

Lecture 4

Transformer



Seoul National University



Human Interface Laboratory

Contents

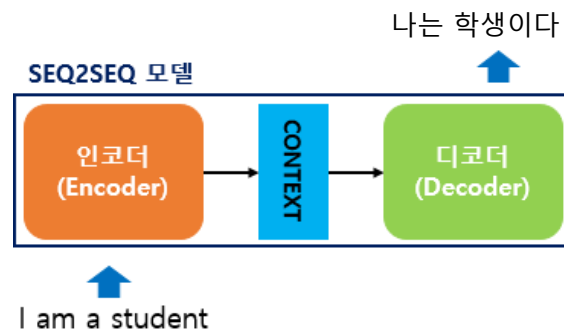
- **Intro**
 - Transformer 구조의 중요성
- **Transformer의 개념**
- **Transformer의 구현**
- **Pre-Trained Transformer 사용해보기**



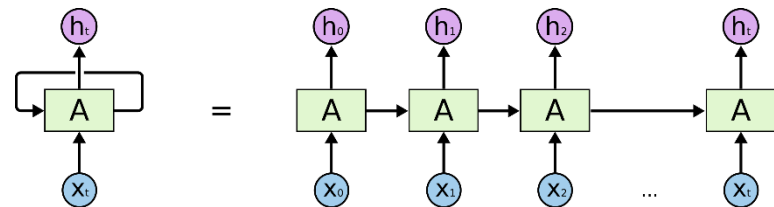
Intro

- **Model Architecture for Seq2Seq**

- Ex) 한국어 문장을 영어 문장으로 번역하는 AI 모델?
 - 영어 문장 -> 입력 벡터(인코딩)
 - 입력 벡터 - (AI 모델) -> 출력 벡터
 - 출력 벡터 -> 한국어 문장(디코딩)
- Seq2Seq: Input Sequence -> context vector -> Output Sequence
- RNN 문제: (특히) 긴 문장의 모델링이 어려움



<https://wikidocs.net/160053>



<https://wikidocs.net/160053>

Intro

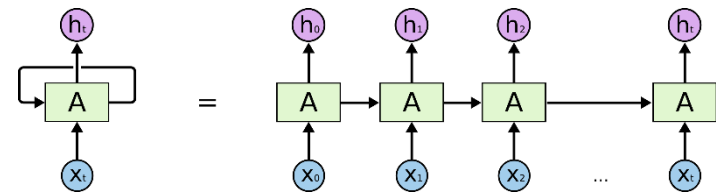
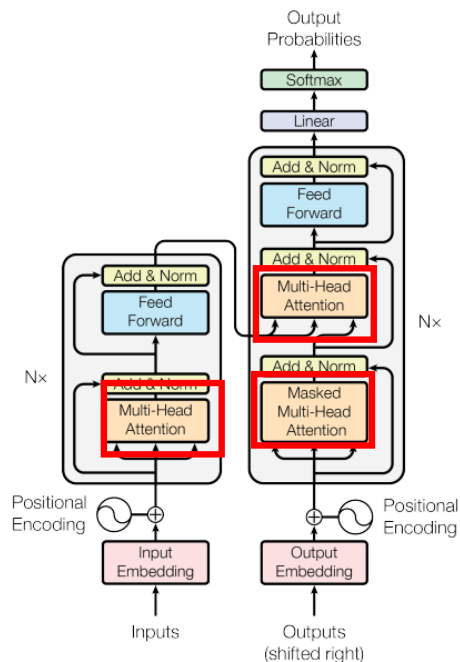
- **Transformer의 중요성**

- Transformer는 RNN 대비 대체로 성능이 높은 Sequence Model
- 최근 대부분의 생성모델에서 Transformer 구조를 활용

Transformer란

- Transformer란

- 성능이 뛰어난 Sequence Model
- Attention Mechanism이 핵심
 - Attention Mechanism을 통해 Input 정보를 보다 직접적으로 반영
 - cf) RNN: hidden state를 통해 Input 정보를 보다 간접적으로 반영



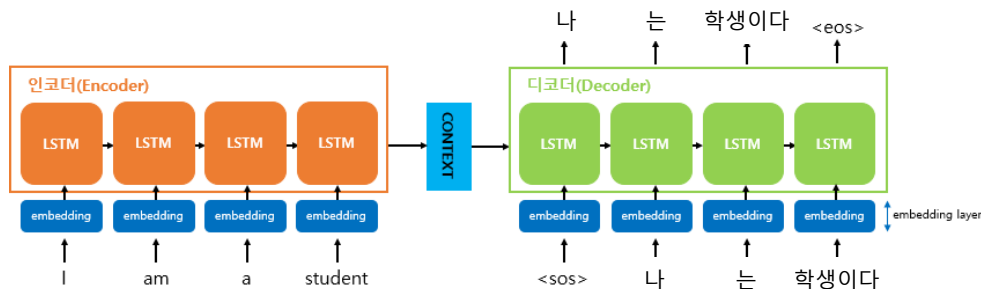
Vaswani, Ashish, et al. "Attention Is All You Need(NIPS)", 2017.

Transformer란

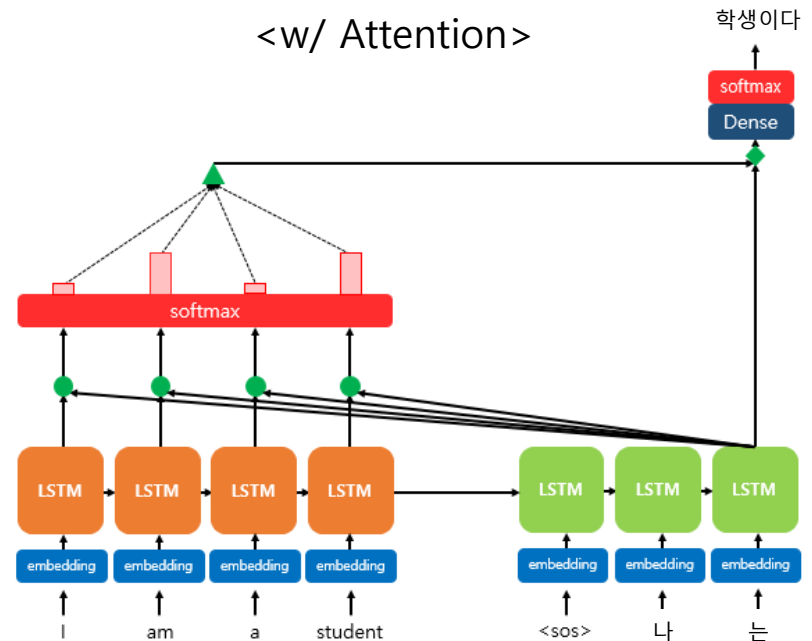
- **Attention**

- Input Sequence에 대한 context vector를 생성하는 메커니즘
- Sequence 각 Token간 연관성을 직접 반영하여 Input을 재조정
 - 모든 Token과의 연관성 반영 가능 => Long Sequence Modeling 용이

<w/o Attention>



<w/ Attention>



<https://wikidocs.net/22893>

Transformer란

- **Attention**

- context vector는 각 Input token vector의 가중합(중요도에 차등)
- 가중치는 Input sequence와 Target sequence 내 token간 연관도
 - Cross-Attention: Target Sequence가 Input Sequence와 다름
 - Self-Attention: Target Sequence가 Input Sequence 자기 자신임

$$Context = Weight \cdot Input$$

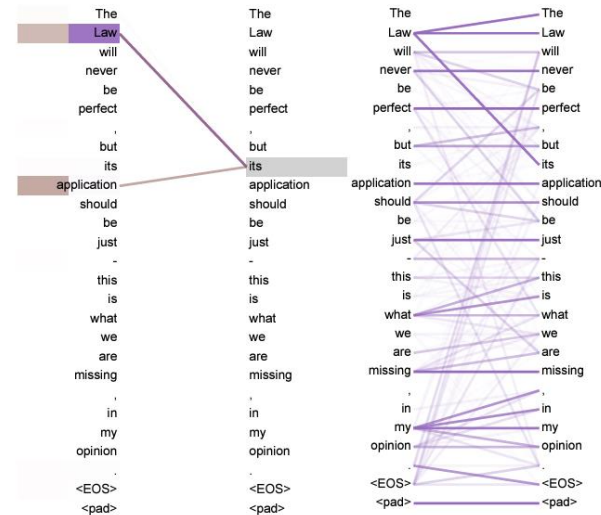
$$Weight = softmax\left(\frac{Score}{\sqrt{d_{model}}}\right)$$

$$Score = Target \cdot Input^T$$



$$Context = Attention(Q, K, V) = softmax\left(\frac{Q \cdot K^T}{\sqrt{d_{model}}}\right) V$$

where $Q = Target$, $K = V = Input$

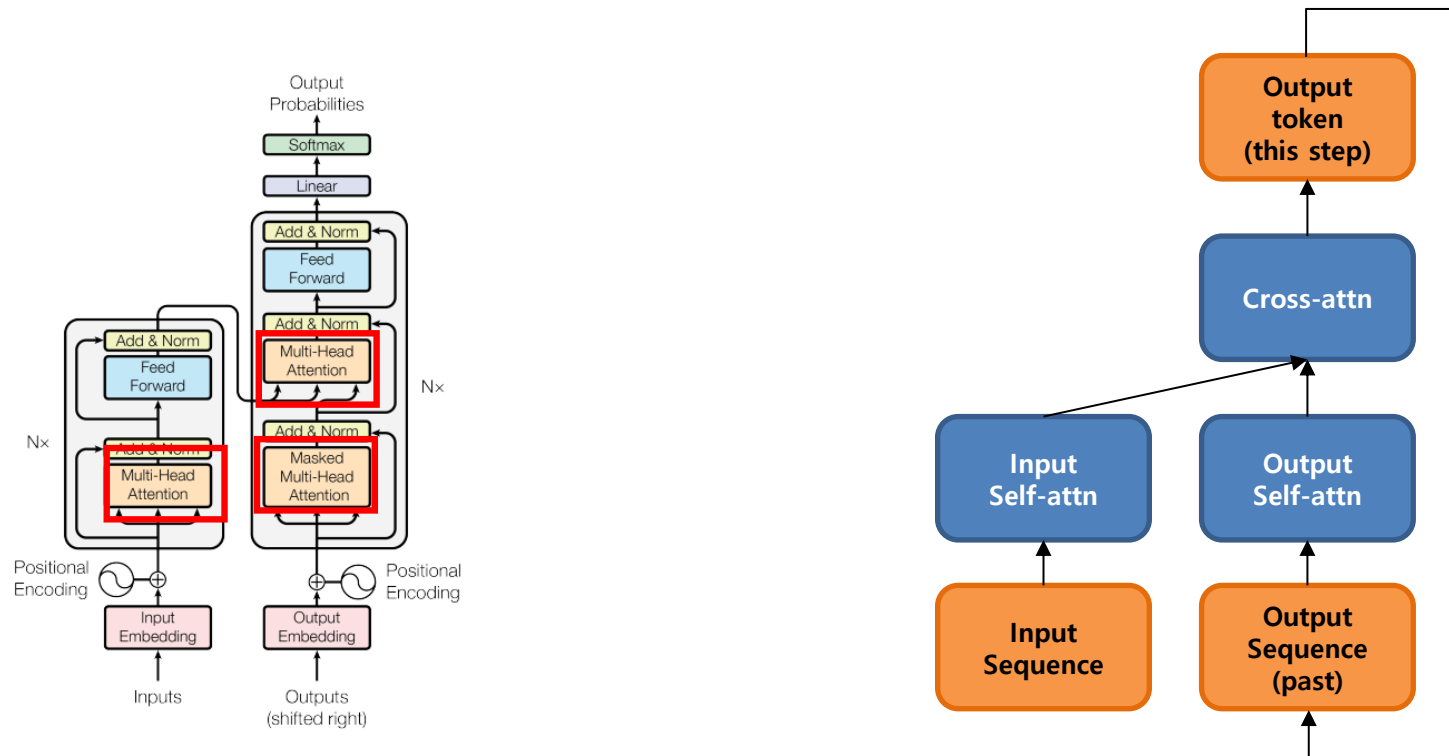


Vaswani, Ashish, et al. "Attention Is All You Need(NIPS)", 2017.

Transformer란

- Transformer Architecture

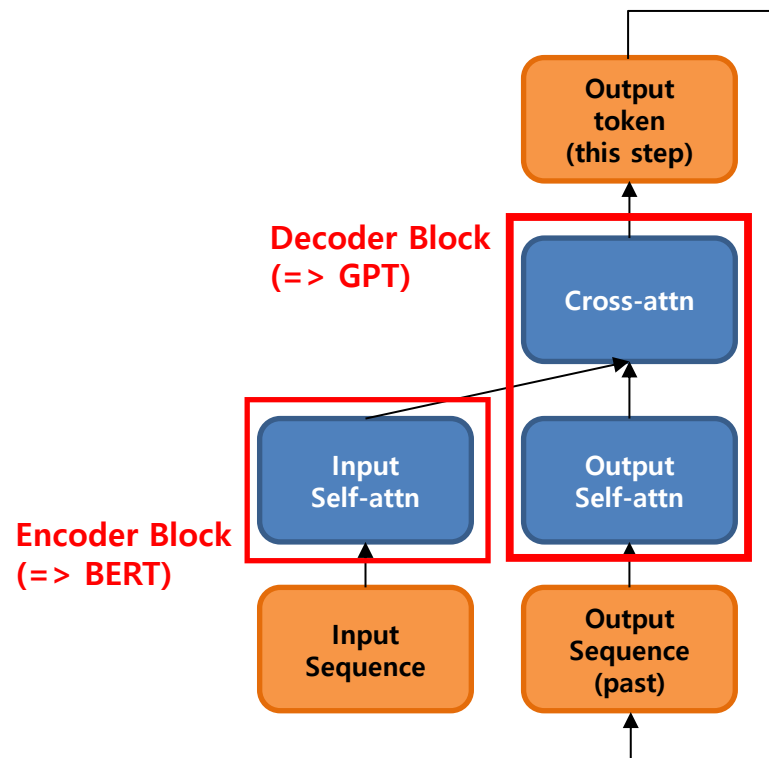
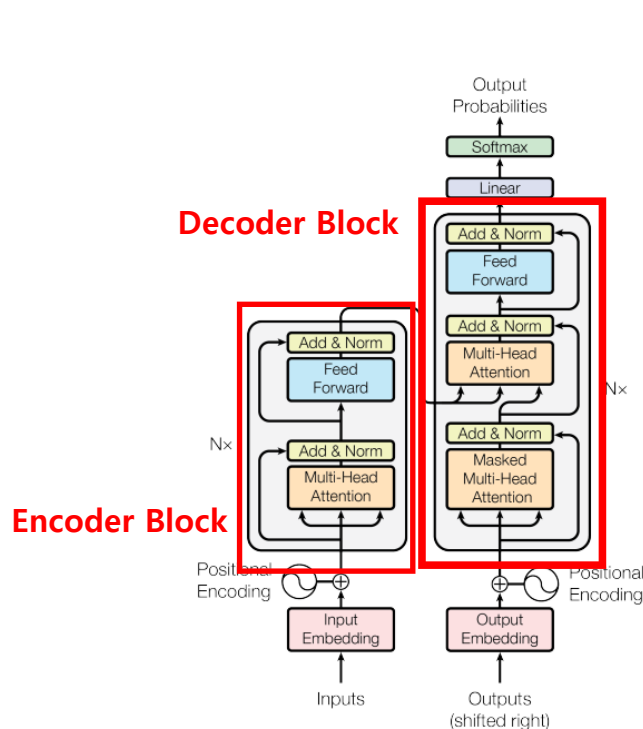
- t 번째 토큰 생성 시 세 종류의 Attention Mechanism 활용
 - Input Sequence에 대한 Self-Attention
 - Output Sequence($1 \sim (t - 1)$ 번째)에 대한 Self-Attention
 - Input context vector와 Output context vector에 대한 Cross-Attention



Transformer란

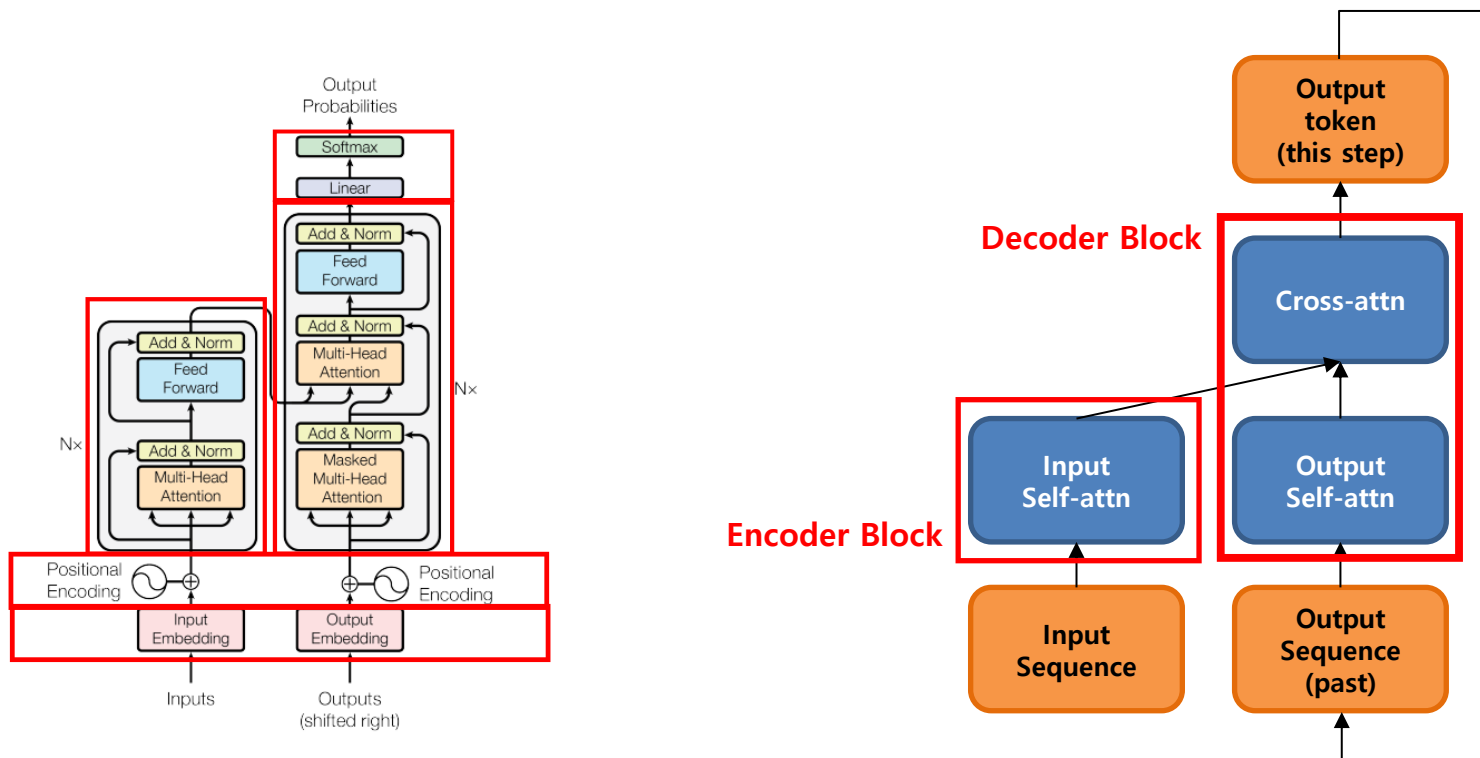
- Transformer Architecture

- 기본적으로 N개의 Encoder Block과 Decoder Block으로 구성됨
 - Encoder Block만 특화: ex. BERT(주어진 문장의 Embedding 생성)
 - Decoder Block만 특화: GPT(언어 생성)



Transformer란

- Implementation Details
 - Positional Encoding
 - Encoder Block(N개 반복)
 - Decoder Block(N개 반복)
 - Computing Output Probabilities



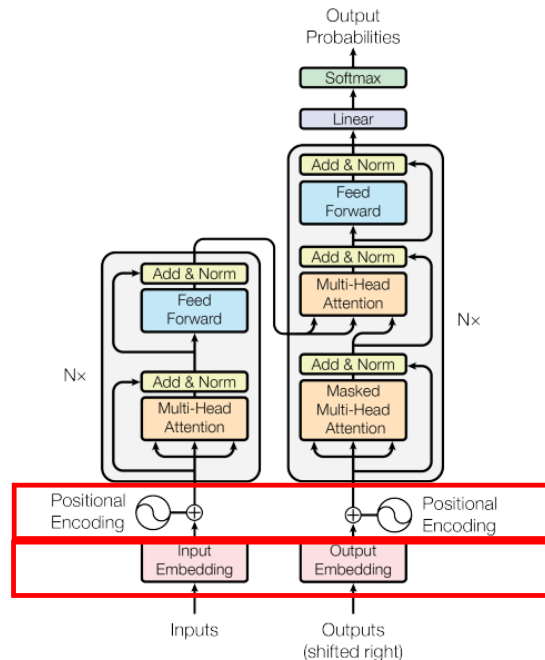
Transformer란

- **Input/Output Embedding**

- Sequence를 신경망이 연산할 수 있는 Embedding Vector로 변환

- **Positional Encoding**

- 토큰 간 순서 정보를 sequence에 반영
 - Attention: Input Sequence의 모든 token을 동시에 처리하므로 순서 정보 소실
 - cf) RNN: Input Sequence의 순서에 따라 hidden state 순차적 갱신, 순서 정보 반영
- Input Sequence에 Positional Encoding 값을 덧셈하는 방식으로 반영



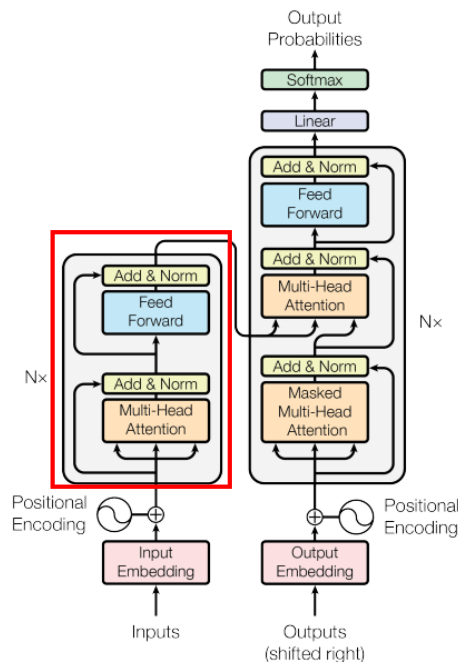
$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

Transformer란

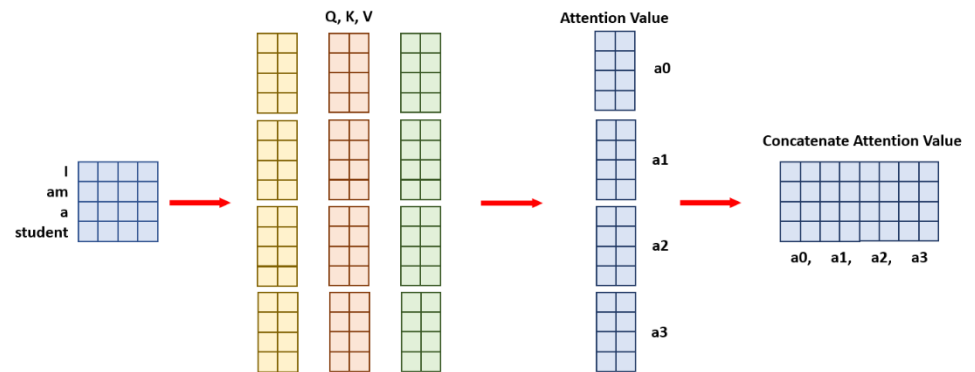
• Encoder Block

- Multi-Head Attention: 서로 다른 독립된 Attention을 수행한 뒤 이어붙임
 - 목적: 입력 데이터의 다양한 관계를 분담하여 학습
 - Ex. Head 1: 주어-목적어 관계, Head 2: 동사-목적어 관계...
 - 구현: Input sequence를 서로 다른 Linear Projection W_i 로 변환시킴
 - 서로 다른 Linear Projection에 의한 독립적인 Attention 결과를 이어붙임(Concat)



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

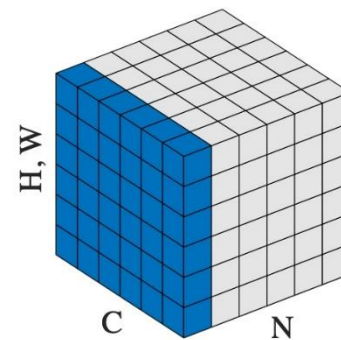
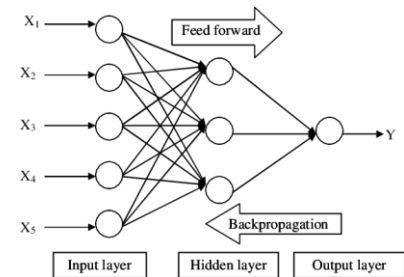
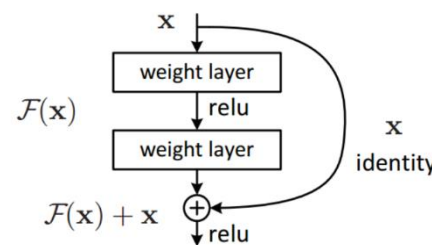
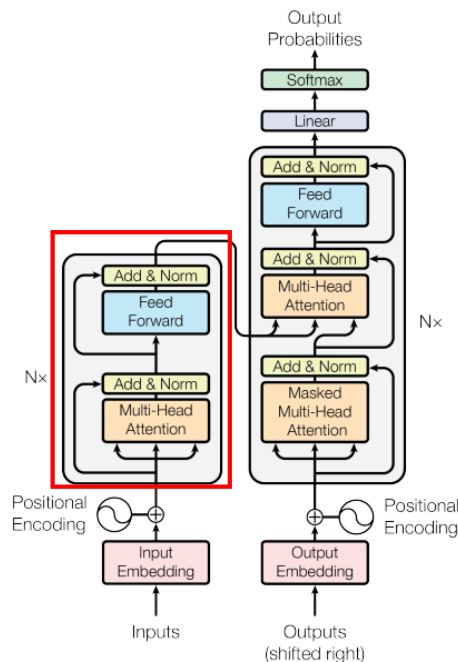


<https://codingopera.tistory.com/44>

Transformer란

- **Encoder Block**

- Skip Connection(Add): 원본 입력 정보를 그대로 전달해 중요 정보 소실 방지
- Feedforward: 벡터 크기 조절, 학습 표현력 강화
- Normalization: 입력 벡터의 평균과 분산을 조정해 학습 안정성 강화



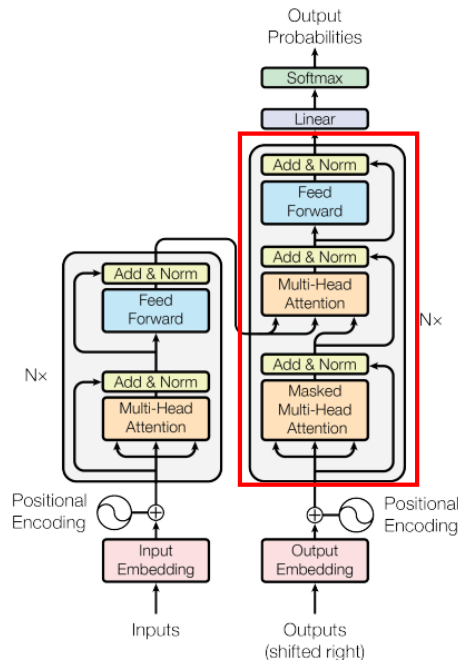
https://www.researchgate.net/figure/Architecture-of-a-typical-multilayer-feed-forward-neural-network_fig1_268380188

Transformer란

- **Decoder Block**

- **Masked Multi-Head Attention**

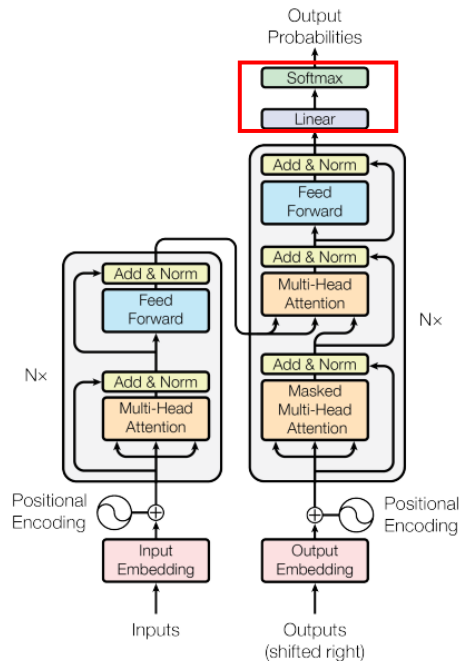
- Train 시 Output sequence에 대한 Self-Attention에 적용
 - t 번째 token을 생성할 경우, output sequence에서 $(t + 1)$ 번째부터의 토큰을 가림
 - Inference에서 t 번째 token 생성 중 미래를 볼 수 없으므로 Train 시에도 맞춰줌
 - Mask: Input sequence와 미래 시점 token 간 연관도를 0으로 설정하여 구현



<https://tigris-data-science.tistory.com/entry/%EC%B0%A8%EA%B7%BC%EC%B0%A8%EA%B7%BC-%EC%9D%B4%ED%95%B4%ED%95%98%EB%8A%94-Transformer4-Masked-Multi-Head-Attention%EA%B3%BC-Decoder>

Transformer란

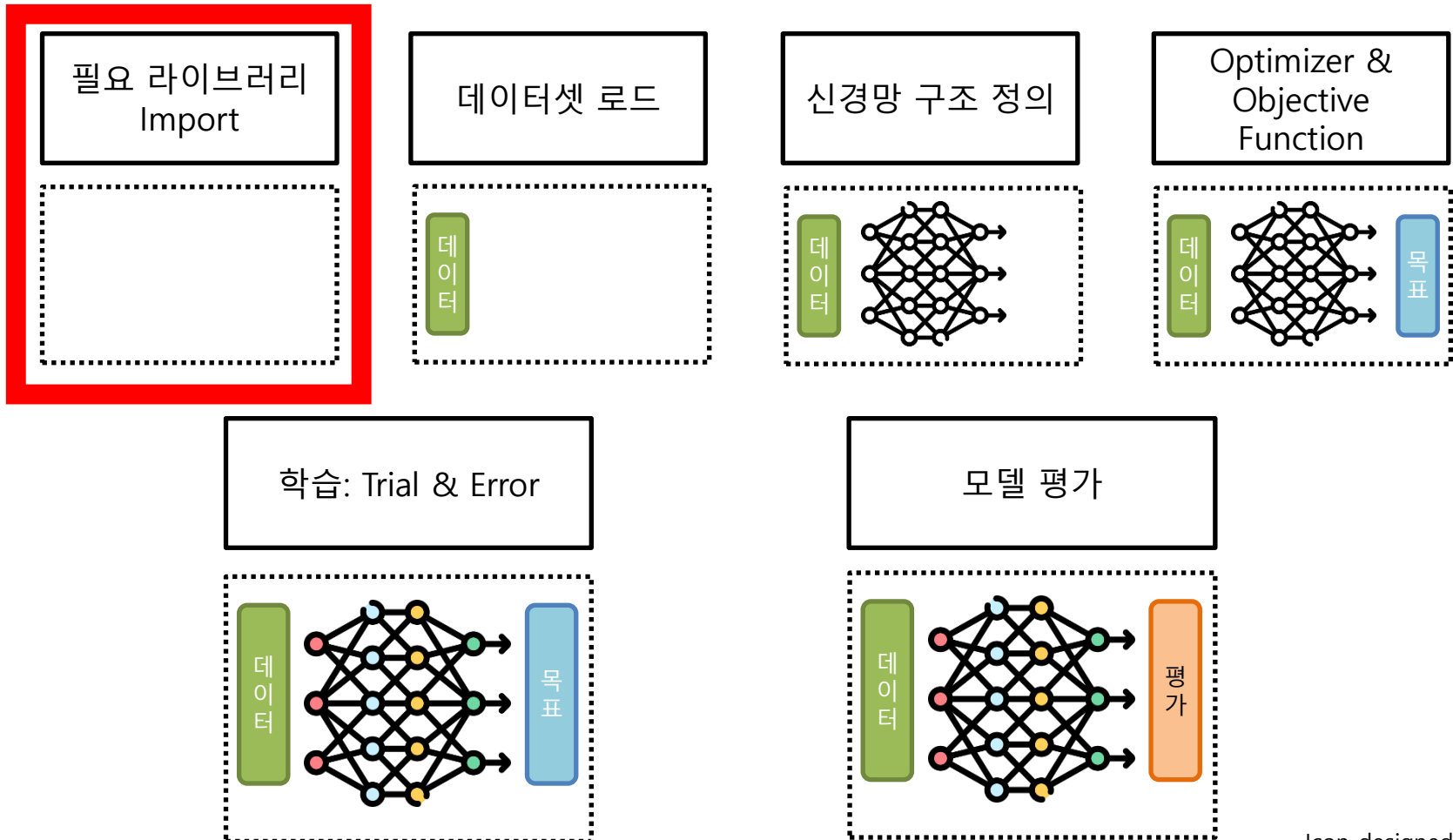
- **Compute Output probabilities**
 - Linear: 벡터 크기 조절
 - ex. 256차원 벡터를 알파벳 개수에 맞춰 26차원으로 변경
 - softmax: 주어진 벡터를 Discrete probability distribution으로 변환



$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^N e^{z_j}}$$

Transformer 구현

- (Recall) 딥러닝 코드의 대략적 구조



Icon designed by freepik

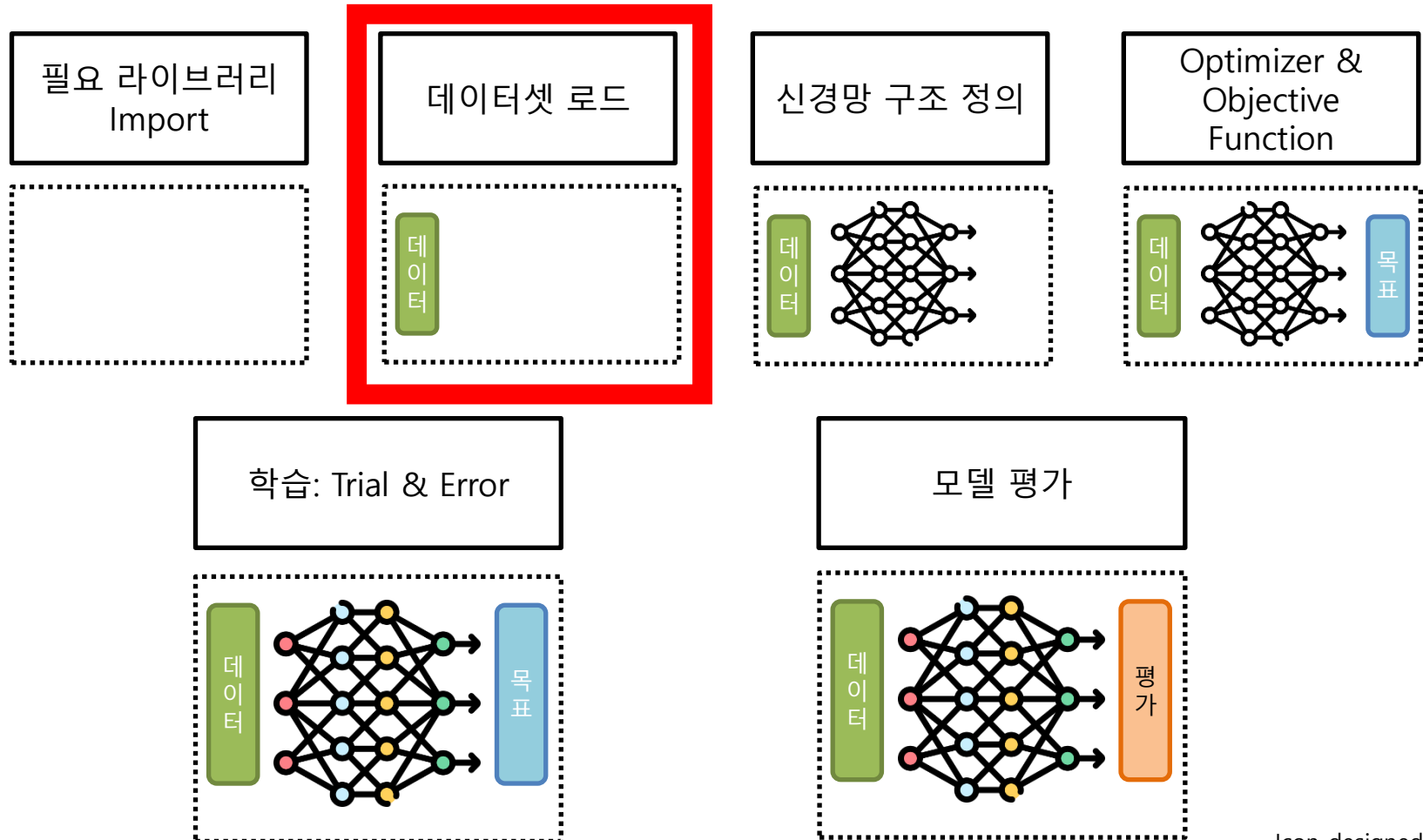
Transformer 구현

- 필요 라이브러리 Import

```
# %% Import Modules
import torch
import torch.nn as nn
import torch.nn.functional as F
import math
```

Transformer 구현

- (Recall) 딥러닝 코드의 대략적 구조



Icon designed by freepik

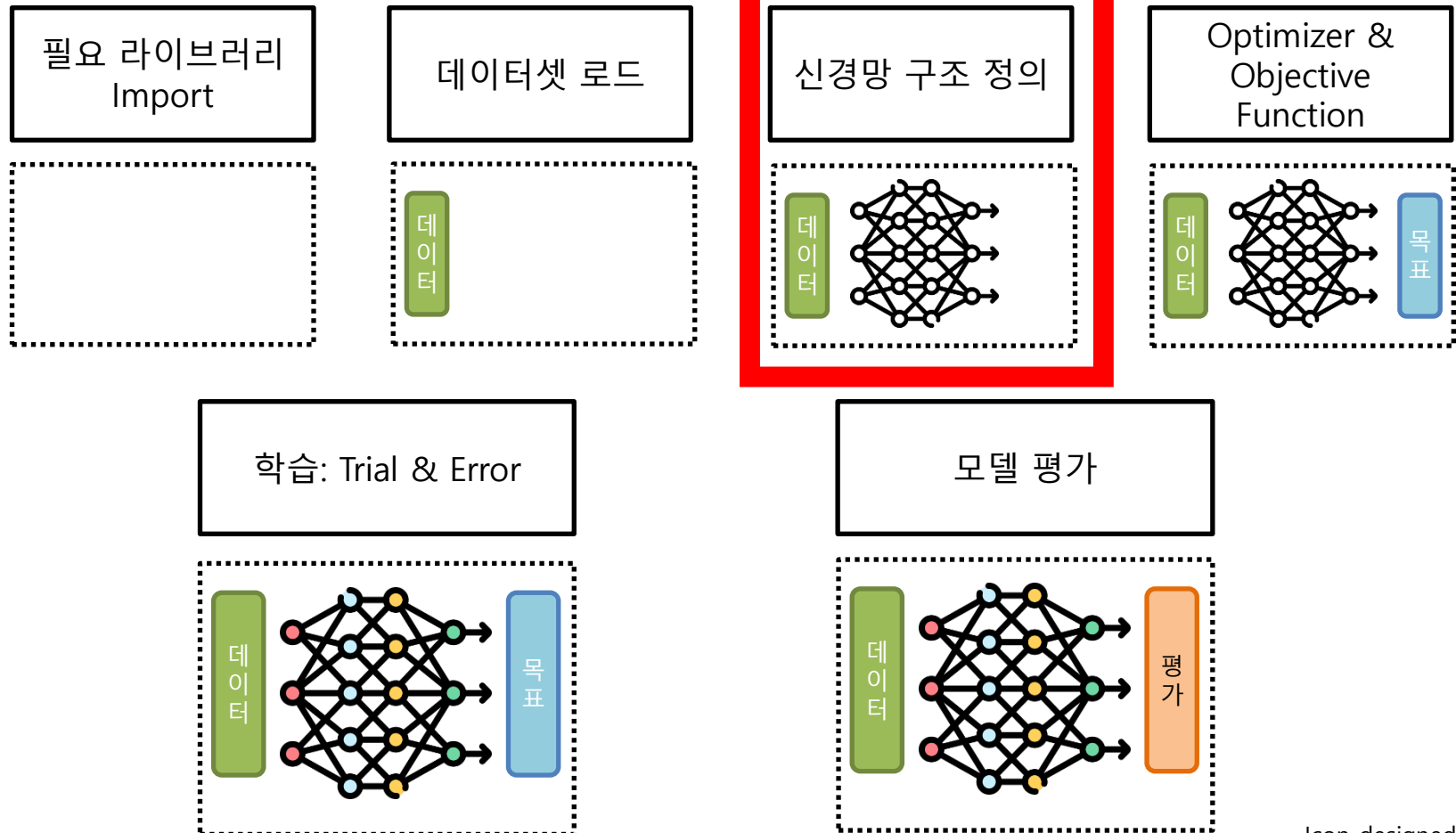
Transformer 구현

- 데이터셋 로드

```
# %% Dataset Preparation
vocab_size = 20
batch_size = 32
seq_len = 10
data_loader = [(torch.randint(0, vocab_size, (batch_size, seq_len)),
                    torch.randint(0, vocab_size, (batch_size, seq_len))) for _ in range(100)]
```

Transformer 구현

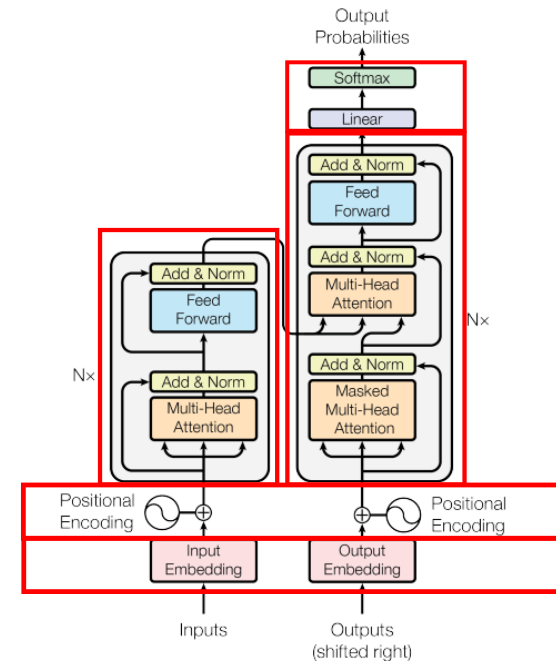
- (Recall) 딥러닝 코드의 대략적 구조



Icon designed by freepik

Transformer 구현

- Transformer 신경망 구조의 구현
 - Input/Output Sequence의 Embedding 획득
 - Positional Encoding
 - Encoder Block
 - Decoder Block
 - Computing Output Probabilities



Transformer 구현

- Transformer 신경망 구조의 구현
 - 전체적 구조

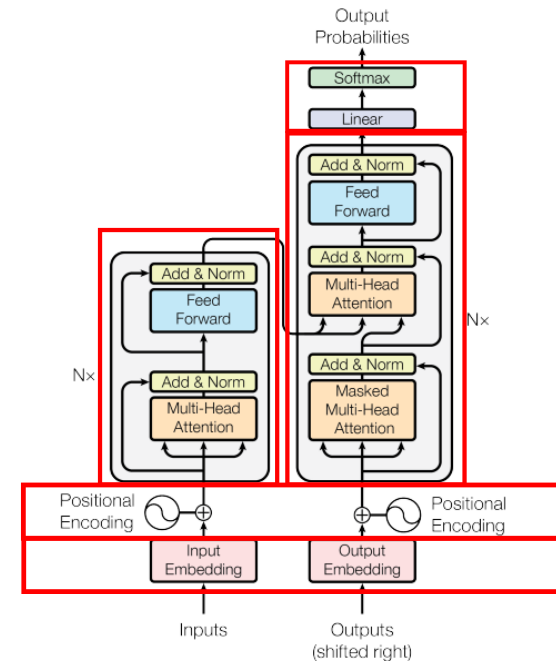
```
# %% Transformer Model
class Transformer(nn.Module):
    def __init__(self, input_dim, output_dim, d_model, n_head, d_ff, num_layers, dropout):
        super(Transformer, self).__init__()
        self.src_embedding = nn.Embedding(input_dim, d_model)
        self.tgt_embedding = nn.Embedding(output_dim, d_model)
        self.positional_encoding = PositionalEncoding(d_model)

        self.encoder_layers = nn.ModuleList([EncoderLayer(d_model, n_head, d_ff, dropout)
        for _ in range(num_layers)])
        self.decoder_layers = nn.ModuleList([DecoderLayer(d_model, n_head, d_ff, dropout)
        for _ in range(num_layers)])
        self.fc_out = nn.Linear(d_model, output_dim)

    def forward(self, src, tgt, tgt_mask):
        src = self.src_embedding(src)
        src = self.positional_encoding(src)
        for layer in self.encoder_layers:
            src = layer(src, None)

        tgt = self.tgt_embedding(tgt)
        tgt = self.positional_encoding(tgt)
        for layer in self.decoder_layers:
            tgt = layer(tgt, src, tgt_mask)

        return self.fc_out(tgt)
```



Transformer 구현

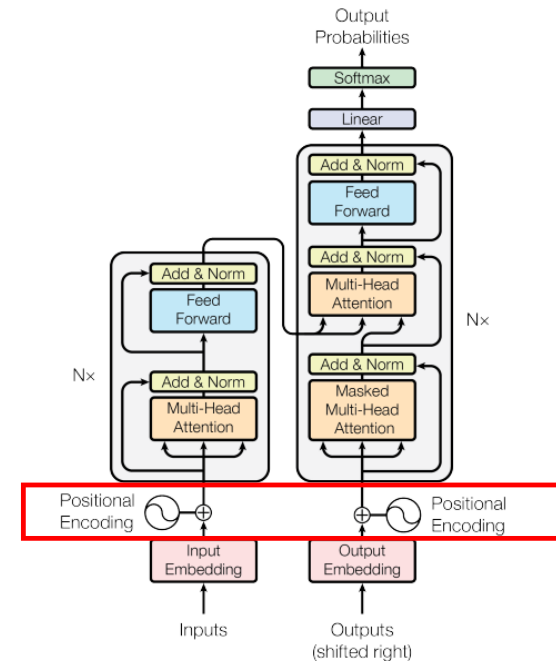
- Transformer 신경망 구조의 구현
 - Positional Encoding

$$PE_{(pos, 2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE_{(pos, 2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

```
# %% Positional Encoding
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super(PositionalEncoding, self).__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)
        self.register_buffer('pe', pe)

    def forward(self, x):
        return x + self.pe[:, :x.size(1), :]
```

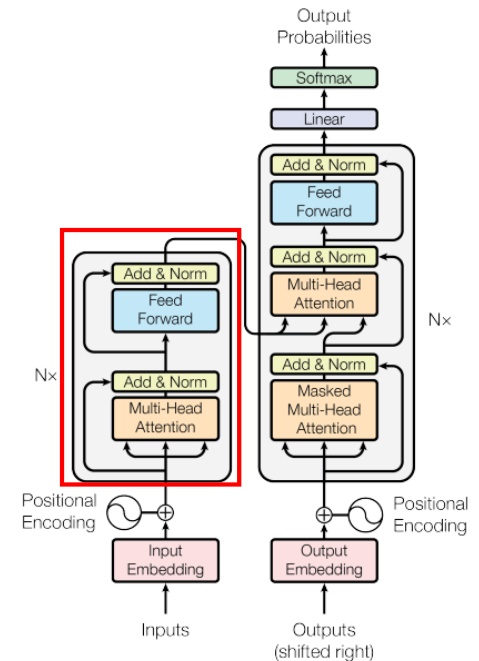


Transformer 구현

- Transformer 신경망 구조의 구현
 - Encoder Block의 전체적 구조

```
# %% Encoder Block
class EncoderBlock(nn.Module):
    def __init__(self, d_model, n_head, d_ff, dropout):
        super(EncoderBlock, self).__init__()
        self.attn = MultiHeadAttention(d_model, n_head)
        self.ffn = FeedForward(d_model, d_ff, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        x = self.norm1(x + self.dropout(self.attn(x, x, x, mask)))
        x = self.norm2(x + self.dropout(self.ffn(x)))
        return x
```



Transformer 구현

- Transformer 신경망 구조의 구현
 - Decoder Block의 전체적 구조

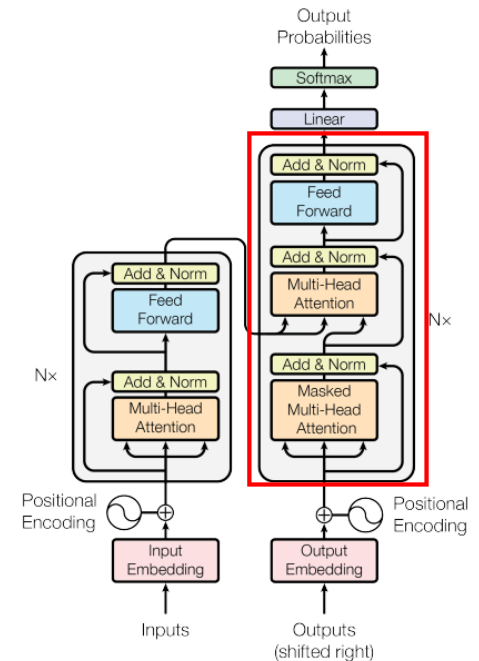
```
## %% Decoder Block
class DecoderBlock(nn.Module):
    def __init__(self, d_model, n_head, d_ff, dropout):
        super(DecoderBlock, self).__init__()

        self.self_attn = MultiHeadAttention(d_model, n_head)
        self.norm1 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout)

        self.cross_attn = MultiHeadAttention(d_model, n_head)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout2 = nn.Dropout(dropout)

        self.ffn = FeedForward(d_model, d_ff, dropout)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout3 = nn.Dropout(dropout)

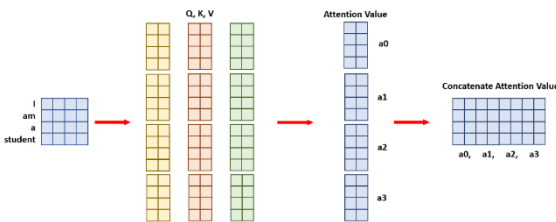
    def forward(self, tgt, memory, tgt_mask):
        tgt = self.norm1(tgt + self.dropout1(self.self_attn(tgt, tgt, tgt, tgt_mask)))
        tgt = self.norm2(tgt + self.dropout2(self.cross_attn(tgt, memory, memory)))
        tgt = self.norm3(tgt + self.dropout3(self.ffn(tgt)))
        return tgt
```



Transformer 구현

- Transformer 신경망 구조의 구현
 - Encoder/Decoder Block: Multi-Head Attention

$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$
 where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$



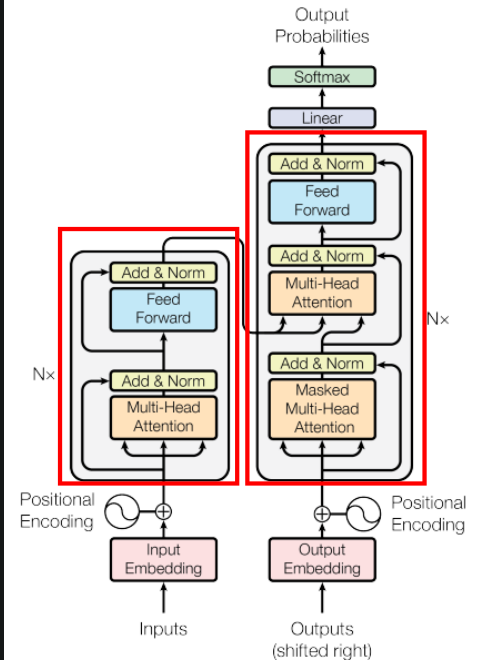
```

# %% Multi-Head Attention
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, n_head):
        super(MultiHeadAttention, self).__init__()
        self.n_head = n_head
        self.attention = ScaleDotProductAttention()
        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_concat = nn.Linear(d_model, d_model)

    def forward(self, q, k, v, mask=None):
        q, k, v = self.w_q(q), self.w_k(k), self.w_v(v)
        q, k, v = self.split(q), self.split(k), self.split(v)
        out, _ = self.attention(q, k, v, mask)
        out = self.concat(out)
        out = self.w_concat(out)
        return out

    def split(self, tensor):
        batch_size, length, d_model = tensor.size()
        d_tensor = d_model // self.n_head
        tensor = tensor.view(batch_size, length, self.n_head, d_tensor).transpose(1, 2)
        return tensor

    def concat(self, tensor):
        batch_size, head, length, d_tensor = tensor.size()
        d_model = head * d_tensor
        tensor = tensor.transpose(1, 2).contiguous().view(batch_size, length, d_model)
        return tensor
    
```

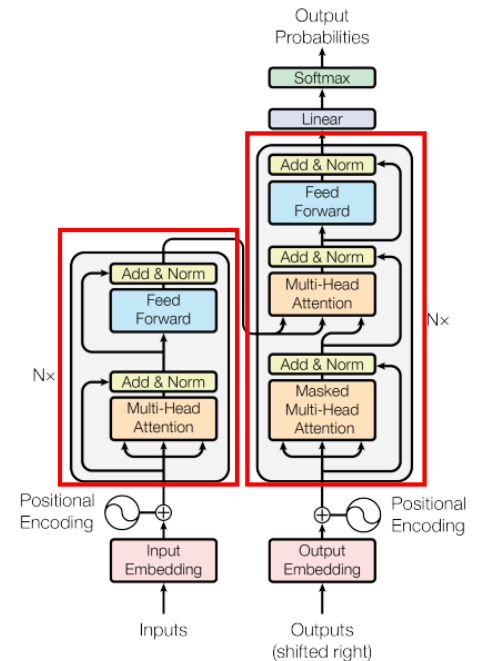


Transformer 구현

- Transformer 신경망 구조의 구현
 - Encoder/Decoder Block: Scaled Dot-Product Attention with mask

```
# %% Scaled Dot-Product Attention
class ScaleDotProductAttention(nn.Module):
    def __init__(self):
        super(ScaleDotProductAttention, self).__init__()
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, q, k, v, mask=None):
        score = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(q.size(-1))
        if mask is not None:
            score = score.masked_fill(mask == 0, -1e9)
        score = self.softmax(score)
        v = torch.matmul(score, v)
        return v, score
```

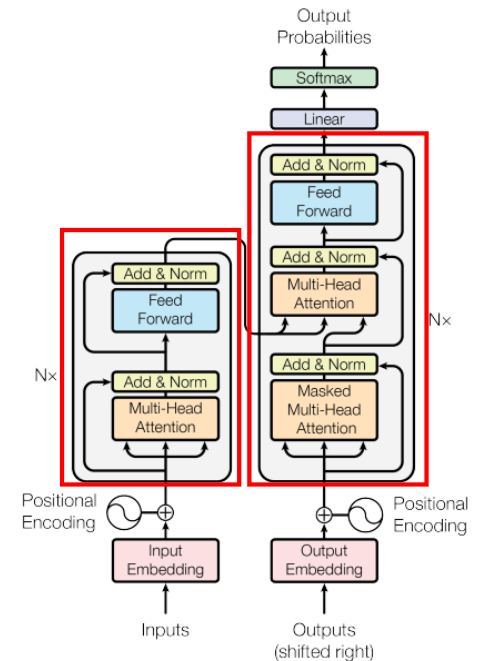


Transformer 구현

- Transformer 신경망 구조의 구현
 - Encoder/Decoder Block: Feedforward

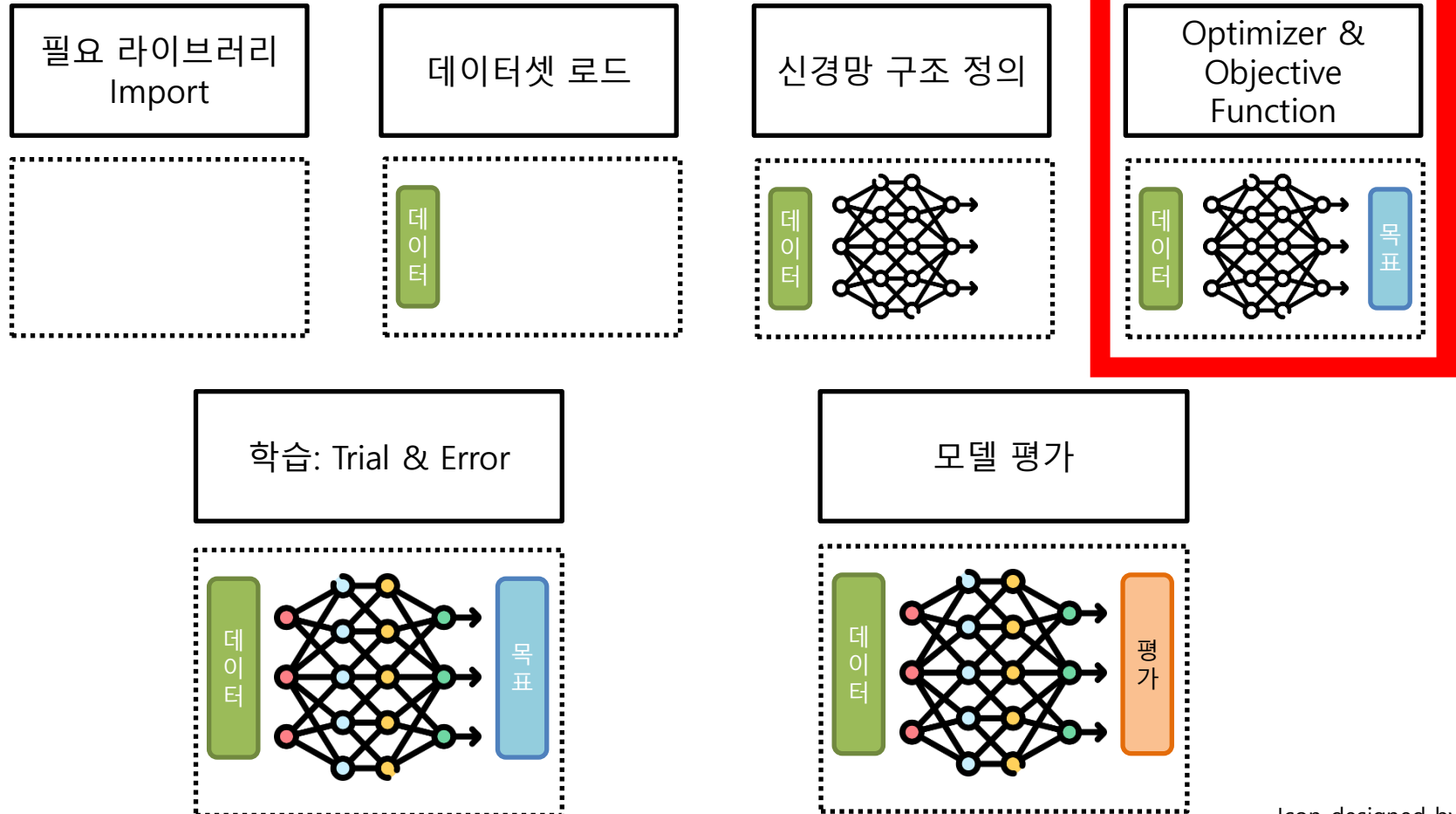
```
# %% Feed Forward Network
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(FeedForward, self).__init__()
        self.linear1 = nn.Linear(d_model, d_ff)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        return self.linear2(self.dropout(F.relu(self.linear1(x))))
```



Transformer 구현

- (Recall) 딥러닝 코드의 대략적 구조



Icon designed by freepik

Transformer 구현

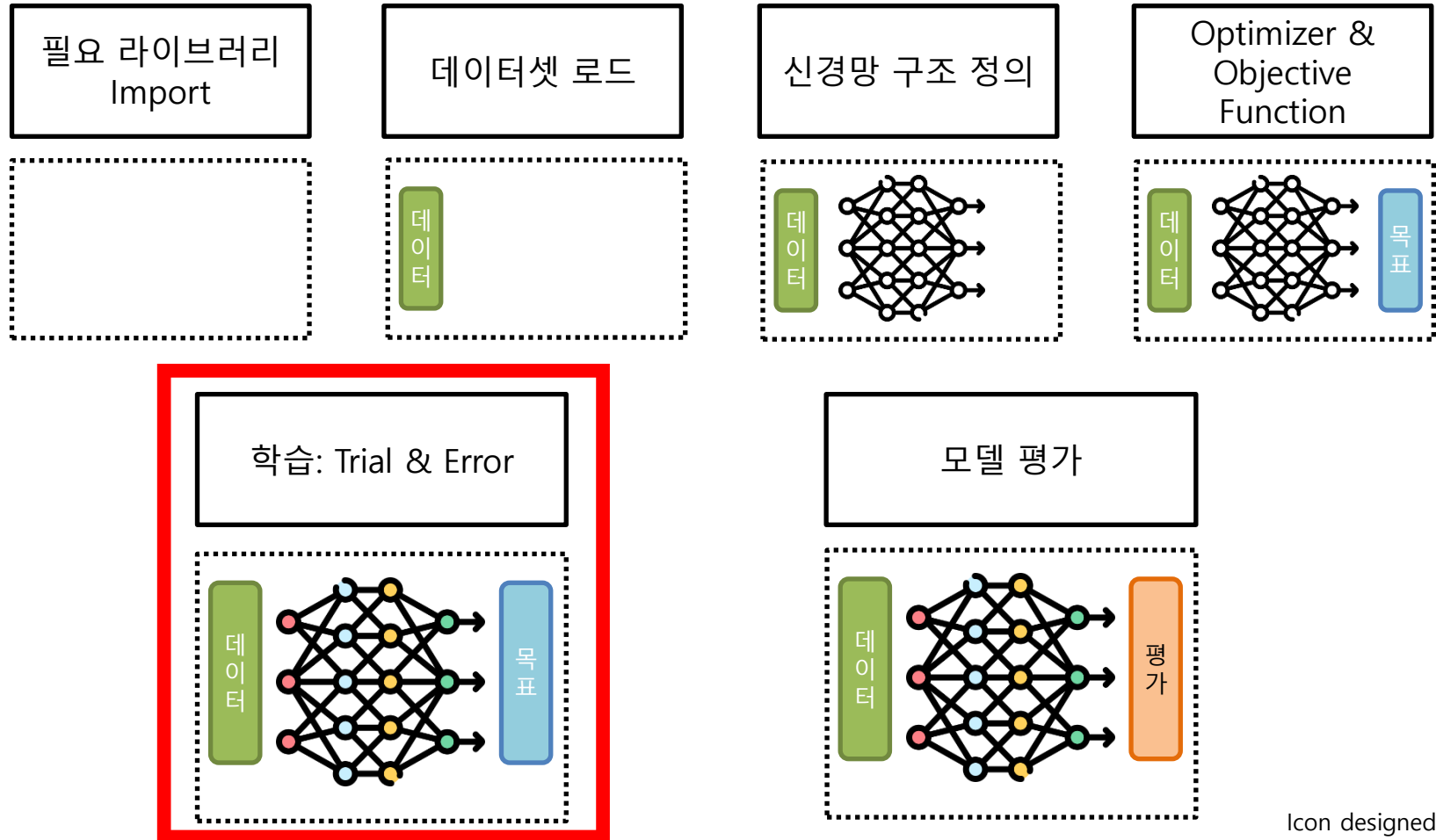
- 모델 호출 및 Optimizer & Loss Function

```
d_model = 512
n_head = 8
d_ff = 2048
dropout = 0.1
num_layers = 6

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = Transformer(vocab_size, vocab_size, d_model, n_head, d_ff, num_layers,
                    dropout).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
```

Transformer 구현

- (Recall) 딥러닝 코드의 대략적 구조



Icon designed by freepik

Transformer 구현

- 학습

```
# %% Train Model Function
def train_model(model, data_loader, optimizer, criterion, device, num_epochs, seq_len):

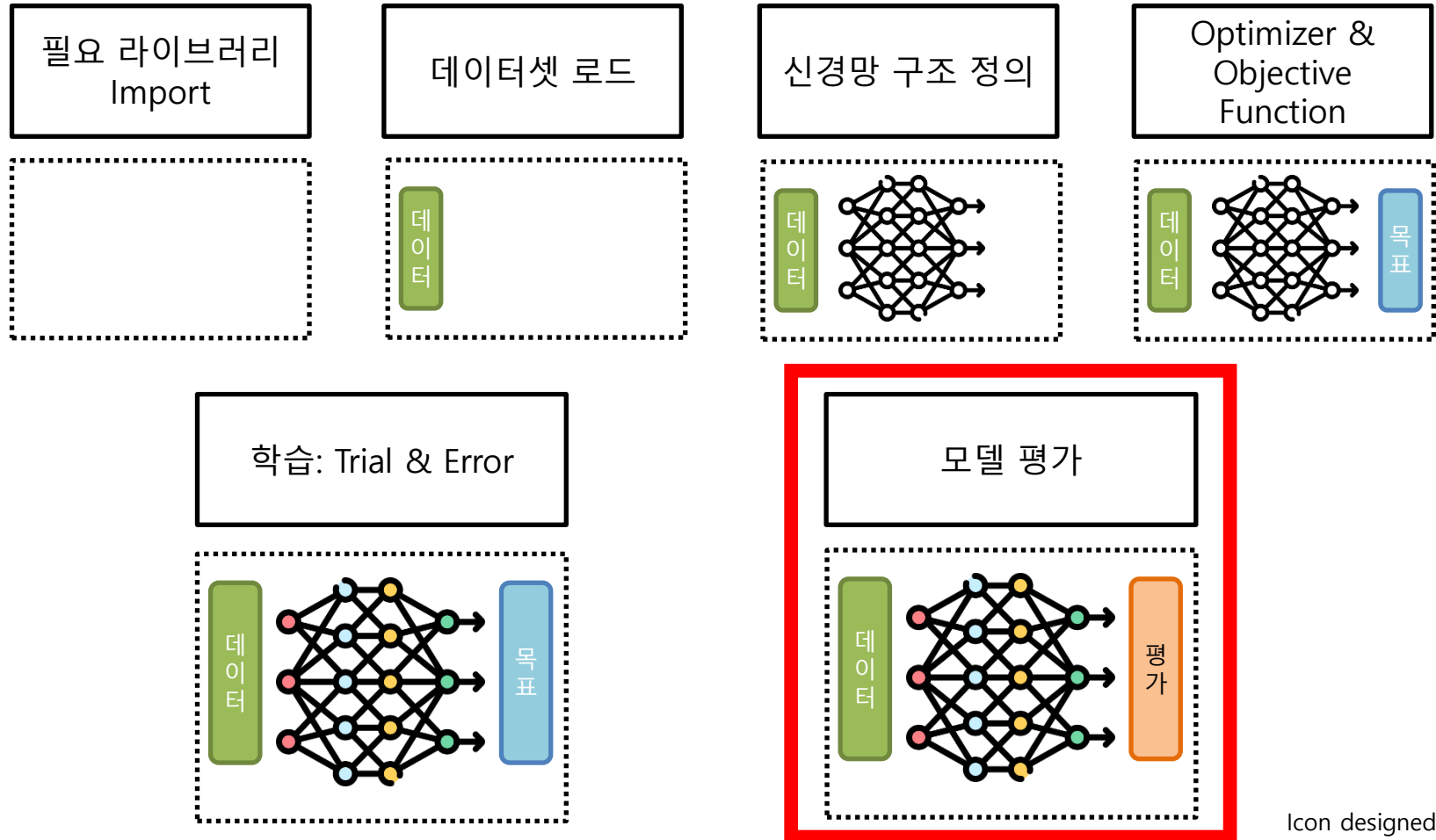
    model.train()
    total_loss = 0

    for src, tgt in data_loader:
        src, tgt = src.to(device), tgt.to(device)
        mask = torch.tril(torch.ones((seq_len, seq_len), device=device))
        optimizer.zero_grad()
        output = model(src, tgt, mask)
        loss = criterion(output.view(-1, output.size(-1)), tgt.view(-1))
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    return total_loss / len(data_loader)
```


Transformer 구현

- (Recall) 딥러닝 코드의 대략적 구조



Icon designed by freepik

Transformer 구현

- 평가

```
# %% Evaluate Model Function
def evaluate_model(model, data_loader, criterion, device, seq_len):

    model.eval()
    total_loss = 0

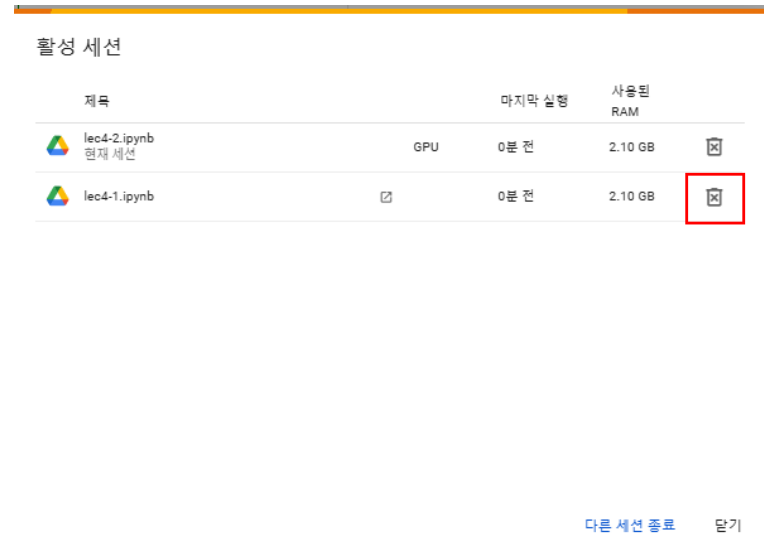
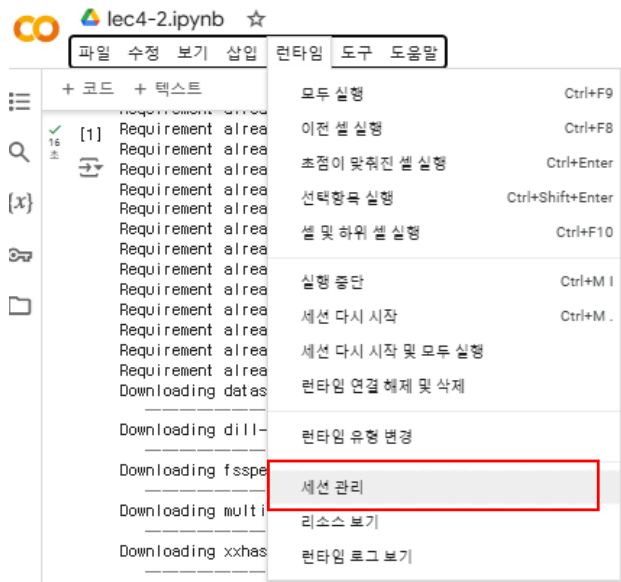
    with torch.no_grad():
        for src, tgt in data_loader:
            src, tgt = src.to(device), tgt.to(device)
            mask = torch.tril(torch.ones((seq_len, seq_len), device=device))
            output = model(src, tgt, mask)
            loss = criterion(output.view(-1, output.size(-1)), tgt.view(-1))
            total_loss += loss.item()

    return total_loss / len(data_loader)
```

Pretrained Transformer 사용해보기

- Colab에서 실습코드 파일 실행

- 구글드라이브에서 lec4-2.ipynb 실행, 런타임 'T4 GPU'로 설정
- 세션 관리에서 lec4-1.ipynb 파일 삭제



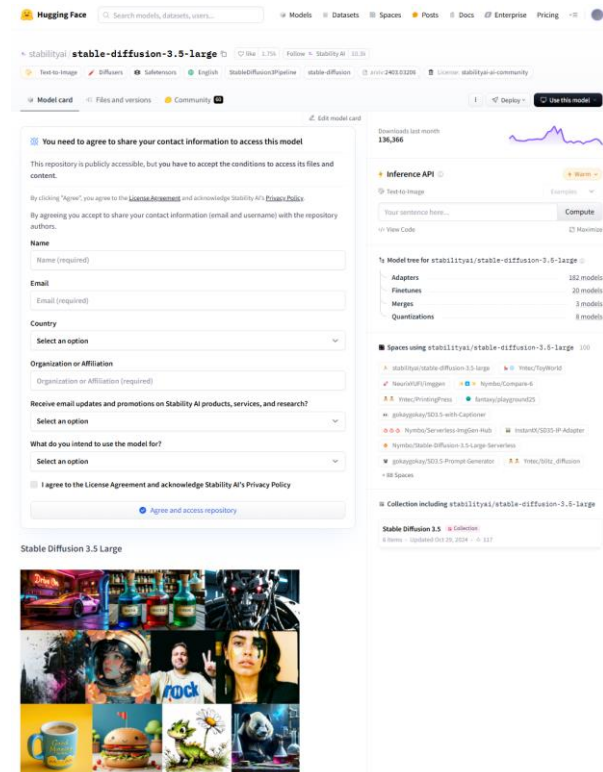
Pretrained Transformer 사용해보기

- **HuggingFace**

- Pre-trained AI model을 간편하게 사용할 수 있는 오픈소스 플랫폼
- Training, Inference의 간소화

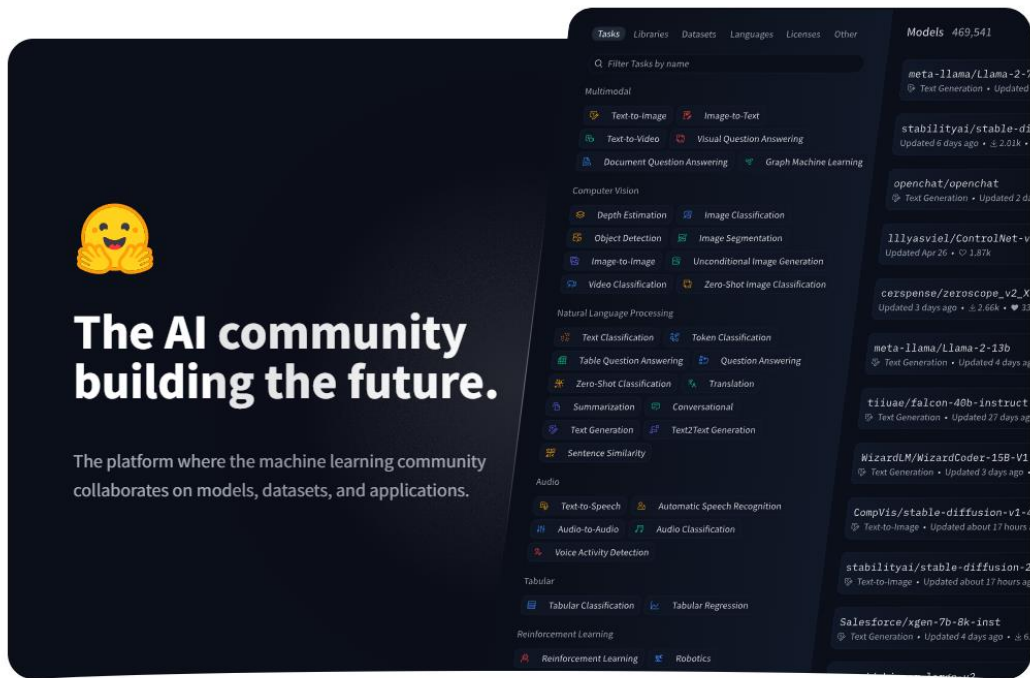


Hugging Face



Pretrained Transformer 사용해보기

- HuggingFace 회원가입
 - <https://huggingface.co/> 에서 계정 생성

The image shows a 'Complete your profile' form on the Hugging Face website. The form has a title 'Complete your profile' and a subtitle 'One last step to join the community'. It contains several input fields: Username, Full name, Avatar (optional) with an 'Upload file' button, Twitter username (optional), GitHub username (optional), LinkedIn profile (optional), Homepage (optional), and AI & ML interests (optional). At the bottom, there's a checkbox for 'I have read and agree with the Terms of Service and the Code of Conduct' and a 'Create Account' button.

Pretrained Transformer 사용해보기

- 라이브러리 설치



```
! pip install transformers datasets sentencepiece accelerate
```

Pretrained Transformer 사용해보기

- Inference 예제

- BERT(Transformer의 Encoder 구조 특화)를 활용한
- 감정 분류 모델(긍정/부정 분류)

```
import os
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification, Trainer, TrainingArguments
from datasets import load_dataset

os.environ["WANDB_DISABLED"] = "true" # for disabling wandb

# load pretrained tokenizer and model
model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=2)

# example input
inputs = tokenizer("I want to submit paper...", return_tensors="pt")

# initial inference for example input: might be random
outputs = model(**inputs.to(model.device))
predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
labels = ["NEGATIVE", "POSITIVE"]
result = labels[torch.argmax(predictions)]
print(result)
```

Pretrained Transformer 사용해보기

- Training(Fine-tuning) 예제

- IMDb(영화 리뷰 데이터셋)에서
- BERT(Transformer의 Encoder 구조 특화) 기반
- 감정 분류 모델(긍정/부정 분류) 훈련

```
# load and preprocess dataset
dataset = load_dataset("imdb") # IMDb 영화 리뷰 데이터셋
train_dataset = dataset["train"].shuffle(seed=42).select(range(1000)) # 일부 데이터로 실험
test_dataset = dataset["test"].shuffle(seed=42).select(range(200))

def preprocess_data(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True, max_length=512)

train_dataset = train_dataset.map(preprocess_data, batched=True)
test_dataset = test_dataset.map(preprocess_data, batched=True)
```

```
# training configuration
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    logging_dir="./logs",
    logging_steps=10,
    save_strategy="epoch",
    report_to="none",
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    tokenizer=tokenizer
)
```

```
# fine-tuning & evaluation
trainer.train()
results = trainer.evaluate()
print(results)
```


Pretrained Transformer 사용해보기

- **Fine-tuning 후 Inference**

- IMDb(영화 리뷰 데이터셋)에서
- BERT(Transformer의 Encoder 구조 특화) 기반
- 감정 분류 모델(긍정/부정 분류) 훈련

```
# inference after fine-tuning
inputs = tokenizer("I want to submit paper...", return_tensors="pt")
outputs = model(**inputs.to(model.device))
predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
labels = ["NEGATIVE", "POSITIVE"]
result = labels[torch.argmax(predictions)]
print(result)
```

Closing

- **Summary**

- Transformer란: Attention Mechanism을 활용한 Sequence Model
 - Attention: 서로 다른 두 Sequence의 token간 연관성을 직접 반영
 - 기본적으로 Encoder – Decoder 구조
 - 개량형으로 Encoder 특화(ex. BERT)와 Decoder 특화(ex. GPT) 있음
- PyTorch를 활용한 Transformer 구현
- HuggingFace를 활용한 Pre-trained model inference & training