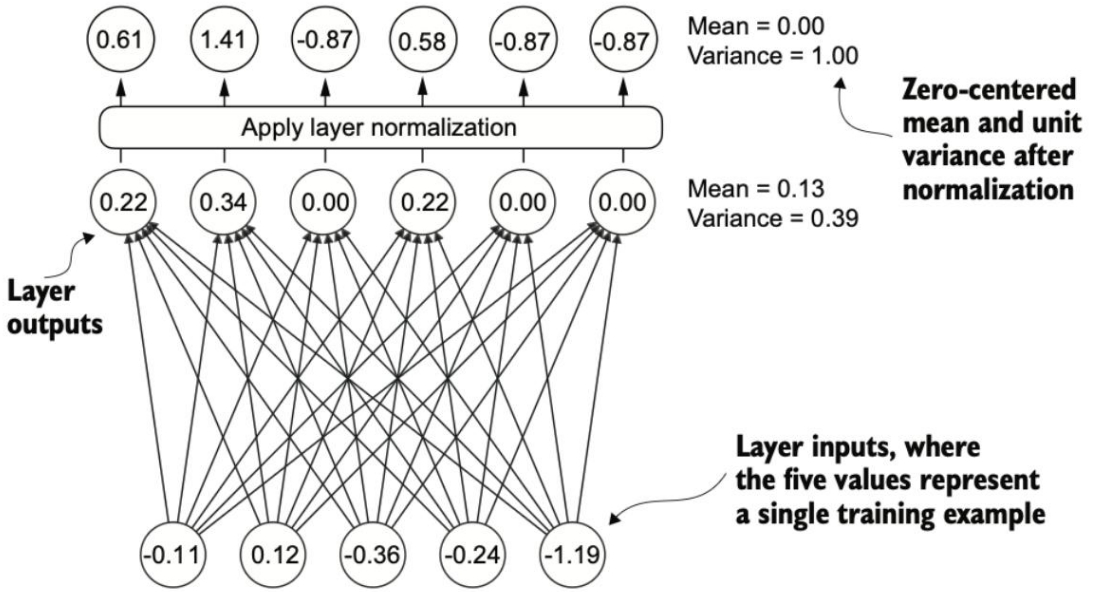

Figure 4.5 An illustration of layer normalization where the six outputs of the layer, also called activations, are normalized such that they have a 0 mean and a variance of 1.

LN 계산 방법

- dim=-1(임베딩 축) 기준으로 행(토큰)별 평균·산
- keepdim=True로 브로드캐스팅 간단
- 정규화 전/후 통계 비교로 효과 확인
- ReLU 이후 분포 왜곡 → LN으로 **평균≈0/분산≈1** 재정렬.
- 부동소수점 오차로 0에 매우 근접한 작은 수가 보일 수 있음(정상).

```
torch.manual_seed(123)
x = nn.Sequential(nn.Linear(5, 6), nn.ReLU())(torch.randn(2, 5))
m, v = x.mean(-1, True), x.var(-1, True)        # 전
xhat = (x - m) / torch.sqrt(v + 1e-8)           # 후
print(xhat.mean(-1, True), xhat.var(-1, True))  # ≈ 0, 1
```

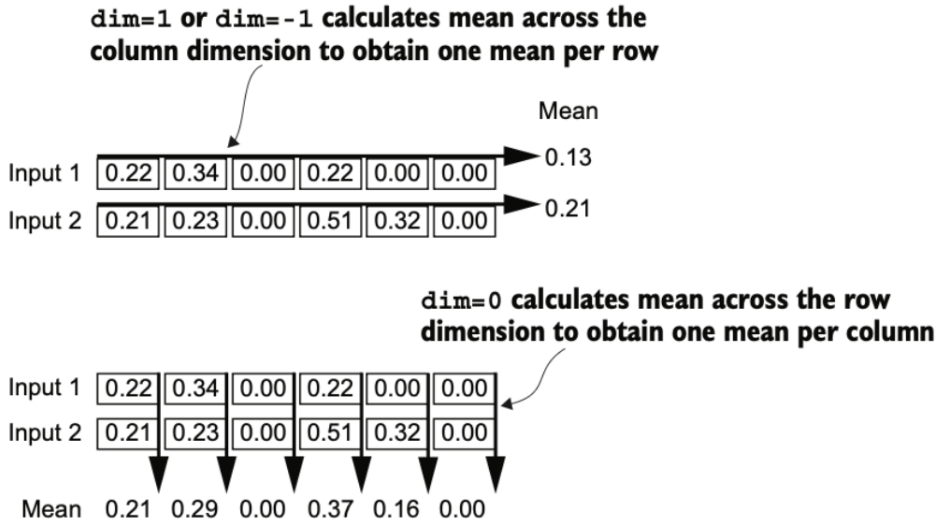

Figure 4.6 An illustration of the dim parameter when calculating the mean of a tensor. For instance, if we have a two-dimensional tensor (matrix) with dimensions `[rows, columns]`, using `dim=0` will perform the operation across rows (vertically, as shown at the bottom), resulting in an output that aggregates the data for each column. Using `dim=1` or `dim=-1` will perform the operation across columns (horizontally, as shown at the top), resulting in an output aggregating the data for each row.

활성함수 (GELU)

- ReLU → 단순하고 빠르지만 음수 입력 = 0, 죽은 뉴런 문제 발생
- GELU(Gaussian Error Linear Unit):부드러운 비선형성, 음수에도 작은 기여도 유지
- GPT-2 포함 대부분 LLM이 GELU 사용

```
class GELU(nn.Module):
    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(
            torch.sqrt(torch.tensor(2.0/torch.pi)) *
            (x + 0.044715 * x**3)
        ))
```

$$\text{GELU}(x) \approx 0.5 \cdot x \cdot \left(1 + \tanh\left[\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715 \cdot x^3\right)\right]\right)$$

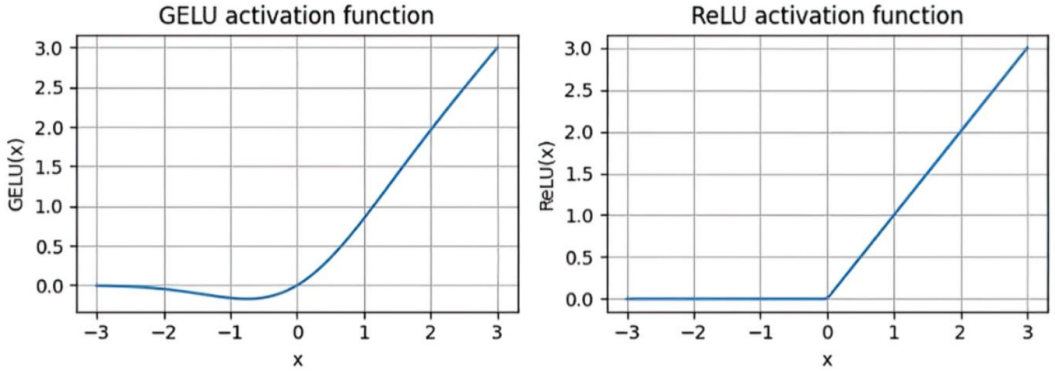

Figure 4.8 The output of the GELU and ReLU plots using matplotlib. The x-axis shows the function inputs and the y-axis shows the function outputs.

Feed-Forward Network

- 구조: Linear(확장) → **GELU** → Linear(축소)
- 입력·출력 차원 동일(768) / 내부는 4배 확장 (3072)
- 토큰별 독립 계산, **문맥 해석 이후 의미 변환 담당**

```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
            GELU(),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"])
        )
    def forward(self, x):
        return self.layers(x)
```

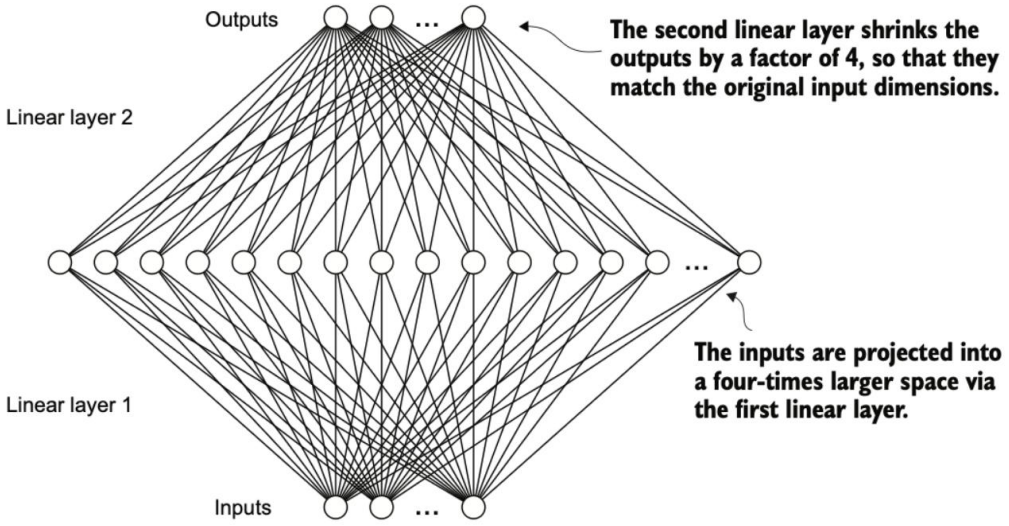

Figure 4.10 An illustration of the expansion and contraction of the layer outputs in the feed forward neural network. First, the inputs expand by a factor of 4 from 768 to 3,072 values. Then, the second layer compresses the 3,072 values back into a 768-dimensional representation.

Skip Connection

- **기울기 소실(Vanishing Gradient)** → 깊은 층의 학습 어려움
- **숏컷 연결**: 입력을 출력에 더해 **짧은 경로(Residual Path)** 확보
- 기울기가 앞층까지 원활히 흐르게 하여 **안정적 학습** 가능
- GPT·ResNet 등 대부분의 심층 모델 핵심 구조

$$y = x + F(x)$$

```
def forward(self, x):
    for layer in self.layers:
        layer_output = layer(x)
        if self.use_shortcut and x.shape == layer_output.shape:
            x = x + layer_output
        else:
            x = layer_output
    return x
```

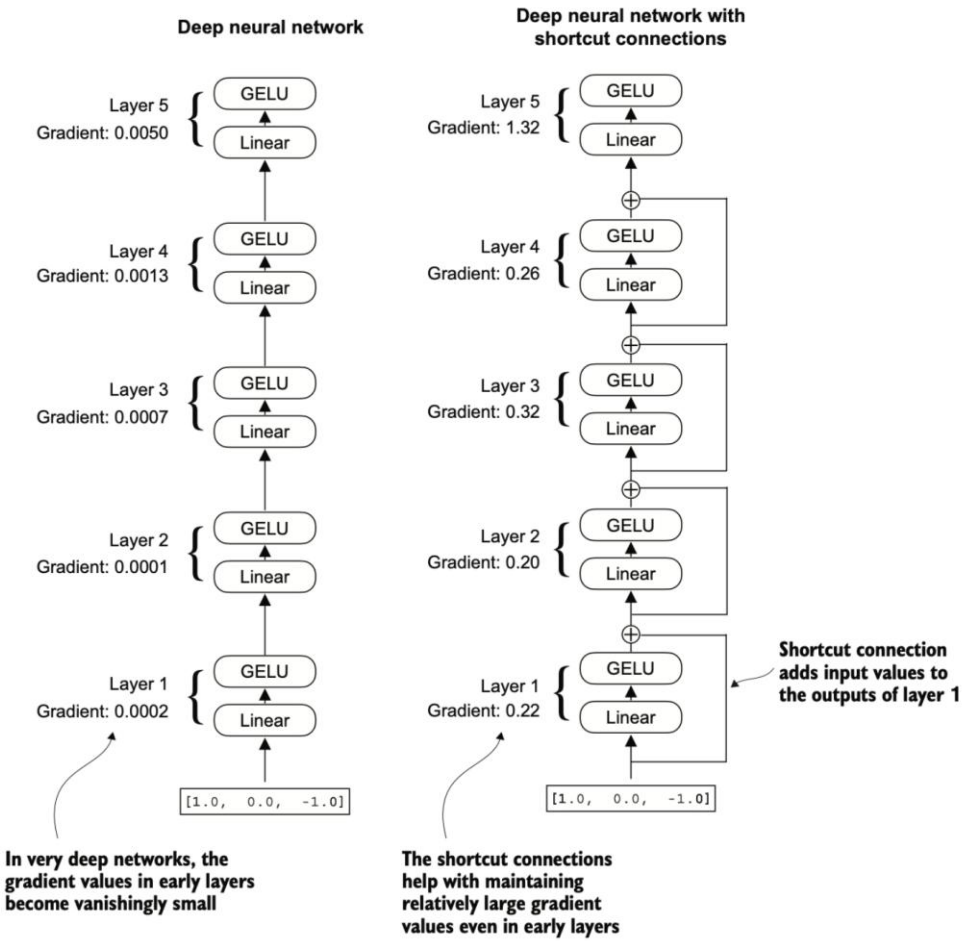

Figure 4.12 A comparison between a deep neural network consisting of five layers without (left) and with shortcut connections (right). Shortcut connections involve adding the inputs of a layer to its outputs, effectively creating an alternate path that bypasses certain layers. The gradients denote the mean absolute gradient at each layer, which we compute in listing 4.5.

Transformer Block

- GPT의 기본 단위: **어텐션 + 정규화 + 피드포워드 + 숏컷 연결**
- 각 블록은 **입력과 동일한 차원(768)** 을 유지하며 정보만 변환
- **Pre-LayerNorm 구조** → 학습 안정성, 기울기 소실 완화
- 12개 블록 반복 → GPT-2 Small (약 1.24억 파라미터) 구성

```
def forward(self, x):
    shortcut = x
    x = self.norm1(x)
    x = self.att(x)
    x = self.drop_shortcut(x)
    x = x + shortcut

    shortcut = x
    x = self.norm2(x)
    x = self.ff(x)
    x = self.drop_shortcut(x)
    x = x + shortcut
    return x
```

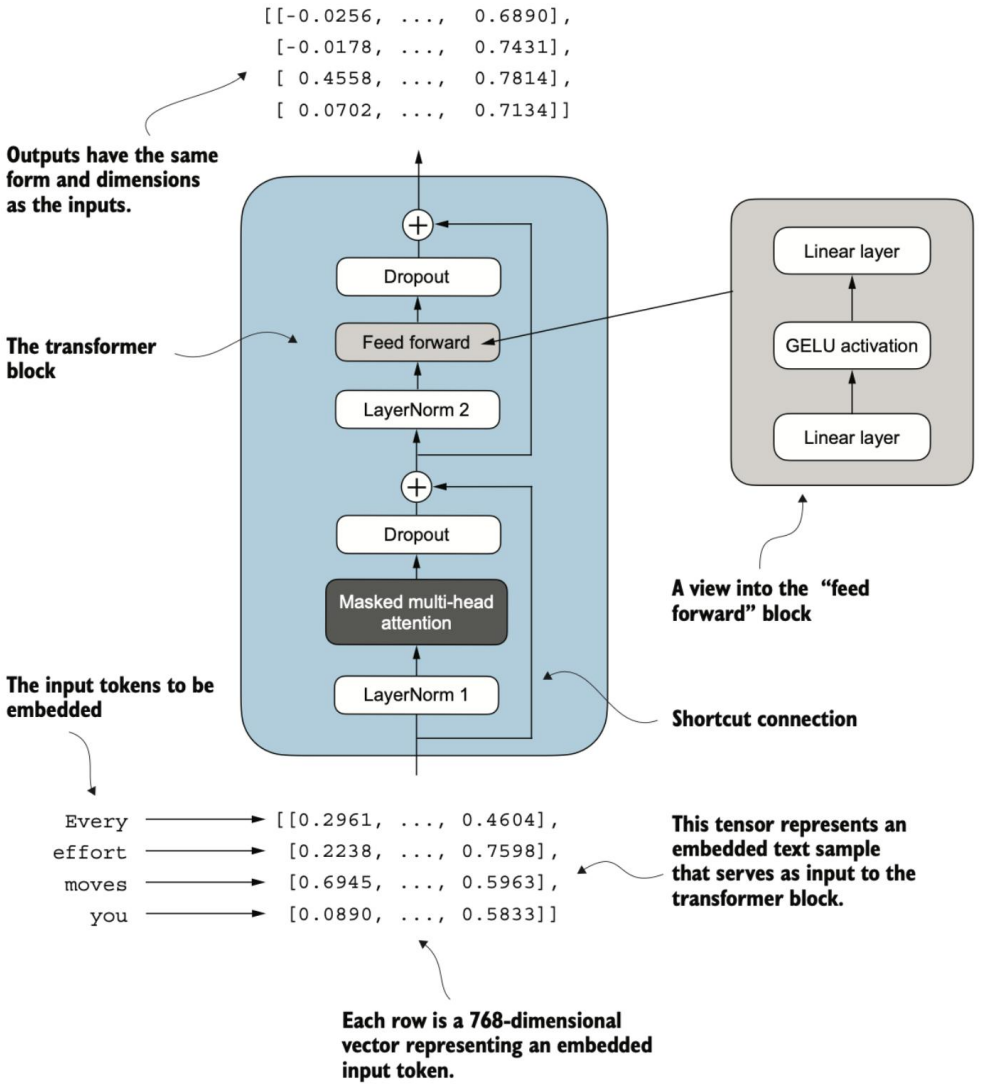

Figure 4.13 An illustration of a transformer block. Input tokens have been embedded into 768-dimensional vectors. Each row corresponds to one token's vector representation. The outputs of the transformer block are vectors of the same dimension as the input, which can then be fed into subsequent layers in an LLM.

GPT 모델 전체 구조

- GPT = **임베딩 → 트랜스포머 블록(n회 반복) → 정규화 → 출력층**
- GPT-2 Small: **12개 블록**, 1억 2,400만 파라미터
- 각 블록은 입력/출력 차원(768) 유지 → **Stack 가능 구조**
- 출력층: **어휘 크기 50,257차원** 로짓 생성

```
class GPTModel(nn.Module):
    def __init__(self, cfg):
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])]
        )
        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(cfg["emb_dim"], cfg["vocab_size"], bias=False)
```

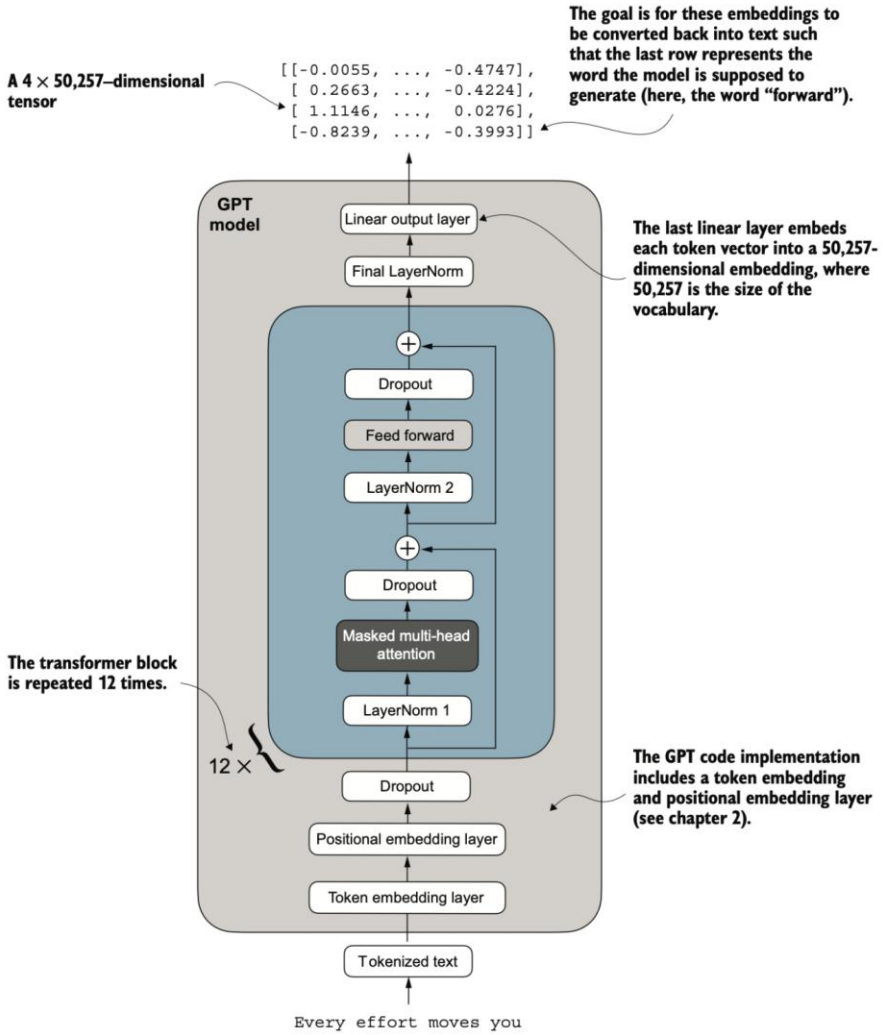

Figure 4.15 An overview of the GPT model architecture showing the flow of data through the GPT model. Starting from the bottom, tokenized text is first converted into token embeddings, which are then augmented with positional embeddings. This combined information forms a tensor that is passed through a series of transformer blocks shown in the center (each containing multi-head attention and feed forward neural network layers with dropout and layer normalization), which are stacked on top of each other and repeated 12 times.

GPT 텍스트 생성

- GPT는 **이전 문맥(context)** 을 보고 한 번에 **하나의 토큰**을 예측
- 매 스텝: logits → softmax → argmax 로 다음 토큰 선택
- 생성된 토큰을 입력에 덧붙여 반복 → **문장 완성 (Greedy Decoding)**
- 학습 전 모델은 무작위 출력을 내지만, 구조는 완전한 LLM 생성 루프

```python
def generate_text_simple(model, idx, max_new_tokens, context_size):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)
        logits = logits[:, -1, :]
        probas = torch.softmax(logits, dim=-1)
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)
        idx = torch.cat((idx, idx_next), dim=1)
    return idx
```

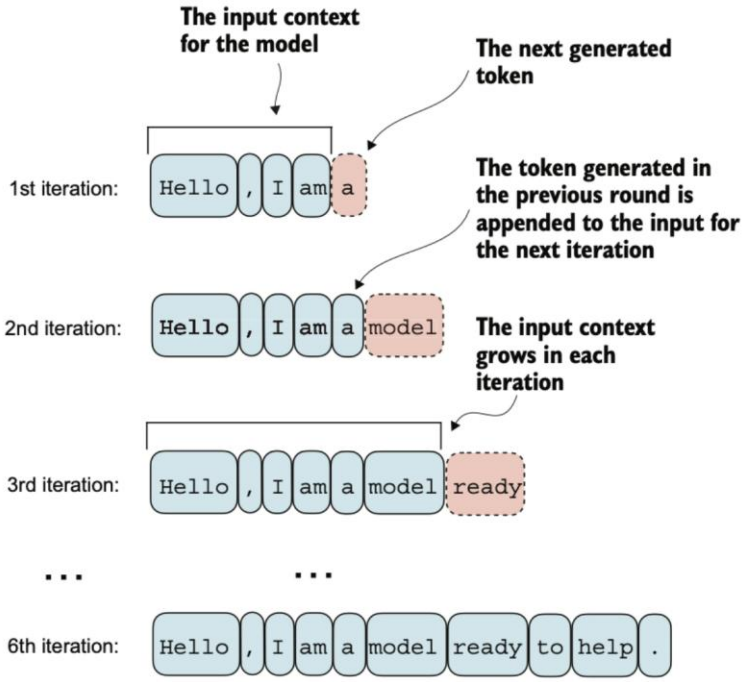

Figure 4.16 The step-by-step process by which an LLM generates text, one token at a time. Starting with an initial input context ("Hello, I am"), the model predicts a subsequent token during each iteration, appending it to the input context for the next round of prediction. As shown, the first iteration adds "a," the second "model," and the third "ready," progressively building the sentence.

정리

- **안정적인 학습 기반**: **레이어 정규화**로 각 층의 출력을 안정시키고, **숏컷 연결**을 통해 깊은 신경망의 그래디언트 소실 문제를 완화합니다.
- **핵심 구성 요소**: GPT 모델은 **'멀티헤드 어텐션'**과 GELU 활성화 함수를 사용한 **'피드포워드 네트워크'**를 결합한 **트랜스포머 블록**을 반복적으로 쌓아 만듭니다.
- **텍스트 생성 원리**: 주어진 문맥을 바탕으로 **다음에 올 단어(토큰)를 순차적으로 예측**하는 방식으로 문장을 생성합니다.
- **학습의 중요성**: 무작위 가중치를 가진 초기 모델은 의미 없는 텍스트를 생성하므로, 일관성 있는 문장을 만들기 위해서는 **대규모 데이터셋을 통한 모델 학습이 필수적**입니다.