

Paper Review

GPT / BERT Paper Review

Improving Language Understanding by Generative Pre-Training - Radford, Alec et al.

Language Models Are Unsupervised Multitask Learners - Radford, Alec et al.

Language Models Are Few-Shot Learners - Brown, Tom B. et al.

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding - Devlin, Jacob et al.

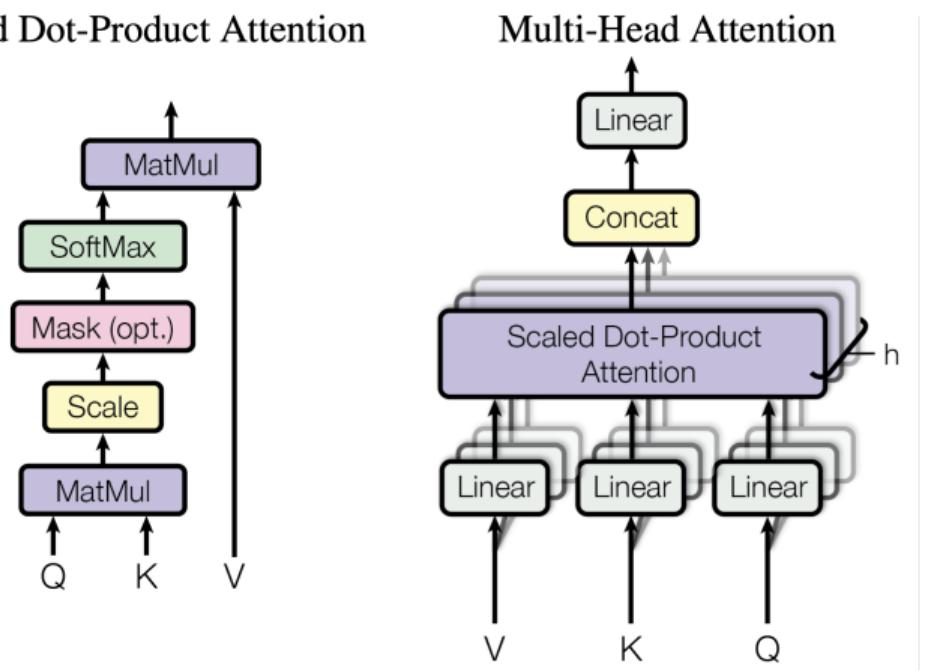
→ CONTENTS

- 1 Paradigm Shift
- 2 GPT(Decoder Only)
- 3 BERT(Encoder Only)
- 4 Comparison
- 5 Code Implement

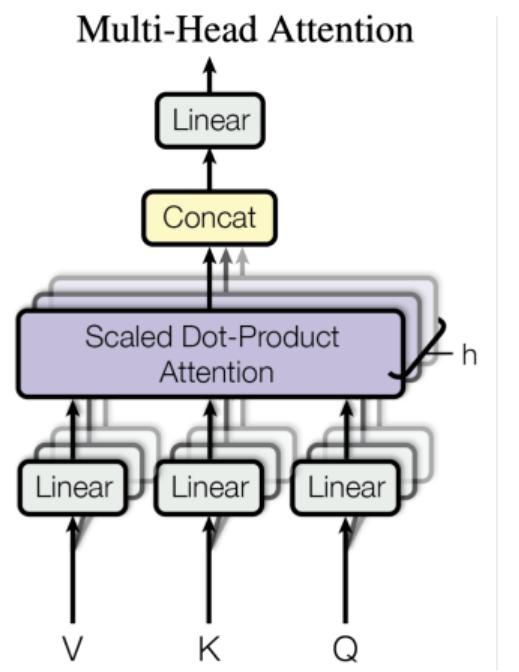
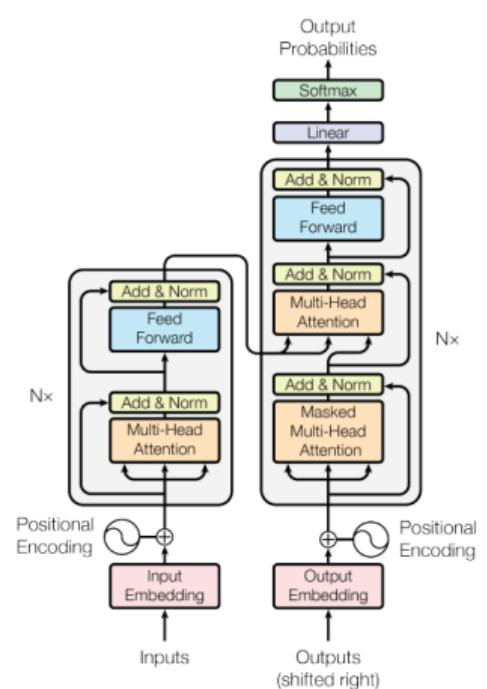
Chapter 1

Transformer Idea

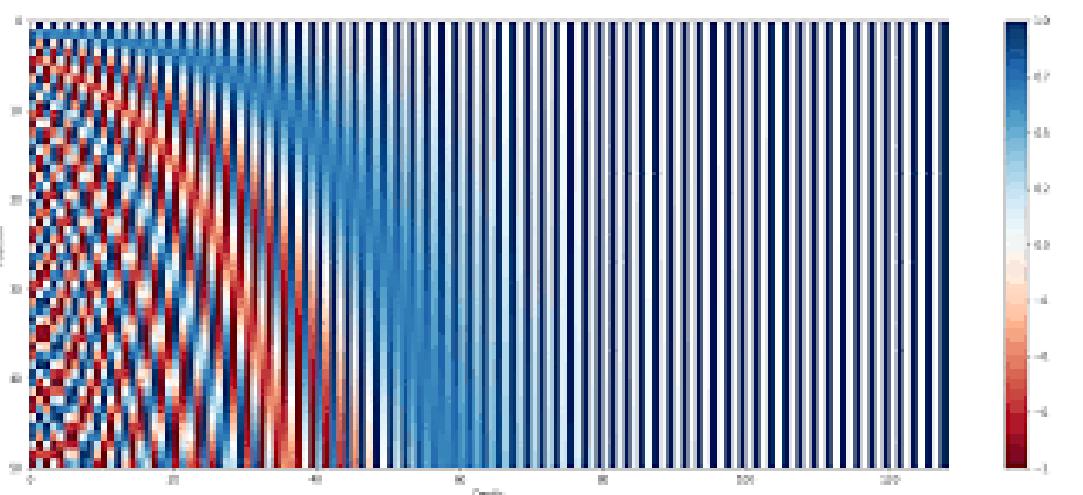
Scaled Dot-Product Attention



Multi-Head Attention

Scaled Dot-Product
Multi Head Self Attention

Architecture

Sinusoidal Positional
Embedding

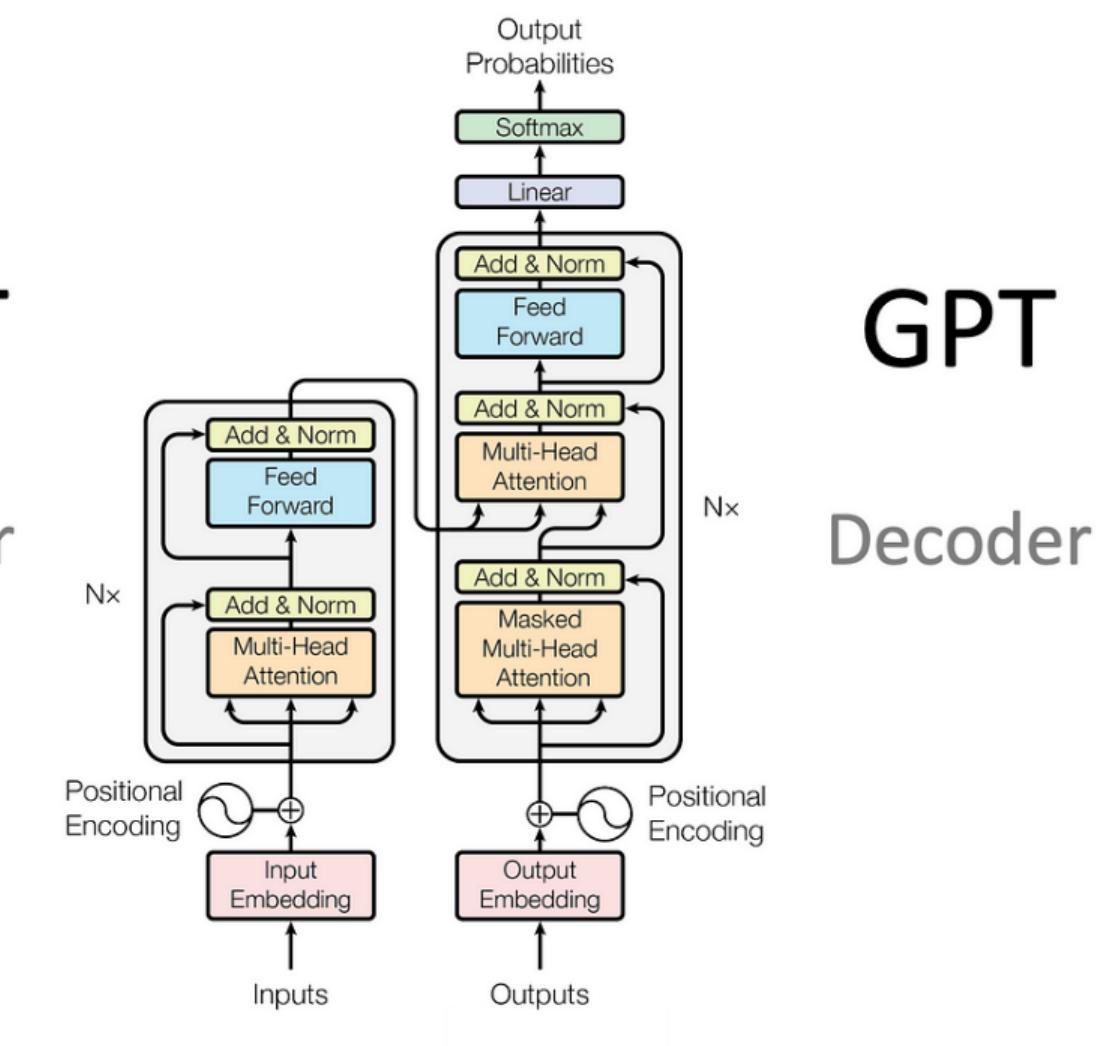
Chapter 1

Paradigm Shift

- 1 Attention으로 병렬처리가 되는데 시간을 더 단축시킬 수는 없을까?
- 2 Encoder-Decoder를 다 써야할까?
- 3 학습가능한 Positional Encoding이 있는데 지도학습이 필요할까?



BERT
Encoder



Chapter 2

GPT(Decoder Only)

GPT 1 Architecture

Decoder Only

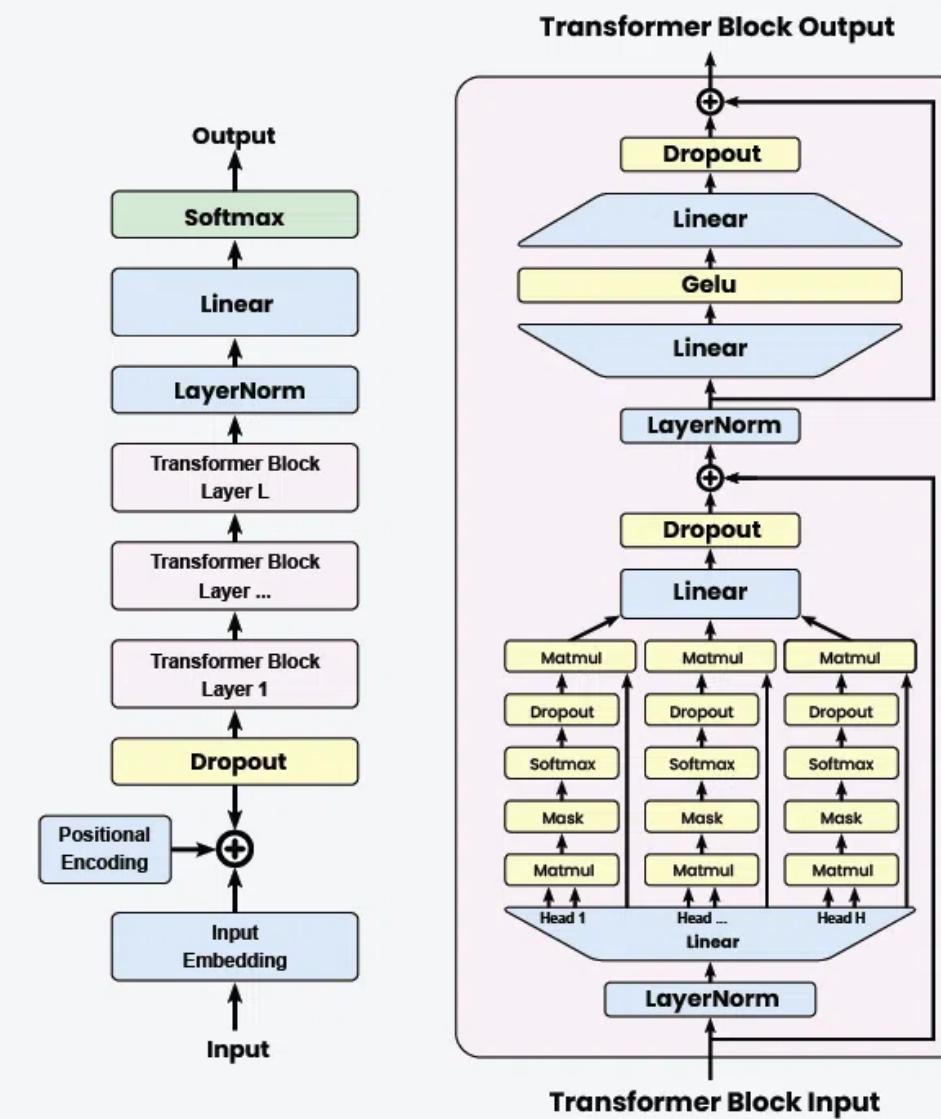
- 기본적으로 GPT 모델의 Architecture는 Transformer Decoder Block의 형태를 유지함

사라진 Cross-Attention

- Transformer Decoder Block 대비 Cross Attention 하는 한 개의 Attention Block을 제거했음
- Encoder가 없기 때문에 Cross Attention 할 대상이 없어졌기 때문

바뀐 Activation Function

- 원래 Transformer가 FFN을 통해 나온 벡터에 ReLU를 사용했으나 GPT는 GELU를 사용함
- GELU는 매끄러운 기울기를 제공하여 학습의 안정성을 강화함



GPT(Decoder Only)

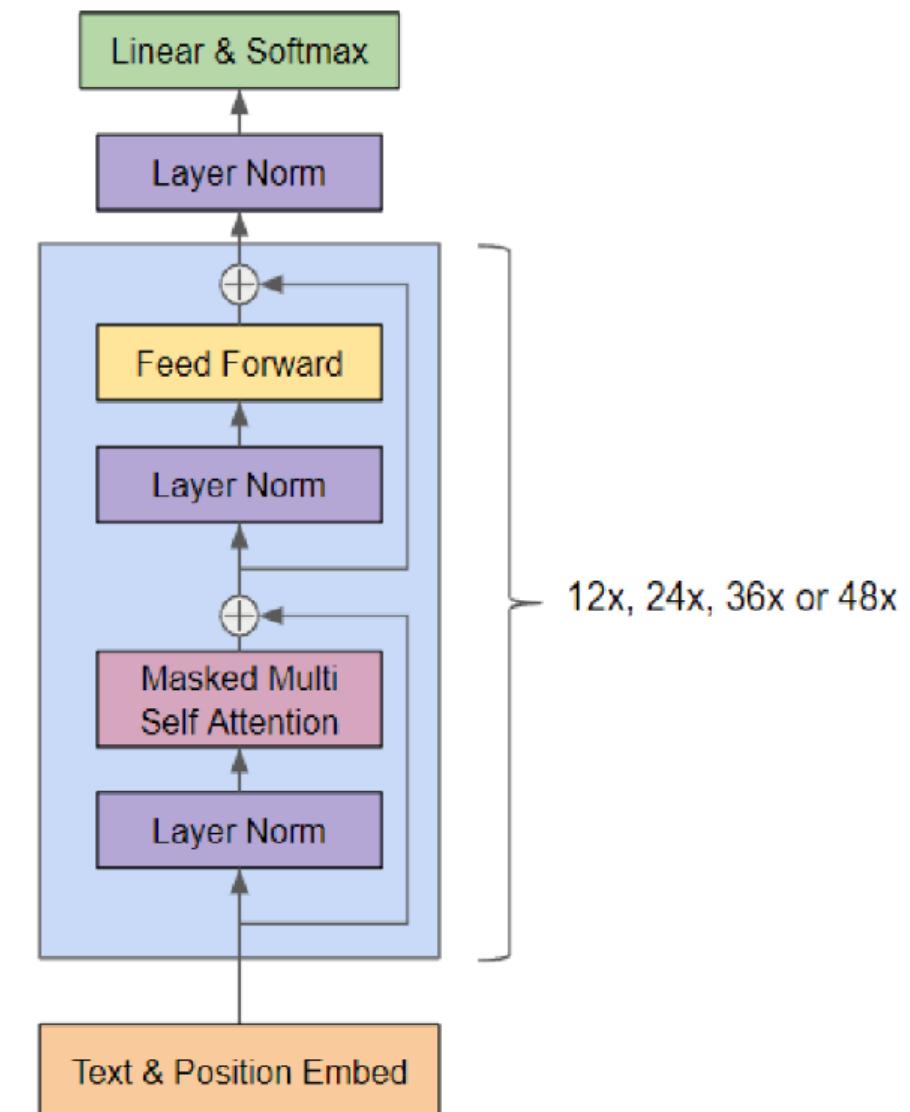
GPT 2 Architecture

Pre-LN

- Residual Connection을 통해 추가되는 데이터가 원본과 동일하여 학습 불안정이 해소됨
- 학습 시에 발생하는 기울기 소실이 이전에 비해 더 감소했기 때문에 더 깊은 레이어를 구성할 수 있게 됨

Final LN

- Pre-LN에서는 잔차가 누적되면 표현의 스케일/분산이 층을 따라 떠밀리듯이 달라질 수 있음
- 맨 끝에 한 번 더 LN(final LayerNorm)을 걸어 출력 분포를 고정
- 학습 안정성과 하이퍼파라미터 민감도를 크게 줄여 주는 실용적 관례로 자리 잡음

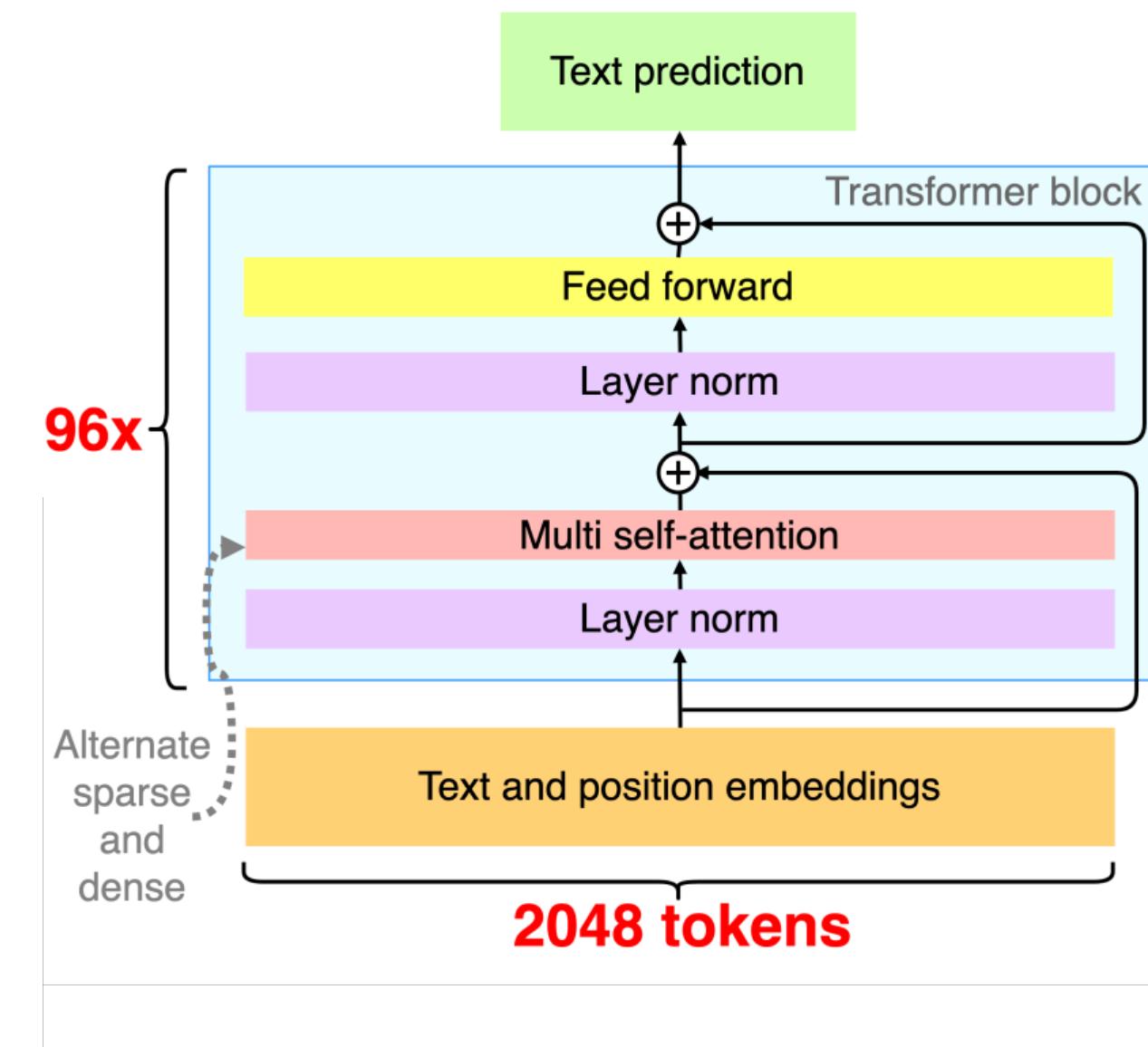
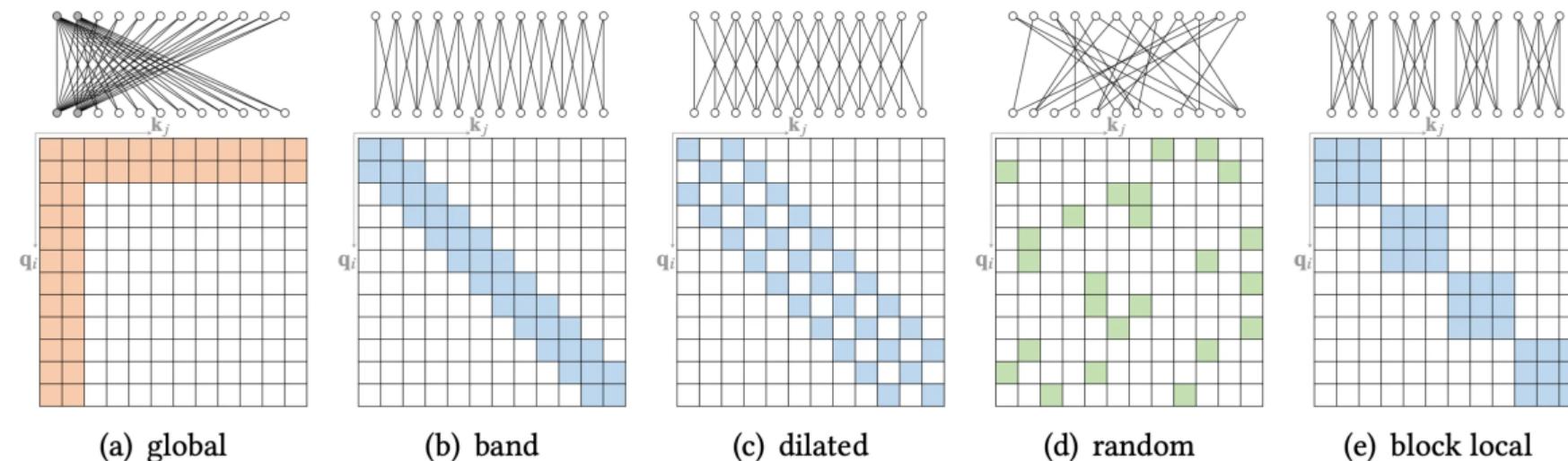


GPT(Decoder Only)

GPT 3 Architecture

Locally Banded Sparse Local Attention

- 일부 레이어에서 완전(dense) 어텐션 대신 로컬 밴드(locally-banded) 스파스 어텐션을 교대로 배치
- 각 토큰이 전 토큰 n개를 다 보는 대신, 자기 주변 w개(슬라이딩 윈도우)만 보게 해서 주의 행렬을 띠(band) 모양으로 만듦



Chapter 2

GPT(Decoder Only)

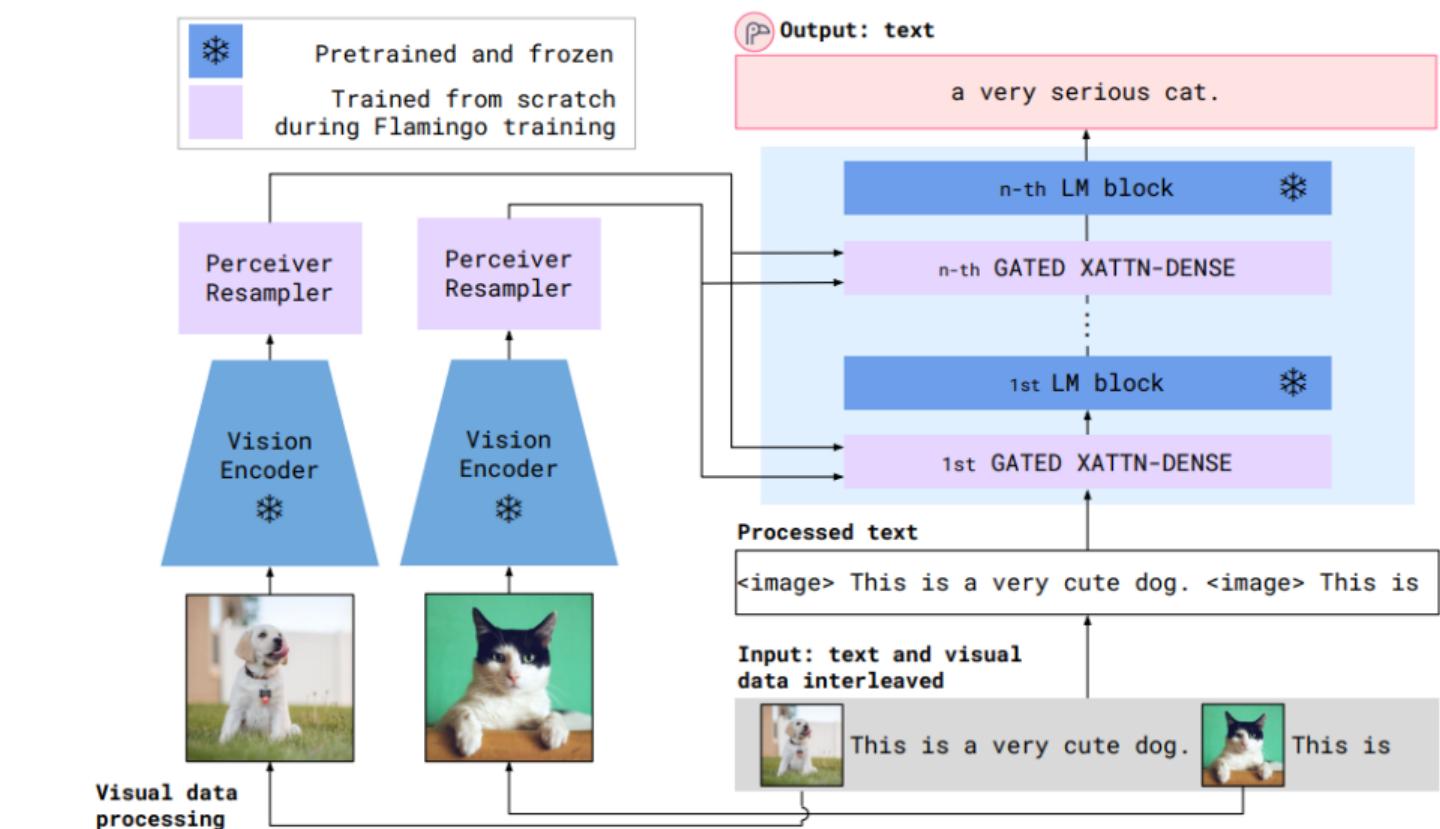
GPT 4, 5는 무엇이 달라졌나

GPT 4

- 멀티모달 수용 - 텍스트뿐 아니라 이미지 입력을 직접 받아 텍스트로 답변
- 세부 스펙 비공개

GPT 5

- 통합형 시스템 - 하나의 제품 경험 안에서 빠른 응답 모델 ↔ 깊은 사고 모델(GPT-5 “thinking”)을 라우터가 자동 선택해 사용
- 복잡도/의도/툴 사용 여부에 따라 ‘생각의 깊이’를 가변적으로 조절
- 세부 스펙 비공개



BERT(Encoder Only)

BERT Architecture

Encoder Only

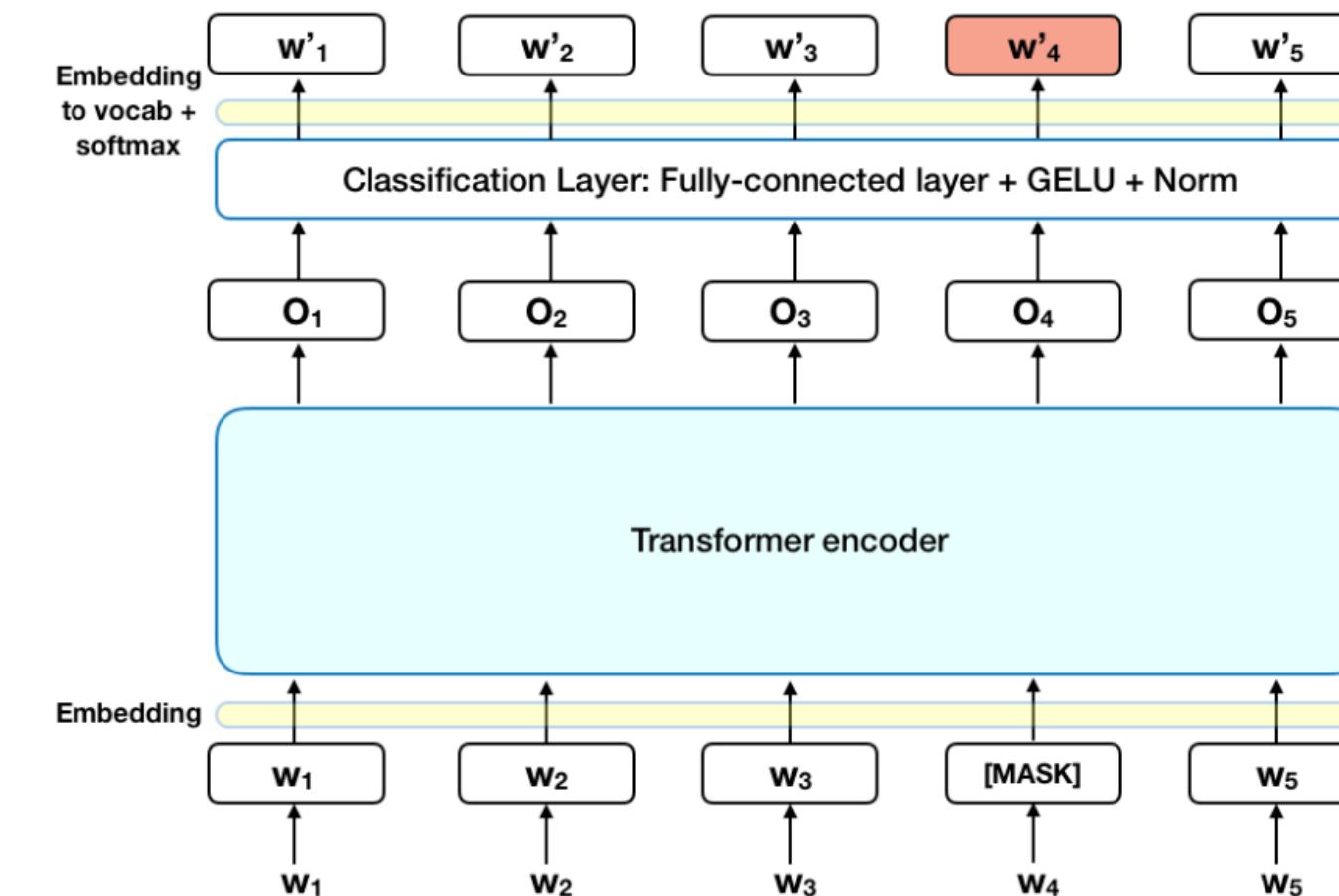
- 기본적으로 BERT 모델의 Architecture는 Transformer Encoder Block의 형태를 유지함

양방향성

- 이름에서 알 수 있듯, 단어의 양방향 문맥을 파악하여 이해와 판별에 유리함

Masked Language Modeling

- 입력 토큰의 15%를 마스킹 타깃으로 선정하여 가리고 그 중에서 80%는 [MASK]로 바꾸고, 10%는 임의의 다른 토큰으로 바꾸고, 10%는 그대로 둠



Chapter 3

BERT(Encoder Only)

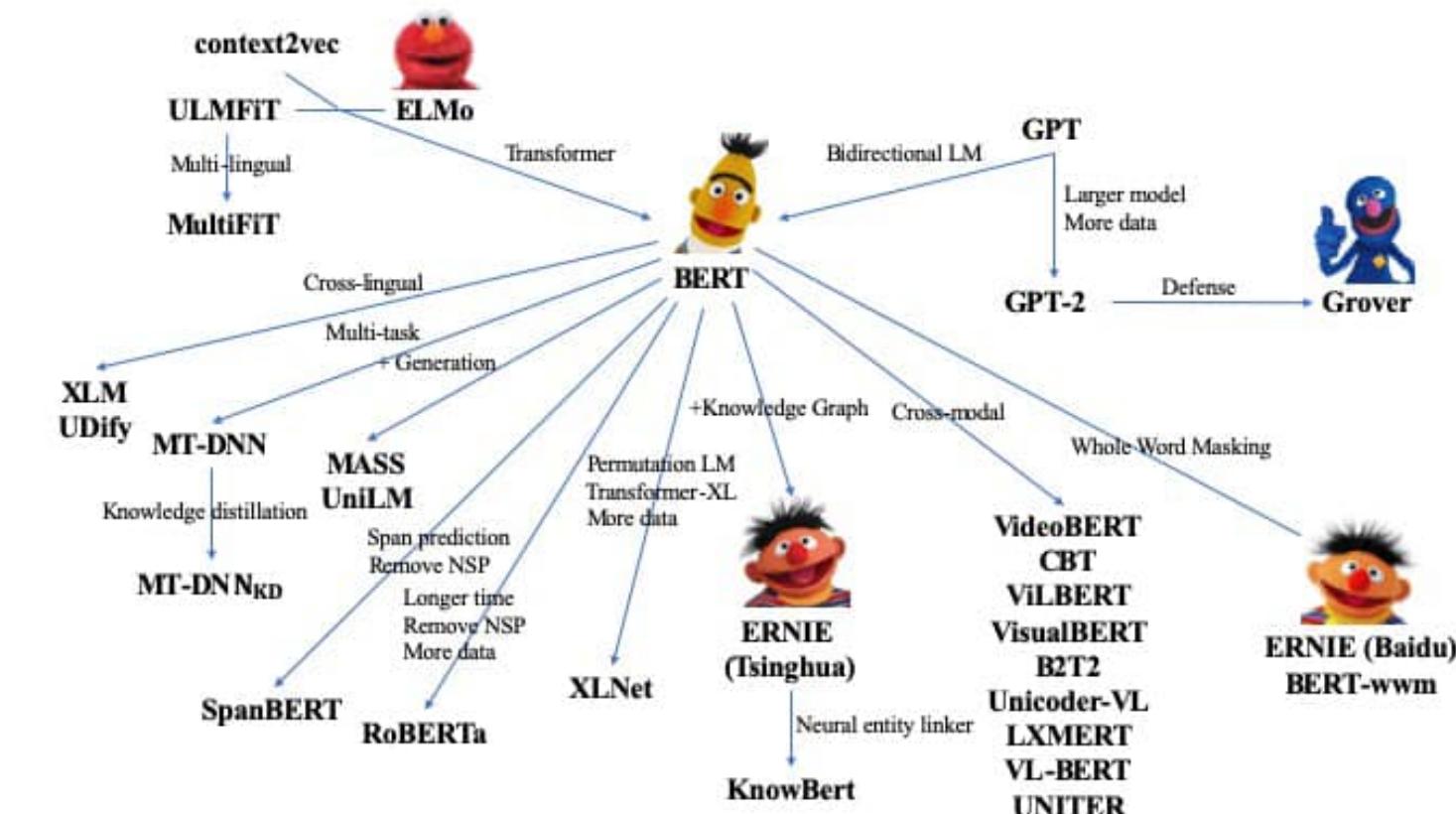
BERT

NSP(Next Sentence Prediction)

- 문장쌍이 실제로 연속인지 이진 분류
- 양방향 문맥 덕에 이해·판별 과제(분류/랭킹/태깅)에 매우 강함.
대량 배치 추론에 병렬로 빠르고 안정적

대표 파생 모델

- RoBERTa(더 큰 데이터, NSP 제거, 동적 마스킹)
- ALBERT(파라미터 공유/분해)
- DeBERTa(상대 위치·분리형 어텐션)
- DistilBERT(경량화)



By Xiaozhi Wang & Zhengyun Zhang @T

Chapter 4

Comparison

1

Purpose

GPT

- 전체 문장 분포 $p(x)$ 를 체인
룰로 직접 모델링
- 생성을 잘하게 만드는 범용
언어모델 지향
- 프롬프트만 바꿔 태스크를
생성 문제로 통일

BERT

- 손상된 입력 복원으로 표
현 $f(x)$ 을 고품질로 학습
- 본질은 이해/판별($p(y|x)$)에
강한 인코더
- 다양한 다운스트림에 임베딩
기반 전이가 목적

2

Architecture

GPT

- Decoder-only 트랜스포머
- 인과(미래 차단) 마스킹으로
왼쪽 문맥만 참조
- 토큰을 한 글자씩 생성하는
구조와 완벽히 정렬

BERT

- Encoder-only 트랜스포머
- 양방향 self-attention으로
좌·우 문맥 동시 활용
- 전 문맥 이해에 최적화(생성
은 비주류)

3

Pre-Training

GPT

- 목표: 다음 토큰 예측
- 대규모 웹 코퍼스로 라벨 없
이 스케일업 용이
- 컨텍스트를 늘릴수록
few/zero-shot 능력 자연
발생

BERT

- 목표: MLM + NSP
- [MASK]/랜덤치환/그대로
혼합으로 분포 미스매치 완
화
- 학습 후 태스크별 파인튜닝
이 전형적 사용법

4

Usage Pattern

GPT

- 프롬프트 엔지니어링 + In-
Context Learning 중심
- 요약/번역/대화/코드 등 개
방형 생성 과제에 적합
- Instruction tuning로 지시
따르기/대화성 강화

BERT

- [CLS] 벡터 + 작은 헤드로
분류/NLI(Natural
Language Inference, 자
연어 추론)
- NER/태깅/랭킹/검색 임베딩
에 강함
- 소량 라벨로도 파인튜닝 성
능이 잘 나옴

5

Serving Efficiency

GPT

- KV-cache로 증분 디코딩
(토큰당 자연 최소화)
- 길어질수록 직렬 생성 비용
증가
- 스케일 커질수록 성능이 예
측 가능하게 상승

BERT

- 완전 병렬 인퍼런스(한 번에
전 토큰 처리)로 자연 감소
- 대량 트래픽 분류/검색에서
비용 효율이 매우 좋음
- 기본 window(512차원) 내
에서 고정 길이 처리가 안정
적

Chapter 5

Code Implement

1 GPT 1

Positional Encoding

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, max_len: int = 5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() *
                            (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0).transpose(0, 1)
        self.register_buffer('pe', pe)

    def forward(self, x):
        return x + self.pe[:x.size(0), :]
```

GPT 1 Model (Transformer Decoder Block)

```
class GPTBlock(nn.Module):
    def __init__(self, d_model: int, n_heads: int, d_ff: int, dropout: float = 0.1):
        super().__init__()
        self.ln1 = nn.LayerNorm(d_model)
        self.ln2 = nn.LayerNorm(d_model)
        self.attn = nn.MultiheadAttention(d_model, n_heads, dropout=dropout, batch_first=True)
        self.ff = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model),
            nn.Dropout(dropout)
        )
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, attn_mask=None):
        # Self-attention
        norm_x = self.ln1(x)
        attn_out, _ = self.attn(norm_x, norm_x, norm_x, attn_mask=attn_mask)
        x = x + self.dropout(attn_out)

        # Feed-forward
        norm_x = self.ln2(x)
        ff_out = self.ff(norm_x)
        x = x + self.dropout(ff_out)

    return x
```

Chapter 5

Code Implement

1 GPT1

```
모델 파라미터 수: 29,165,328
훈련 시작...
Epoch 0, Loss: 9.2388
테스트 번역: '나는 밥을 먹는다' -> 'I'

-----
Epoch 20, Loss: 2.3618
테스트 번역: '나는 밥을 먹는다' -> 'I have breakfast'

-----
Epoch 40, Loss: 0.6896
테스트 번역: '나는 밥을 먹는다' -> 'I have breakfast'

-----
Epoch 60, Loss: 0.1947
테스트 번역: '나는 밥을 먹는다' -> 'I have breakfast'

-----
Epoch 80, Loss: 0.1034
테스트 번역: '나는 밥을 먹는다' -> 'I have breakfast'

-----
최종 번역 테스트:
'나는 밥을 먹는다' -> 'I have breakfast'
'나는 물을 마신다' -> 'I drink water'
'나는 책을 읽는다' -> 'I read a book'
```

Chapter 5

Code Implement

2 GPT 2

Pre-LN / Final LN

```
# GPT-2 스타일의 Transformer Block (Pre-LN 구조)
class GPT2Block(nn.Module):
    def __init__(self, d_model: int, n_heads: int, d_ff: int, dropout: float = 0.1):
        super().__init__()
        self.ln1 = nn.LayerNorm(d_model)
        self.ln2 = nn.LayerNorm(d_model)
        self.attn = GPT2MultiHeadAttention(d_model, n_heads, dropout)

        # GPT-2 스타일 피드포워드 (GELU 활성화 함수 사용)
        self.mlp = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(), # GPT-2는 GELU 사용
            nn.Linear(d_ff, d_model),
            nn.Dropout(dropout)
        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, x, causal_mask=None):
        # Pre-LN 구조 (GPT-2 스타일)
        # Self-attention
        norm_x = self.ln1(x)
        attn_out, _ = self.attn(norm_x, causal_mask)
        x = x + self.dropout(attn_out)

        # Feed-forward
        norm_x = self.ln2(x)
        ff_out = self.mlp(norm_x)
        x = x + self.dropout(ff_out)

    return x
```

GPT 2 Model

```
class GPT2(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config

        self.wte = nn.Embedding(config.vocab_size, config.d_model) # Word Token Embedding
        self.wpe = GPT2PositionalEncoding(config.d_model, config.max_seq_len) # Word Position Embedding
        self.h = nn.ModuleList([
            GPT2Block(config.d_model, config.n_heads, config.d_ff, config.dropout)
            for _ in range(config.n_layers)
        ])

        self.ln_f = nn.LayerNorm(config.d_model)
        self.lm_head = nn.Linear(config.d_model, config.vocab_size, bias=False)
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
        elif isinstance(module, nn.LayerNorm):
            torch.nn.init.zeros_(module.bias)
            torch.nn.init.ones_(module.weight)

    def create_causal_mask(self, seq_len):
        """Causal mask 생성 (GPT 스타일)"""
        mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1).bool()
        return mask.to(DEVICE)

    def forward(self, input_ids, attention_mask=None):
        seq_len = input_ids.size(1)
        x = self.wte(input_ids) + self.wpe(input_ids)
        causal_mask = self.create_causal_mask(seq_len)
        for block in self.h:
            x = block(x, causal_mask)
        x = self.ln_f(x)
        logits = self.lm_head(x)

    return logits
```

Chapter 5

Code Implement

Sparse Attention

3 GPT 3

```
class GPT3MultiHeadAttention(nn.Module):
    def __init__(self, d_model: int, n_heads: int, dropout: float = 0.0):
        super().__init__()
        assert d_model % n_heads == 0

        self.d_model = d_model
        self.n_heads = n_heads
        self.d_k = d_model // n_heads

        self.c_attn = nn.Linear(d_model, 3 * d_model)
        self.c_proj = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, causal_mask=None):
        batch_size, seq_len = x.size(0), x.size(1)
        qkv = self.c_attn(x)
        q, k, v = qkv.split(self.d_model, dim=2)
        q = q.view(batch_size, seq_len, self.n_heads, self.d_k).transpose(1, 2)
        k = k.view(batch_size, seq_len, self.n_heads, self.d_k).transpose(1, 2)
        v = v.view(batch_size, seq_len, self.n_heads, self.d_k).transpose(1, 2)
        scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(self.d_k)
        if causal_mask is not None:
            scores = scores.masked_fill(causal_mask == 0, -1e9)
        attention_weights = F.softmax(scores, dim=-1)
        attention_weights = self.dropout(attention_weights)
        attention_output = torch.matmul(attention_weights, v)
        attention_output = attention_output.transpose(1, 2).contiguous().view(
            batch_size, seq_len, self.d_model)
        output = self.c_proj(attention_output)

    return output, attention_weights
```

GPT 3 Model

```
class GPT3(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.wte = nn.Embedding(config.vocab_size, config.d_model)
        self.wpe = GPT3PositionalEncoding(config.d_model, config.max_seq_len)
        self.h = nn.ModuleList([
            GPT3Block(config.d_model, config.n_heads, config.d_ff, config.dropout)
            for _ in range(config.n_layers)
        ])
        self.ln_f = nn.LayerNorm(config.d_model)
        self.lm_head = nn.Linear(config.d_model, config.vocab_size, bias=False)
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
        elif isinstance(module, nn.LayerNorm):
            torch.nn.init.zeros_(module.bias)
            torch.nn.init.ones_(module.weight)

    def create_causal_mask(self, seq_len):
        mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1).bool()
        return mask.to(DEVICE)

    def forward(self, input_ids, attention_mask=None):
        seq_len = input_ids.size(1)
        x = self.wte(input_ids) + self.wpe(input_ids)
        causal_mask = self.create_causal_mask(seq_len)
        for block in self.h:
            x = block(x, causal_mask)
        x = self.ln_f(x)
        logits = self.lm_head(x)

    return logits
```

Chapter 5

Code Implement

BERT style Multi-head Attention

2 GPT 2

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model: int, n_heads: int, dropout: float = 0.1):
        super().__init__()
        assert d_model % n_heads == 0

        self.d_model = d_model
        self.n_heads = n_heads
        self.d_k = d_model // n_heads

        self.w_q = nn.Linear(d_model, d_model)
        self.w_k = nn.Linear(d_model, d_model)
        self.w_v = nn.Linear(d_model, d_model)
        self.w_o = nn.Linear(d_model, d_model)

        self.dropout = nn.Dropout(dropout)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        attention_weights = F.softmax(scores, dim=-1)
        attention_weights = self.dropout(attention_weights)
        output = torch.matmul(attention_weights, V)
        return output, attention_weights

    def forward(self, query, key, value, mask=None):
        batch_size = query.size(0)
        Q = self.w_q(query).view(batch_size, -1, self.n_heads, self.d_k).transpose(1, 2)
        K = self.w_k(key).view(batch_size, -1, self.n_heads, self.d_k).transpose(1, 2)
        V = self.w_v(value).view(batch_size, -1, self.n_heads, self.d_k).transpose(1, 2)
        attention_output, attention_weights = self.scaled_dot_product_attention(Q, K, V, mask)
        attention_output = attention_output.transpose(1, 2).contiguous().view(
            batch_size, -1, self.d_model)
        output = self.w_o(attention_output)

        return output, attention_weights
```

BERT Model

```
class BertModel(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.embeddings = BertEmbeddings(config)
        self.encoder = BertEncoder(config)
        self.pooler = BertPooler(config)
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
            if module.bias is not None:
                module.bias.data.zero_()
        elif isinstance(module, nn.Embedding):
            module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
        elif isinstance(module, nn.LayerNorm):
            module.bias.data.zero_()
            module.weight.data.fill_(1.0)

    def get_input_embeddings(self):
        return self.embeddings.word_embeddings

    def set_input_embeddings(self, value):
        self.embeddings.word_embeddings = value

    def forward(self, input_ids, attention_mask=None, token_type_ids=None, position_ids=None):
        if attention_mask is None:
            attention_mask = torch.ones_like(input_ids)

        # 어텐션 마스크 확장
        extended_attention_mask = attention_mask[:, None, None, :]
        extended_attention_mask = extended_attention_mask.to(dtype=torch.float32)
        extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0

        embedding_output = self.embeddings(input_ids, token_type_ids, position_ids)
        encoder_outputs = self.encoder(embedding_output, extended_attention_mask)
        pooled_output = self.pooler(encoder_outputs)

        return encoder_outputs, pooled_output
```

Chapter 5

Code Implement

4 BERT

```
BERT 모델 파라미터 수: 109,689,034
BERT 번역 훈련 시작...
Epoch 0, Loss: 2.3510
BERT 번역 결과: '나는 밥을 먹는다' -> 'I read a book'
-----
Epoch 1, Loss: 2.2386
BERT 번역 결과: '나는 밥을 먹는다' -> 'I listen to music'
-----
Epoch 2, Loss: 2.1416
BERT 번역 결과: '나는 밥을 먹는다' -> 'I exercise'

최종 BERT 번역 테스트:
'나는 밥을 먹는다' -> 'I exercise'
'나는 물을 마신다' -> 'I listen to music'
'나는 책을 읽는다' -> 'I exercise'
```

Paper Review

References

- Radford, Alec, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. "Improving Language Understanding by Generative Pre-Training." OpenAI Technical Report.
- Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. "Language Models Are Unsupervised Multitask Learners." OpenAI Technical Report.
- Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, et al. 2020. "Language Models Are Few-Shot Learners." Advances in Neural Information Processing Systems 33 (NeurIPS 2020).
- Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT 2019), 4171–4186. Minneapolis, MN: ACL.
- Nguyen, Toan Q., and Julian Salazar. 2019. "Transformers without Tears: Improving the Normalization of Self-Attention." In Proceedings of the 16th International Workshop on Spoken Language Translation (IWSLT 2019). -> Pre-LN 제시(제안) 논문

Paper Review

감사합니다.
QnA