Homework #1

이름과 학번을 기재해 주세요

- Name : 이준용
- Student Id : 202430026
- Submission date : 2024 / 11 / 15

In [1]:
```
# 이름과 학번을 기재해 주세요
NAME = "이준용"
ID = "202430026"
```

In [2]:
```
# NOTE - 해당 셀 수정 금지
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from torch.utils.data import Dataset, DataLoader
import random
import torch.optim as optim
import logging

logger = logging.getLogger(__name__)
file_handler = logging.FileHandler(f'hw2_{ID}_{NAME}.log')
file_handler.setLevel(logging.INFO)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)
logger.setLevel(logging.INFO)

logger.info('HELLO WORLD!!')
```

In [3]:
```
# NOTE - 해당 셀 수정 금지
def set_seed(seed):
    logger.info("Set seed")
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
SEED = 42
set_seed(SEED)
```

In [4]:
```
# NOTE - 해당 셀 수정 금지
logger.info("Q1")
```

Q1.1. 다중 조건부 확률 계산을 위한 함수 구현 (20점)

다음의 확률 이론들에 대한 설명을 참조하여, 어떤 사건 A와 조건들 B,C,...에 대해, `The Law of Total Probability`, `Chain rule`, `Bayes' Theorem`을 이용하여 다중 조건부 확률을 계산하는 함수를 구현합니다.

1. Chain Rule (체인 룰)

체인 룰은 여러 사건이 연속적으로 일어나는 확률을 각 사건의 조건부 확률로 분해해 계산할 때 사용됩니다. 체인 룰의 기본 형태는 아래와 같습니다.

- 예를 들어, 두 사건 A 와 B 에 대해 체인 룰은 다음과 같이 나타낼 수 있습니다.

$$P(A, B) = P(B \mid A) \cdot P(A)$$

- 여러 사건이 주어진 경우 체인 룰은 다음과 같이 확장됩니다.

$$P(A, B, C, D) = P(D \mid A, B, C) \cdot P(C \mid A, B) \cdot P(B \mid A) \cdot P(A)$$

---

2. The Law of Total Probability (전체확률의 법칙)

전확률의 법칙은 사건 B 의 확률을 조건부 확률로 분해하여 구할 수 있는 법칙입니다. 이 법칙은 다음과 같은 수식으로 나타낼 수 있습니다:

- 사건 A 와 사건 A 가 일어나지 않을 확률 not A 가 주어졌을 때,

$$P(B) = P(B \mid A) \cdot P(A) + P(B \mid \text{not A}) \cdot P(\text{not A})$$

wikipedia - 전체확률의 법칙

---

3. Bayes' Theorem (베이즈 정리)

베이즈 정리는 조건부 확률을 뒤집어 계산할 때 사용됩니다. 사건 B 가 주어졌을 때 사건 A 가 발생할 확률을 구할 수 있으며, 전체확률의 법칙을 바탕으로 계산됩니다.

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)}$$

$$\frac{P(B|A_1)P(A_1)}{P(B)} = \frac{P(B|A_1)P(A_1)}{P(B|A_1)P(A_1) + P(B|A_2)P(A_2)}$$

wikipedia - 베이즈 정리

---

4. conditional probability with multiple conditions (다중 조건부 확률)

다중 조건부 확률은 사건 A 가 여러 조건들 $B, C, D, \ldots$ 아래에서 발생할 확률을 의미합니다. 예를 들어, A 가 B 와 C 가 동시에 일어났을 때 발생할 확률 $P(A \mid B, C)$ 을 계산하는 것이 다중 조건부 확률 계산입니다.

이때 각 조건들이 순차적으로 주어졌다고 생각하고 체인 룰을 통해 확률을 분해하여 계산할 수 있습니다.

1. 먼저 $P(A \mid B, C)$ 와 같은 확률을 구하려고 할 때, 체인 룰을 이용하여 이를 분해할 수 있습니다.

$$P(A \mid B, C) = \frac{P(A, B, C)}{P(B, C)}$$

2. 체인 룰을 이용해 $P(A, B, C)$ 를 계산합니다.

$$P(A, B, C) = P(A \mid B, C) \cdot P(B \mid C) \cdot P(C)$$

3. 이렇게 계산한 확률들을 통해 전체 확률을 구하고, 다중 조건부 확률을 계산할 수 있습니다.

In [ ]:
```python
def chain_rule(probabilities):
    """
    TODO - 체인 룰을 사용하여 다중 조건부 확률을 계산
    - probabilities: 조건부 확률들의 리스트
    예: [P(A|B), P(B|C), P(C|D), ...]
    """
    joint_probability = 1.0
    for i in probabilities:
        joint_probability *=i

    return joint_probability

def law_of_total_probability(prior_A, likelihood_conditions_given_A, like
    """
    TODO - 전체 확률의 법칙을 사용하여 P(B, C, ...)를 계산
    - prior_A: 사건 A의 사전 확률
    - likelihood_conditions_given_A: 사건 A 하에서의 조건부 확률 리스트 [P(B|A),
    - likelihood_conditions_given_not_A: 사건 A' 하에서의 조건부 확률 리스트 [P(
    """
    # P(B, C, ... | A) 계산
    joint_given_A = chain_rule(likelihood_conditions_given_A)
    # P(B, C, ... | A') 계산
    joint_given_not_A = chain_rule(likelihood_conditions_given_not_A)
    # 전체 확률의 법칙 적용
    joint_likelihood = prior_A * joint_given_A +(1-prior_A)*joint_given_n
    return joint_likelihood

def bayes_theorem(prior_A, likelihood_conditions_given_A, joint_likelihoo
    """
    TODO - 베이즈 정리를 통해 다중 조건 하에서 조건부 확률을 계산
    - prior_A: 사건 A의 사전 확률
    - likelihood_conditions_given_A: 사건 A 하에서 조건부 확률 리스트 [P(B|A),
    - joint_likelihood_conditions: 다중 조건의 결합 확률 P(B, C, ...)
    """
    # 체인 룰을 이용해 P(B, C, ... | A)를 계산
    joint_given_A =chain_rule(likelihood_conditions_given_A)

    # 베이즈 정리를 적용해 조건부 확률 계산
    posterior = (joint_given_A*prior_A) / joint_likelihood_conditions

    return posterior
```

Q 1.2. (15점)

어떤 병에 걸릴 확률은 5%입니다. 특정 병이 있는 경우 증상 A, 증상 B, 증상 C가 각각 나타날 확률은 다음과 같습니다:

- 질병이 있을 때 : `P(A|질병)=0.8` , `P(B|A,질병)=0.7` , `P(C|B,A,질병)=0.6`
- 질병이 없을 때 : `P(A|질병없음)=0.1` , `P(B|A,질병없음)=0.2` , `P(C|B,A,질병없음)=0.3`

다음의 조건을 만족하는 `질병에 걸렸을 확률 : P(질병|A,B,C)` 를 직접 계산하고 풀이과정을 서술하세요.

- 어떤 확률이론을 '왜'썼는지 구체적으로 서술하셔야 합니다.
- 최종 결과는 소수점 아래 넷째 자리까지만 써주시면 됩니다.

A 1.2.

Q 1.3. (5점)

Q 1.2.의 문제를 **위에서 구현한 함수를 이용하여** 계산하세요.

In [7]:
```
prior_disease = 0.05
likelihood_conditions_given_disease = [0.8,0.7,0.6]
likelihood_conditions_given_no_disease = [0.1,0.2,0.3]
# 전체 확률의 법칙을 통해 P(A, B, C) 계산
joint_likelihood_conditions =law_of_total_probability(prior_disease,likel
# 베이즈 disease | a, b, c
posterior = bayes_theorem(prior_disease,likelihood_conditions_given_disea

print(f"P(질병|A, B, C) : {posterior:.4f}")
```

```
P(질병|A, B, C) : 0.7467
```

In [8]:
```
# NOTE - 해당 셀 수정 금지
logger.info("Q2")
```

Q 2.1. Simple GPT 모델 구현 (40점)

다음의 class 들의 빈칸을 채워 최종적으로 Simplified Generative Pre-trained Transformer (GPT) 모델을 구현하세요.

코드의 빈칸은 Q1부터 Q15까지 총 15개입니다.

In [9]:
```
# NOTE - Positional Encoding, 위치 인코딩을 통해 시퀀스의 위치 정보를 제공함
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        # NOTE - d_model은 트랜스포머 모델의 임베딩 차원 수
        super(PositionalEncoding, self).__init__()
        # TODO - Q1 : 위치 인코딩을 저장할 행렬 생성
        # positional_embedding_matrix = torch._____(_____, _____)
```

```
        positional_embedding_matrix = torch.zeros(max_len,d_model)

        # TODO - Q2 : 위치 벡터 생성
        # position = torch._____(_____, ______, dtype=torch.float)._____(
        position=torch.arange(0,max_len,dtype=torch.float).unsqueeze(1)

        # 각 차원에 대한 주기적인 함수 적용을 위한 스케일링 값 계산
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-np.l
        # TODO - Q3 : 짝수 인덱스에는 사인 함수 적용
        # positional_embedding_matrix[:, 0::2] = torch._____(_____ * div_
        positional_embedding_matrix[:,0::2]= torch.sin(position*div_term)

        # TODO - Q3 : 홀수 인덱스에는 코사인 함수 적용
        # positional_embedding_matrix[:, 1::2] = torch._____(_____ * div_
        positional_embedding_matrix[:,0::2]= torch.cos(position*div_term)

        # 배치 차원을 추가하여 모양 맞춤
        positional_embedding_matrix = positional_embedding_matrix.unsquee

        # 모델이 학습하지 않도록 고정된 상태로 위치 인코딩 저장
        self.register_buffer('positional_embedding_matrix', positional_em

    def forward(self, x):
        # TODO - Q4 :입력 텐서에 위치 인코딩 더하기
        x = x + self.positional_embedding_matrix[:, :x.shape[1], :]
        return x
```

In [21]:

```
# NOTE - Multi-Head Attention, 멀티 헤드 어텐션을 통해 여러 어텐션을 병렬로 수행하여
import math
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        # NOTE - d_model은 트랜스포머 모델의 임베딩 차원 수
        # NOTE - num_heads는 어텐션 헤드의 수
        super(MultiHeadAttention, self).__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by nu
        self.d_model = d_model
        self.num_heads = num_heads
        # TODO - Q4 : self.d_k는 멀티헤드 어텐션에서 각 어텐션 헤드의 차원 수
        # self.d_k = _____
        self.d_k =int(d_model/num_heads)

        # TODO - Q5 : 가중치 행렬 초기화
        # self.W_q = nn._____(d_model, d_model)  # Query 변환
        # self.W_k = nn._____(d_model, d_model)  # Key 변환
        # self.W_v = nn._____(d_model, d_model)  # Value 변환
        # self.W_o = nn._____(d_model, d_model)  # 출력 변환
        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)

    def forward(self, Q, K, V, mask=None):
        batch_size = Q.size(0)

        # 1. 선형 변환과 헤드 분할
        # TODO - Q6 : (batch_size, seq_length, d_model) -> (batch_size, n
        # Q = self.W_q(Q).view(batch_size, -1, self.num_heads, self.d_k).
        # K = self.W_k(K).view(batch_size, -1, self.num_heads, self.d_k).
        # V = self.W_v(V).view(batch_size, -1, self.num_heads, self.d_k).
        Q = self.W_q(Q).view(batch_size, -1, self.num_heads, self.d_k).pe
```

```
        K = self.W_k(K).view(batch_size, -1, self.num_heads, self.d_k).pe
        V = self.W_v(V).view(batch_size, -1, self.num_heads, self.d_k).pe

        # TODO - Q7 :2. Scaled Dot-product 어텐션 스코어 계산
        # scores = torch._____(Q, K.transpose(-2, -1)) / _____
        scores=torch.matmul(Q,K.transpose(-2,-1)) /math.sqrt(self.d_k)
        # 3. 마스킹 적용 (필요한 경우)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        # 4. 소프트맥스 적용하여 어텐션 가중치 계산
        attention = F.softmax(scores, dim=-1)

        # TODO - Q8 : 5. Value와 가중치를 곱하여 최종 출력 계산
        out = torch.matmul(attention,V)

        # 6. 헤드 연결과 원래 형태로 변환
        out = out.transpose(1, 2).contiguous().view(batch_size, -1, self.
        return self.W_o(out)
```

In [22]:
```
# NOTE - Transformer Decoder Block, 트랜스포머 디코더 블록, 멀티 헤드 어텐션과 피드
class TransformerBlock(nn.Module):
    def __init__(self, d_model, num_heads):
        super(TransformerBlock, self).__init__()
        # 멀티 헤드 어텐션 레이어
        self.attention = MultiHeadAttention(d_model, num_heads)
        # TODO - Q9 : 레이어 정규화
        # self.norm1 = _____
        self.norm1=nn.LayerNorm(d_model)
        # self.norm2 = _____
        self.norm2 = nn.LayerNorm(d_model)

        # TODO - Q10 : 피드 포워드 네트워크
        self.feed_forward = nn.Sequential(
            nn.Linear(d_model, d_model * 4),  # 확장
            nn.ReLU() ,
            nn.Linear(d_model * 4, d_model)   # 압축
        )

    def forward(self, x, mask=None):
        # 1. 멀티 헤드 어텐션 + 잔차 연결과 정규화
        attention = self.attention(x, x, x, mask)
        x = self.norm1(x + attention)
        # TODO - Q11 : 2. 피드 포워드 + 잔차 연결과 정규화
        # ff = self._____
        ff=self.feed_forward(x )
        x = self.norm2(x + ff)
        return x
```

In [23]:
```
# GPT 단순화 버전
class GPT(nn.Module):
    def __init__(self, vocab_size, d_model, num_heads, d_ff, num_layers,
        super(GPT, self).__init__()
        # TODO - Q12 : 입력 토큰에 대한 임베딩
        # self.embedding = _____
        self.embedding=nn.Embedding(vocab_size, d_model)
        # 위치 인코딩 추가
        self.positional_encoding = PositionalEncoding(d_model, max_len)
        # 단순화된 트랜스포머 디코더 블록 (num_layers 개 사용)
```

```
        self.layers = nn.ModuleList([TransformerBlock(d_model, num_heads)
        # TODO - Q13 : 출력층 정의 (다음 단어 예측)
        # self.fc_out = _____
        self.fc_out = nn.Linear(d_model, vocab_size)

    def forward(self, x):
        # 임베딩과 위치 인코딩을 입력에 적용
        x = self.embedding(x)
        x = self.positional_encoding(x)
        # 트랜스포머 블록 통과시킴
        for layer in self.layers:
            x = layer(x)
        # 마지막으로 출력층을 통해 다음 단어에 대한 로짓 계산
        logits = self.fc_out(x)
        return logits
```

In [24]:
```
# HACK - Hyperparameters, 모델의 임의의 하이퍼파라미터 설정
vocab_size = 1000  # 임의의 vocab size
d_model = 128
num_heads = 4  # num_heads 값을 설정
d_ff = 256  # 피드 포워드 네트워크의 차원 수 설정
num_layers = 1  # 레이어 수
max_len = 50
epochs = 70
learning_rate = 0.001  # 학습률 설정
```

In [25]:
```
# HACK - 데이터 예시 (간단한 토큰 시퀀스)
data = [
    random.sample(range(vocab_size), 10)
    for _ in range(1000)
    ]

# 데이터셋 준비
# 입력 토큰 시퀀스를 텐서로 변환
tokens = torch.tensor(data, dtype=torch.long)
# 라벨도 동일하게 입력과 같은 데이터로 사용 (다음 단어 예측을 위해)
labels = torch.tensor(data, dtype=torch.long)

# 모델 초기화
model = GPT(vocab_size, d_model, num_heads, d_ff, num_layers, max_len)
```

In [27]:
```
# 손실 함수 정의 (크로스 엔트로피 손실)
criterion = nn.CrossEntropyLoss()
# 옵티마이저 정의 (Adam 옵티마이저 사용)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# 모델 훈련
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()  # 기울기 초기화
    output = model(tokens)  # 모델에 입력 데이터 전달
    # 손실 계산 (출력과 라벨을 비교)
    loss = criterion(output.view(-1, vocab_size), labels.view(-1))
    # TODO - Q14 : 역전파를 통해 기울기 계산
    loss.backward
    optimizer.step()  # 옵티마이저로 파라미터 업데이트
    if epoch%10 == 0:
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

```
print("Training finished!")
```

```
Epoch [1/70], Loss: 7.0626
Epoch [11/70], Loss: 7.0626
Epoch [21/70], Loss: 7.0626
Epoch [31/70], Loss: 7.0626
Epoch [41/70], Loss: 7.0626
Epoch [51/70], Loss: 7.0626
Epoch [61/70], Loss: 7.0626
Training finished!
```

In [30]:
```
# 텍스트 생성 예시
vocab = {str(i): i for i in range(vocab_size)}
vocab_reverse = {i: str(i) for i in range(vocab_size)}

# NOTE - 간단한 텍스트 생성 함수 호출
def generate_text(model, start_text, vocab, vocab_reverse, max_length=100
    """텍스트 생성 함수"""
    model.eval()
    current_text = start_text
    # 시작 텍스트를 인코딩
    encoded = torch.tensor([[vocab[char] for char in start_text.split()]]

    with torch.no_grad():
        for _ in range(max_length):
            # 현재 시퀀스에 대한 예측
            output = model(encoded)
            # temperature sampling을 위한 logits 조정
            logits = output[0, -1] / temperature
            # 확률 분포 계산
            probs = F.softmax(logits, dim=-1)
            # 다음 토큰 샘플링
            next_char_idx = torch.multinomial(probs, 1, replacement=True)
            # TODO - Q15 : 생성된 텍스트에 새 문자 추가
            current_text += " " + vocab_reverse[next_char_idx]
            # 입력 시퀀스 업데이트
            encoded = torch.cat([encoded[:, 1:], torch.tensor([[next_char

    return current_text
```

In [31]:
```
# NOTE - 랜덤 출력 발생, 토큰 생성여부만 판단
start_text = "1 5 8 3 3"
generated_text = generate_text(model, start_text, vocab, vocab_reverse, m
print("Generated text : ", generated_text)
```

```
Generated text :  1 5 8 3 3 766 517 933 340 748 776 197 242 446 839 805 57
2 436 139 782 893 416 608 775 114 306 47 334 881 228 469 251 193 943 17
```

Q 2.2. (10점)

MultiHeadAttention 클래스에서 사용되는 Q(Query), K(Key), V(Value)의 역할을 각각 설명하세요.

특히, 각 행렬이 멀티 헤드 어텐션에서 입력 간의 관계를 계산하는 데 어떻게 기여하는지 서술해야 합니다.

batch_size, seq_length, d_model는 배치크기, 입력시퀀스의 길이, transformer 모델의 임베팅 차원을 가르키고 또한 num_heads와 dk는 어텐션 헤드수와 각 헤드에서 사용되는 차원을 의미한다.

입력 텐서 x 의 차원은 (batch_size, seq_length, d_model) 인데, 입력텐서는 각 단어에 대해 d_model 차원 임베딩을 가집니다.

Q(query) 는 Q=x*Wq(q가중치행렬)로 계산이되고 이때 차원은 (batch_size, seq_length, d_model)입니다.

K(key), V(value) 모두 x에 가중치행렬을 곱했고 차원역시 모두 동일합니다.

멀티 헤드 어텐션에서는 num_heads라는 하이퍼파라미터를 정의했는데 q,k,v를 num_heads의 개수로 나누어 각 헤드에 대해 병렬로 어텐션을 진행합니다.

Q = self.W_q(Q).view(batch_size, -1, self.num_heads, self.d_k).permute(0, 2, 1, 3) K = self.W_k(K).view(batch_size, -1, self.num_heads, self.d_k).permute(0, 2, 1, 3) V = self.W_v(V).view(batch_size, -1, self.num_heads, self.d_k).permute(0, 2, 1, 3)

그것이 위코드입니다. 이렇게 변환하면 q의 차원은(batch_size,num_heads,seq_length,dk)가 되고 k와 v에 대해서도 동일하게 변환하여 dk차원 으로 계산을 수행하게 됩니다.

이후에 scores를 계산하는 부분인 각헤드에 대해 어텐션을 계산하게 되는데 헤드 h에 대해 Q와 K의 내적을 구한다음 dk의 제곱근으로 나누어 스케일링합니다.(차원은 (batch_size,num_heads,seq_length,seq_length)이 됩니다.)

이때 Q와 K내적은 단어간의 유사도를 구하는 역할을 하게됩니다.

마지막으로 어텐션 가중치를 적용해서 즉 softmax를 적용해서 각 단어 간의 관계를 어텐션 가중치로 변환을 진행합니다. softmax(scores) * v 와 같은 식이 되는데 이때 가중치행렬에 v를 곱함으로써 각 위치에서 중요한 정보가 반영된 값을 얻게 됩니다.

최종 output 텐서의 차원은 (batch_size,num_heads,seq_length,dk)이고 각 헤드에서 서로다른 정보를 반영한 값들이 return 됩니다.

모든 헤드의 output를 최종출력차원인 d_model로 되돌려서 (batch_size,num_heads,d_model)이 됩니다.

즉 정리하면 Q,K의 내적은 단어 간의 관계를 나타내고 이때 특정 단어가 다른 단어들과 얼마나 관련이 있는지 알려주고

V는 최종적으로 합쳐져서 입력 문맥의 유사도를 반영하는 역할을 합니다.

멀티헤드어텐션은 여러개 헤드를 통해 어텐션을 병렬로 계산하여 더 풍부한 문맥관계를 학습할수 있게 합니다.

A 2.2.

Q 2.3. (10점)

MultiHeadAttention에서 여러 개의 어텐션 헤드를 사용하는 이유와, 각 어텐션 헤드가 Q, K, V를 통해 서로 다른 관계를 학습할 수 있는 이유를 설명하세요.

멀티헤드어텐션에서 어려 개의 헤드를 병렬로 사용하는 이유는 다양한 문맥 학습을 할 수 있고 단어간의 관계에 대한 정보를 학습할 수 있게 합니다. 또한 헤드끼리는 독립적으로 학습되서 서로 다른 단어 문맥의 특징을 학습할 수 있게 됩니다.

각 헤드에서는 qkv행렬 계산할때 고유한 가중치 행렬인 wq wk wv를 사용함으로써 서로다른 관계와 구조적 특징을 학습함.

q와 k 내적하는 계산식에서 어텐션 스코어는 각 헤드마다 다른값은로 계산이 되어, 서로다른 가중치를 학습할 수 있게됩니다.

A 2.3.

Bonus (1점)

과제에 대한 피드백을 남겨주세요!

과제를 진행하면서 트랜스포머의 논문에 나오는 수식에 대해 직접 코딩을 함으로써 깊이 이해할 수 있게 되었으며, 매일 사용하는 gpt와 트랜스포머에 대해 입력한 텍스트가 어떤 과정을 거치고 계산이 되어 output으로 나오게 되는지 이해할 수 있게 되었습니다.

수고하셨습니다 :)
제출 전 셀을 모두 실행 후 제출 바랍니다!

In [32]:
```
# NOTE - 해당 셀 수정 금지
logger.info("END")
```

In [ ]:
```

```