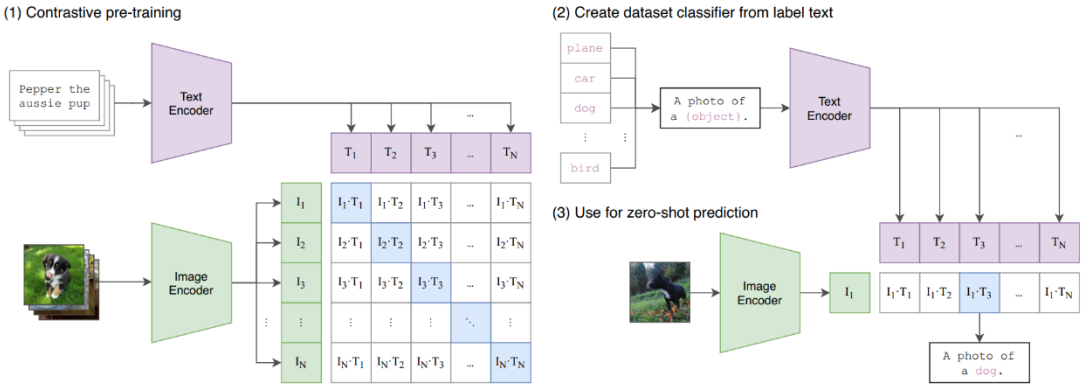
CLIP 모델 구현

김강수

1. CLIP Architecture



CLIP 모델 구조

CLIP 논문(*A. Radford et al. 2021*)에서 제시된 모델의 구조는 위 그림과 같다. 논문에서 제시된 Text Encoder는 GPT-2(*A. Radford et al. 2019*)를 Image Encoder로는 ViT(*A. Dosovitskiy. et al. 2020*)와 ResNet-50(*He. et al. 2016*)을 사용했다. 논문에 기재된 내용과 최대한 동일한 구현을 수행하기 위해 ViT-B/16와 GPT-2를 사용하였다. 구현된 코드는 아래에서 확인 가능하다.
https://github.com/GangsuKim/CLIP-pytorch

2. Model Hyperparameters

CLIP의 모델과 학습에 사용된 *Hyperparameter*들은 arXiv 버전(*A. Dosovitskiy. et al. 2020*)에서 확인 할 수 있었다. 이를 토대로 실제 구현에 사용된 *Hyperparameter*는 아래와 같다.

Hyperparameter	Value
Batch size	32
Vocabulary size	-
Adam β_1	0.9
Adam β_2	0.98
Adam ϵ	10^{-6}

Model	Learning Rate	Embedding dimension	Input resolution	ViT layers	ViT width	ViT heads	Text Transformer layers	Text Transformer width	Text Transformer Heads
ViT-B/16	5×10^{-4}	512	224	12	768	12	12	512	8

3. Model Architecture

3.1. Text Encoder

Text Encoder는 transformers의 GPT-2[1]라이브러리를 사용하였다. 모든 인코더들은 하나의 CLIP Model class에 포함되어 있다.

```
# Text Encoder
self.bpe = GPT2Tokenizer.from_pretrained("openai-community/gpt2")
self.text_encoder_config = GPT2Config(vocab_size=n_vocab, n_positions=76, n_embd=512, n_layer=12, n_head=8
                                       , output_hidden_states=True)
self.text_encoder = GPT2LMHeadModel(self.text_encoder_config)
self.W_t = nn.Linear(self.text_encoder_config.n_embd, n_embedding_space)
```

Text encoder 선언

HuggingFace의 transformers 라이브러리는 GPT2LMHeadModel (GPT-2)모델과 함께 GPT2Tokenizer (tokenizer)와 GPT2Config (config class)를 제공한다. 해당 class들을 사용하여 Text Encoder를 구성하였다. GPT2Tokenizer는 OpenAI에서 제공하는 사전 훈련된 vocabulary를 사용하였다.

3.2. Image Encoder

Image Encoder는 GitHub의 open source code[2]를 clone하여 사용하였다. 해당 실험에 사용된 모델은 ViT-B/16이다.

```
# Image Encoder
self.image_encoder = ViT(image_size=224, patch_size=16, dim=768, depth=12, heads=8, mlp_dim=3072)
self.W_i = nn.Linear(self.image_encoder.dim, n_embedding_space)
```

Image encoder 선언

CLIP 논문에서 사용된 ViT 모델은 두가지 부분이 수정된 ViT 모델을 사용하였다. 아래 그림에서 노란색 부분은 본문의 [1]Patch embedding 이후 Layer Normalization 수행 부분이며 빨간색 부분은 [2]Classification head (MLP) 제거가 수행 된 부분이다.

```
def forward(self, img):
    x = self.to_patch_embedding(img)
    x = self.patch_embedding_norm(x)  # Additional Layer Normalization From CLIP Paper
    b, n, _ = x.shape

    cls_tokens = repeat(self.cls_token, pattern: '1 1 d -> b 1 d', b = b)
    x = torch.cat( tensors: (cls_tokens, x), dim=1)
    x += self.pos_embedding_norm(self.pos_embedding[:, :(n + 1)])  # Additional Layer Normalization From CLIP Paper
    x = self.dropout(x)

    x = self.transformer(x)

    x = x.mean(dim = 1) if self.pool == 'mean' else x[:, 0]

    x = self.to_latent(x)
    # x = self.mlp_head(x) # Return only latent From CLIP Paper
    return x
```

ViT Model의 코드에서 수정된 부분

[1]https://huggingface.co/openai-community/gpt2
[2]https://github.com/lucidrains/vit-pytorch

3.3. Temperature Parameter

CLIP 논문에서는 학습된 temperature parameter τ를 사용했지만 본 구현에서는 편의를 위해 CLIP Model의 class에 nn.Parameter를 사용하여 파라미터로 선언하였다.

```
# Temperature Parameter
self.temperature = temperature
self.t = nn.Parameter(torch.FloatTensor([self.temperature]))
```

Temperature Parameter 선언

3.4. Overall

CLIP 모델의 전반적인 선언부는 아래와 같다.

```
class CLIP(nn.Module):
    GangsuKim *
    def __init__(self, tokenizer: GPT2Tokenizer, n_embedding_space: int = 512, temperature: float = 0.07, n_vocab: int = 50257):
        super(CLIP, self).__init__()

        # Text Encoder
        self.bpe = tokenizer
        self.text_encoder_config = GPT2Config(vocab_size=n_vocab, n_positions=76, n_embd=512, n_layer=12, n_head=8, output_hidden_states=True)
        self.text_encoder = GPT2LMHeadModel(self.text_encoder_config)
        self.W_t = nn.Linear(self.text_encoder_config.n_embd, n_embedding_space)

        # Image Encoder
        self.image_encoder = ViT(image_size=224, patch_size=16, dim=768, depth=12, heads=8, mlp_dim=3072)
        self.W_i = nn.Linear(self.image_encoder.dim, n_embedding_space)

        # Temperature Parameter
        self.temperature = temperature
        self.t = nn.Parameter(torch.FloatTensor([self.temperature]))
```

CLIP 모델의 선언부

4. Forward

4.1. Text Tokenizing

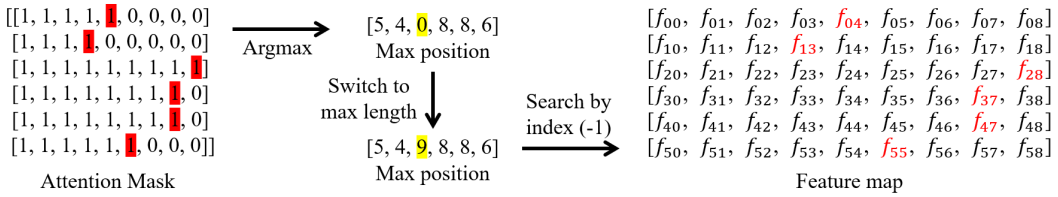
Text encoder가 학습을 수행하기 위해서는 문자열의 형태가 아닌 model이 이해 할 수 있는 형태인 tensor 형태로 변환을 해 주어야 한다. 이를 위해 논문에서 언급한 BPE(Byte Pair Encoding)를 사용하는 GPT2Tokenizer를 사용해야 한다. 이를 batch 단위로 수행하기 위해서는 tokenizing을 batch 상태가 입력으로 들어오는 forward 단계에서 수행을 해야 한다. 따라서 tokenizer를 forward 부분에서 사용하게 되었다.

```
def forward(self, image, texts):
    # extract feature representations of each modality

    # Text Tokenizing
    text_inputs = self.bpe(texts, padding=True, return_tensors="pt").to('cuda' if torch.cuda.is_available() else 'cpu')
```

Text tokenizing

Text encoder에서 feature는 [EOS] token의 마지막 transformer layer의 가장 마지막 attention의 feature map을 사용한다고 논문에서 언급되어 있다. GPT2Tokenizer는 [EOS] token의 위치를 따로 알려주는 메소드가 없기에 이를 수행하기 위해서는 batch의 각 데이터마다 [EOS] 토큰의 위치를 추적해야 한다. 이를 추적하기 위해 tokenizer가 반환하는 attention mask를 사용하여 [EOS] 토큰의 위치를 추적했다. 아래 그림은 [EOS] 토큰을 추적하는 방법론의 도식화 이다.



[EOS] token 추적의 도식화

위 방법에 대한 구현은 아래와 같다. 1 부터 시작하는 index를 list comprehension으로 바로 적용하기 위해 index 값에서 1을 빼 주었다.

```
# Get [EOS] token's index from text token embeddings
eos_places = torch.argmin(text_inputs.attention_mask, dim=1)
eos_places[eos_places == 0] = len(text_inputs.attention_mask[-1])
eos_places -= 1
```

[EOS] token 추적 코드

4.2. Feature extraction and Projection

이를 토대로 text와 image의 특징을 추출한다. 추출을 위한 코드는 아래와 같다. h는 text encoder의 마지막 attention layer의 hidden state이다.

```
h = self.text_encoder(**text_inputs).hidden_states[-1]  # T_f = text_encoder(T) - 1
T_f = h[torch.arange(h.size(0)), eos_places, :]  # T_f = text_encoder(T) - 2
I_f = self.image_encoder(image)  # I_f = image_encoder(I)
```

feature extraction

이후 learned projection을 통해 embedding 차원으로 projection을 수행하지만 이 또한 편의를 위해 학습 가능한 선형 레이어인 W_i와 W_t를 통해 임베딩 차원으로 투영을 수행한다.

```
# joint multimodal embedding
T_e = F.normalize(self.W_t(T_f), dim=1)  # T_e = l2_normalizer(np.dot(T_f, W_t), axi=1)
I_e = F.normalize(self.W_i(I_f), dim=1)  # I_e = l2_normalizer(np.dot(I_f, W_i), axi=1)
```

Linear projection & Normalize

4.3. Calculate pairwise cosine similarities

최종적으로 text embedding과 image embedding의 cosine 유사도를 계산 하고 $\exp(\tau)$로 정규화를 수행한 logit 값을 반환한다.

```
# scaled pairwise cosine similarities [n, n]
logit = torch.matmul(I_e, T_e.T) * torch.exp(self.t)  # logits = np.dot(I_e, T_e.T) * np.exp(t)
return logit
```

Calculate cosine similarities

4.4. Loss

손실 함수 계산은 학습 부분에서 수행한다.

```
logits = model(images, label)  # logits = np.dot(I_e, T_e.T) * np.exp(t)

# symmetric loss function
matrix_labels = torch.eye(images.size(0)).to(device)  # labels = np.arange(e)
loss_i = criterion(logits, matrix_labels)  # loss_i = cross_entropy_loss(logits, labels, axis=0)
loss_t = criterion(logits.T, matrix_labels)  # loss_t = cross_entropy_loss(logits, labels, axis=1)
loss = (loss_i + loss_t) / 2  # loss = (loss_i + loss_t) / 2
```

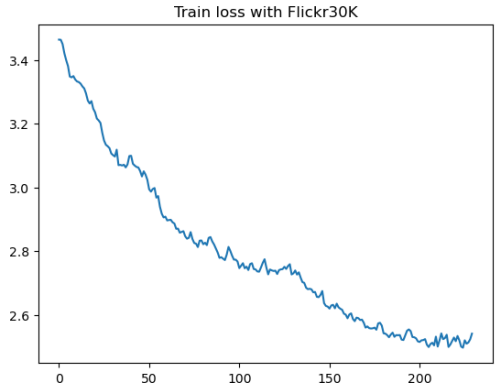
Calculate loss function

5. Self-supervised pre-training

모델의 사전학습은 이미지 captioning 분야에서 주로 사용되는 데이터 셋인 Flickr30K[3] 데이터셋을 사용하여 학습을 수행하였다. Flickr30K 데이터셋은 31,783장의 데이터로 구성되어 있으며 각 이미지당 5개의 caption이 존재한다. 하지만 실제 학습에서는 중복을 방지하기 위해 이미지당 1개의 caption만 사용하여 학습을 수행하였다. 학습은 1개의 RTX 3090 환경에서 32의 batch size로 16시간

[3]https://shannon.cs.illinois.edu/DenotationGraph/

동안 수행했다. 학습 시 loss의 양상은 아래와 같은 모습이다. Loss가 감소하기는 하지만 데이터 양의 한계 및 학습 환경의 한계로 인해 매우 낮은 수치까지 내려가는 양상을 보이지는 않았다.



Training loss graph

5. Zero-shot prediction

Zero-shot prediction은 본 논문에서도 사용된 Food101[4] 데이터셋을 사용하였다. Food101 데이터셋은 총 101개의 음식 class가 포함되어 있는 데이터 셋으로 각 class마다 1,000장의 이미지 총 101,000장의 이미지가 존재한다. 이 데이터를 통해 zero-shot prediction을 수행한 결과의 분류 성능은 약 1.1%로 그리 높지 않았다. 모델의 학습 정도와 pre-train에 사용된 이미지의 전반적인 내용이 사람의 상호작용 위주로 되어 있었던 반면에, zero-shot prediction에 사용된 데이터셋은 음식 사진만 존재하는 데이터 셋으로 완전히 반대되는 성향의 데이터로 학습을 한 뒤 zero-shot prediction을 수행하였기에 좋지 않은 결과가 나온 것 이라고 판단했다. 이를 보완하기 위해 기존 모델의 200 epoch의 checkpoint를 토대로 MSCoCo2014[5] 데이터 셋을 사용해서 동일한 환경에서 7 epoch 동안 추가 학습을 진행하였다. 이를 토대로 동일한 Food101 데이터 셋에 대해서 zero-shot prediction을 수행한 인식 결과는 1.2% 정도로 크게 차이가 나지 않는 모습을 보였다. Zero-shot prediction을 수행하기에는 모델이 충분한 학습을 하지 않은 것으로 판단하여 CIFAR-10[6] 데이터 셋의 test데이터를 사용하여 zero-shot prediction을 수행해 보았다. 이 결과 인식 성능은 약 17.1%로 기존의 Food101 데이터 셋에 비해 높은 인식률을 보이는 것을 확인 할 수 있었다. 이를 통해 pre-train 모델이 학습하는 데이터가 편향된 부류의 데이터 셋 이라면 모델의 성능을 저하 시킬 수 있다 라는 결론을 얻었다.

	CLIP-Flickr	CLIP-Flickr-MSCoCo
Food101	1.1%	1.1%
CIFAR-10	12.2%	16.8%

Image classification result (Acc)

6. Code compare

OpenAI에서 공식적으로 제공하는 코드를 통해서 비교를 수행 해 본 결과 코드의 전반적인 양상은 비슷하나 @ 연산자를 매우 자주 사용하는 모습을 확인 할 수 있었다. CLIP의 코드를 통해 이해 해 본 결과는 행렬 곱 연산일 것으로 추정된다. 또한 많은 module에서 layer 를 선언하여 사용

[4]https://data.vision.ee.ethz.ch/cvl/datasets_extra/food-101/
[5]https://cocodataset.org/
[6]https://www.cs.toronto.edu/~kriz/cifar.html

하지 않고 연산자를 통해 수행하려는 모습들이 많이 보였다.