Distributed Representation

Embedding

Distributed representation

Distributed Representation - ML(Machine Learning)의 주요 주제

- 개념(concept) 또는 Entity를 심볼(symbol)로 표현하지 않고, 신경망의 여러 노드(node)에 분산시켜 표현
- 개념은 분산된 노드의 'activation' 값에 따라서 결정됨 – 신경망의 장점인 데이터로부터 학습이 가능

Embedding

- 예전에는 이산 심볼(discrete symbol)을 연속 벡터 공간으로 projection하는 것에 주로 사용
- 최근에는 원래 embedding말고도, latent representation, feature vectors로도 사용

Embedding – Distributed Representation의 현대적 기법

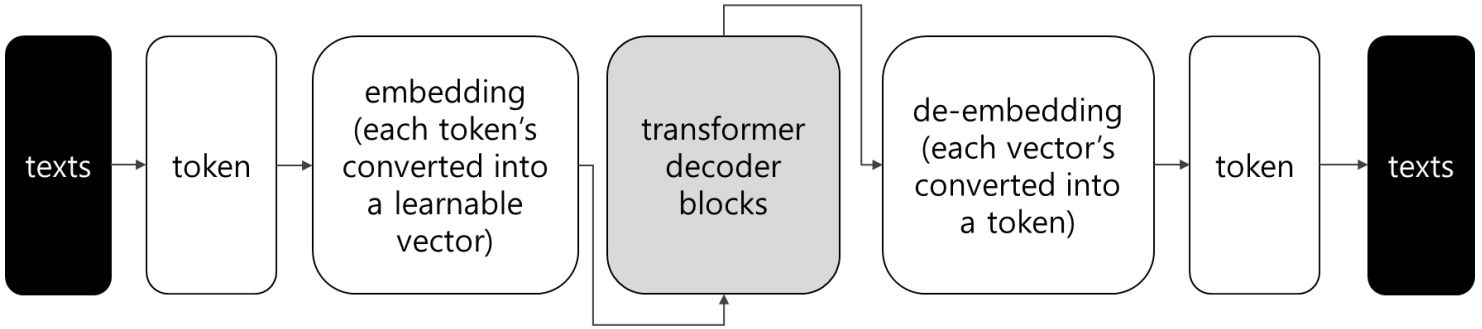
Distributed representation은 넓은 의미의 개념적 용어이고, embedding은 데이터를 고정 차원 벡터 공간에 매핑하는 구체적 기법이지만, 최근에는 embedding이라는 용어를 구분하지 않고 사용

입력 데이터를 Neural Network안에서 어떻게 표현해야 할까?

입력 데이터를 Neural Network안에서 어떻게 표현해야 할까?

- Text의 경우 Tokenizer 출력인 ID sequence로 구성
- RGB Image의 경우 HW 크기에 1-pixel 당 0~255의 r-g-b값 총 3byte로 구성
- Robot의 state vector는 real value로 구성된 다차원 벡터

Embeddings in NLP

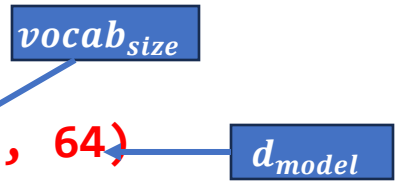


텍스트 → Tokenizer → ID sequence 출력 → Embedding Table → ID 마다 고차원 vector 할당

- Embedding table는?
- $vocab_{size}$: 토크나이저(tokenizer)에서 정한 총 토큰 수
- d_{model} : 트랜스포머 네트워크(Transformer Network) 모델의 입력 벡터 크기

Embedding Table 구현

- nn.Parameter




```
import torch.nn as nn
vectors = torch.randn(100, 64)
emb_table = nn.Parameter(vectors)
ids = torch.tensor([[0, 2, 5], [0, 99, 24]])
emb = emb_table[ids]
print(emb.shape)
```

```
torch.Size([2, 3, 64])
```

- nn.Embedding

```
import torch.nn as nn
emb_table = nn.Embedding(100, 64)
ids = torch.tensor([[0, 2, 5], [0, 99, 24]])
emb = emb_table(ids)
print(emb.shape)
```

```
torch.Size([2, 3, 64])
```

Positional Encoding(Embedding)

입력 데이터(Text/Image)를 embedding vectors의 sequence로 변환하고 곧 바로 Transformer Network에 입력하게 되면, Transformer Network 특성상 원래 입력 데이터에서 가지고 있었던 토크의 위치/순서 정보를 상실하게 된다.

→이를 방지하기 위하여 각 토큰의 위치 정보를 encoding하여 embedding vector에 영향을 주는 기법을 position encoding이라고 함.

Positional Embedding 종류

Absolute positional embedding (additive 방식)

- 입력 토큰 벡터 x에 위치 벡터 p를 더한 후에 Transformer Network로 입력됨
- 종류: Sine-Cosine 인코딩, Learnable Parameters

Rotary 계열 (multiplicative, rotation 방식) – Relative position

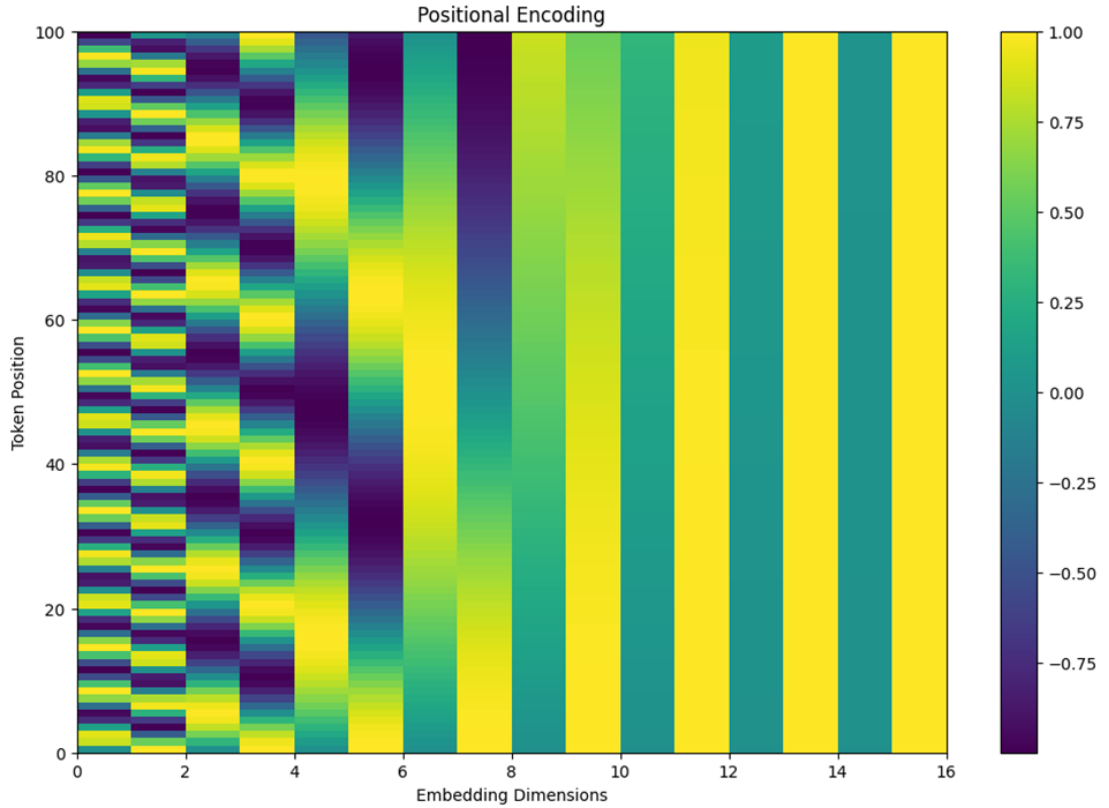
- 토큰 embedding 벡터 x는 그대로 입력, Transformer Network 안에서 Q, K 값을 위치에 따라 rotation하고 attention 함
- 종류: RoPE, XPos, YaRN

Original (Sinusoidal) Positional Encoding

- Given a position p in the sequence and a dimension i of the embedding (where $2 \leq i \leq d_{model}$), the dimensionality of the model), the positional embeddings are defined as:

$$PE(p, 2i) = \sin\left(\frac{p}{10000^{2i/d_{model}}}\right)$$

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



Sinusoidal Positional Encoding 구현 예

▪ 클래스 definition 일부

```
# position encodin을 위한 (max_len, d_model) matri를 생성하고 '0'으로 초기화
pe = torch.zeros(max_len, d_model)

p = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1) # (max_len, 1)

div_term = torch.exp(torch.arange(0, d_model, 2).float()*(-math.log(10000.0)/d_model))
pe[:, 0::2] = torch.sin(p * div_term) # p * div_term → (max_len, 1) * (d_model//2)
pe[:, 1::2] = torch.cos(p * div_term) # broadcasting → (max_len, d_model//2)
pe = pe.unsqueeze(0)  # Add batch dimension

self.register_buffer('pe', pe) # PyTorch에서 position encoding을 gpu로 move할 수 있게 등록

def forward(self, x):
 """     x: Tensor of shape (batch_size, seq_len, d_model) """

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

RoPE (Rotary Position Embedding)

Absolute positional embedding 계열

- Transformer Network 입력 전에 positional embedding을 'adding'.
- Attention operation: $Attn(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$

RoPE 계열

- 토큰 embedding을 Position Encoding을 더하지 않고 직접 Transformer Network에 입력
- Attention할 때, Q, V의 토큰 벡터의 차원을 홀수-짝수로 나누어 2D 좌표처럼 취급, 로테이션 하여 계산
- $Q_i^{RoPE} = R_{\theta_i}Q_i,\ K_j^{RoPE} = R_{\theta_j}K_j$
- $s_{i,j} = \frac{Q_i^{RoPE}\cdot K_j^{RoPE}}{\sqrt{d}} = \frac{\left(R_{\theta_i}Q_i\right)\cdot\left(R_{\theta_j}K_j\right)}{\sqrt{d}} = \frac{\left(R_{\theta_i}Q_i\right)^T\left(R_{\theta_j}K_j\right)}{\sqrt{d}} = \frac{Q_i^T R_{\theta_j-\theta_i}K_j}{\sqrt{d}}$
  - 여기서, $R_\theta^T R_\phi = R_{\phi-\theta}$
- 즉, 위치 i와 j의 차이 $\theta_j - \theta_i$ 가 s_{ij}에 반영됨 → 상대적 위치 정보가 attention score에 직접 반영되는 구조

RoPE 구현

1. 사전 준비: cos/sin 테이블 구성 → 최대 길이 L_{max}, 모델 차원 d_{model} (짝수)라고 하면,

$$inv_{freq_j} = \frac{1}{10000^{2j/d_{model}}}, \qquad j = 0, 1, \dots, \frac{d_{model}}{2} - 1$$

$$\theta_{pos,j} = pos \cdot inv_{freq_j}, \qquad pos = 0, 1, \dots, L_{max} - 1$$

$\theta_{pos,j}$를 가지고 cos, sin 테이블 구성 → shape는 $\left[L_{max}, \frac{d_{model}}{2}\right]$

2. 시퀀스 T에 대하여, 토큰 위치 $pos = 0 \dots T - 1$ 에 대해, Q/K의 짝수-홀수 차원 (x_{2j}, x_{2j+1})을 묶어서 다음과 같이 회전한 후에 (x'_{2j}, x'_{2j+1}) 를 interleave해서 원래 크기의 토큰으로 복원함

$$\begin{bmatrix} x'_{2j}(pos) \\ x'_{2j+1}(pos) \end{bmatrix} = \begin{bmatrix} cos\theta_{pos,j} & -sin\theta_{pos,j} \\ sin\theta_{pos,j} & cos\theta_{pos,j} \end{bmatrix} \begin{bmatrix} x_{2j}(pos) \\ x_{2j+1}(pos) \end{bmatrix}$$

3. 회전된 Q, K를 이용하여 attention score 계산 – V는 회전하지 않음

% 위 회전 계산은 복소수를 사용하여 동일하게 계산할 수 있음

Embeddings in Vision

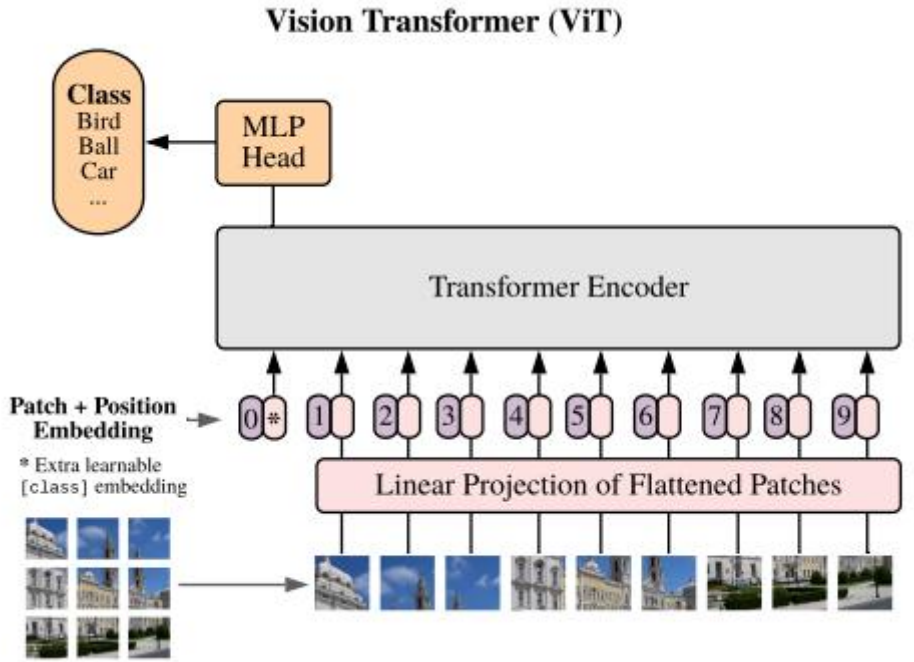
- 이미지를 고정 크기의 패치(patch)로 나눈 후 이를 일종의 토큰으로 간주하여 임베딩
- 이미지 → 패치 분할 → 벡터화 → 선형 변환

- 구현은

① conv2D (stride를 patch size로 해서) 한 후 flatten

② unfold 함수를 사용한 후 Linear 적용

- 일반적으로 conv2D 방식 선호 – 빠르고 메모리 효율



Conv2D 방식 예

```
# 입력 이미지: x = [1, 3, 224, 224]
# 커널 사이즈(패치 사이즈): kernel_size = patch_size = 16
# 스트라이드(패치 사이즈): stride = pathch_size = 16
# 입력채널크기(r-g-b bytes): 3
# 출력채널크기(모델 입력 사이즈): out_chans =  d_model  = 768

proj = nn.Conv2d(in_chans, out_chans, kernel_size=patch_size, stride=patch_size)
x = proj(x)              # x.shape? [1, 768, 14, 14]
x = x.flatten(2)         # x.shape? [1, 768, 196], flatten(start=0, end=-1)
x.transpose(1, 2) → [1, 196, 768]   # 768-dimension의 토큰 196 시퀀스
```