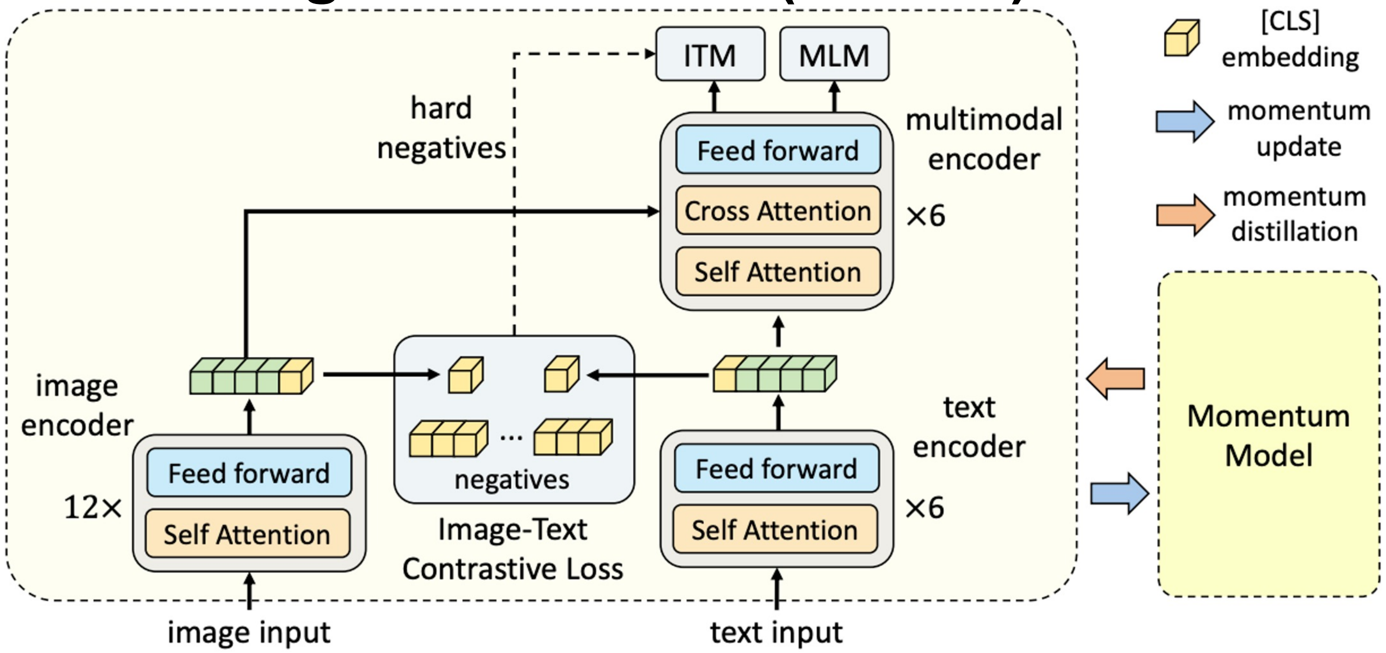
Align before Fuse: Vision and Language Representation Learning with Momentum Distillation -code-

AAI Lab. 서원희, 최창수

Index

1. Introduction

ALign BEfore Fuse(ALBEF) 구조



- Image encoder, Text encoder, Multimodal encoder로 구성됨
  - 입력 이미지 I는 $\{\vec{v}_{cls}, \vec{v}_1, \ldots, \vec{v}_N\}$, 입력 텍스트 T는 $\{\vec{w}_{cls}, \vec{w}_1, \ldots, \vec{w}_N\}$ 임베딩으로 인코딩 됨
  - Image encoder: visual transformer ViT-B/16의 12-layer 사용
  - Text encoder: BERTbase 모델의 first 6-layer 사용
  - Multimodal encoder: BERTbase 모델의 last 6-layer사용
- Multimodal encoder의 각 layer에서 cross attention을 통해 Image features와 Text feature를 Fusion

Pre-training Objectives

Image-Text Contrastive Learning(ITC)

- 멀티모달 인코더에서 image, text feature를 fusion하기 전 unimodal encoder를 학습 하는 것을 목적
- **같은 image-text pair(positive)로 부터 얻은 feature는 similarity가 높아지도록, 다른 image-text pair(negative)로 부터 얻은 features는 similarity가 낮아지도록 학습**
- ex) 만약 batch가 64라면, 하나의 image embedding에 대해 같은 pair인 text embedding은 similarity score가 높아지게, 나머지 negative text embeddings는 socre가 낮아지도록 학습 하는 것

Masked Language Modeling(MLM)

- **이미지와 mask를 씌우지 않은 text를 활용해 mask 씌운 단어를 맞추는 적을 목적**
- BERT와 마찬가지로 input tokens의 15%를 랜덤하게 마스크함

Image-Text Matching(ITM)

- **image-text pair가 positive(matched)인지 negative(not matched)인지 예측하는 objective**
- 멀티모달 인코더의 [CLS] embedding을 fully-connected layer를 거친후 softmax로 matching 여부를 예측

$$\mathcal{L} = \mathcal{L}_{itc} + \mathcal{L}_{mlm} + \mathcal{L}_{itm}$$

Momentum Distillation

- Pre-training에 사용되는 image-text 쌍은 대부분 웹에서 수집되며 Noise가 많은 경향이 있음.
  - Positive pairs는 텍스트가 이미지와 무관한 단어를 포함하거나 이미지가 텍스트에서 서술되지 않은 객체를 포함하는 등 보통 연관성이 약함.
  - ITC 학습의 경우 이미지에 대한 negative texts가 이미지의 내용과 일치할 수 있음.
  - MLM의 경우, 영상을 동일하게 잘 설명하는(또는 더 잘 설명하는) 원본의 주석과 다른 단어가 있을 수 있음. 그러나 ITC와 MLM에 대한 one-hot label은 정확성과 관계없이 모든 negative prediction에 벌칙을 부과함.

이를 위해 본 논문에서는 Momentum 모델에 의해 **생성된 pseudo-target로부터 학습**하는것을 제안

Momentum 모델: 유니모달, 멀티모달 인코더의 Exponential Moving Average(EMA)로 구성된 continuously-evolving teacher

타겟이 학습을 진행할 때 마다 업데이트 된다는것을 의미

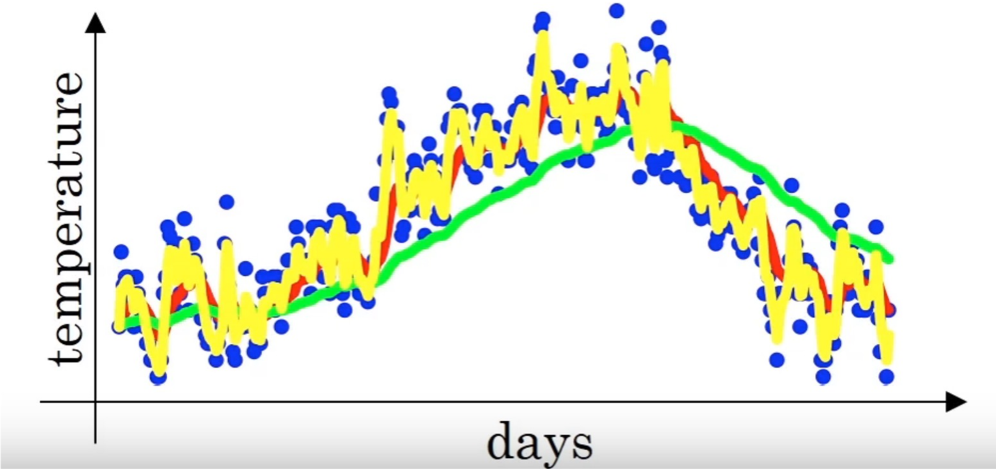
$$\mathcal{L}_{\mathrm{itc}}^{\mathrm{mod}} = (1-\alpha)\mathcal{L}_{\mathrm{itc}} + \frac{\alpha}{2}\mathbb{E}_{(I,T)\sim D}\left[\mathrm{KL}(\boldsymbol{q}^{\mathrm{i2t}}(I) \parallel \boldsymbol{p}^{\mathrm{i2t}}(I)) + \mathrm{KL}(\boldsymbol{q}^{\mathrm{t2i}}(T) \parallel \boldsymbol{p}^{\mathrm{t2i}}(T))\right]$$

$$\mathcal{L}_{\mathrm{mlm}}^{\mathrm{mod}} = (1-\alpha)\mathcal{L}_{\mathrm{mlm}} + \alpha\mathbb{E}_{(I,\hat{T})\sim D}\mathrm{KL}(\boldsymbol{q}^{\mathrm{msk}}(I,\hat{T}) \parallel \boldsymbol{p}^{\mathrm{msk}}(I,\hat{T}))$$

What is EMA?

$$V_t = \beta \times V_{t-1} + (1-\beta) \times \Theta_t$$

- 이때 β 는 0~1 사이의 값을 갖는 하이퍼파라미터, 세타(Θ)는 새로 들어온 데이터, V는 현재의 경향을 나타내는 값
- β의 크기가 커질 수록 v_t에서 고려하는 데이터의 크기 (window size)가 커지므로, 보다 곡선이 smooth해짐.
  따라서 급격한 변화에 둔감해짐.
- β가 커진다는 의미는 결국, previous value에 더 큰 가중치를 주는 것이기 때문에, 현재의 value에 영향력을 감소시킴

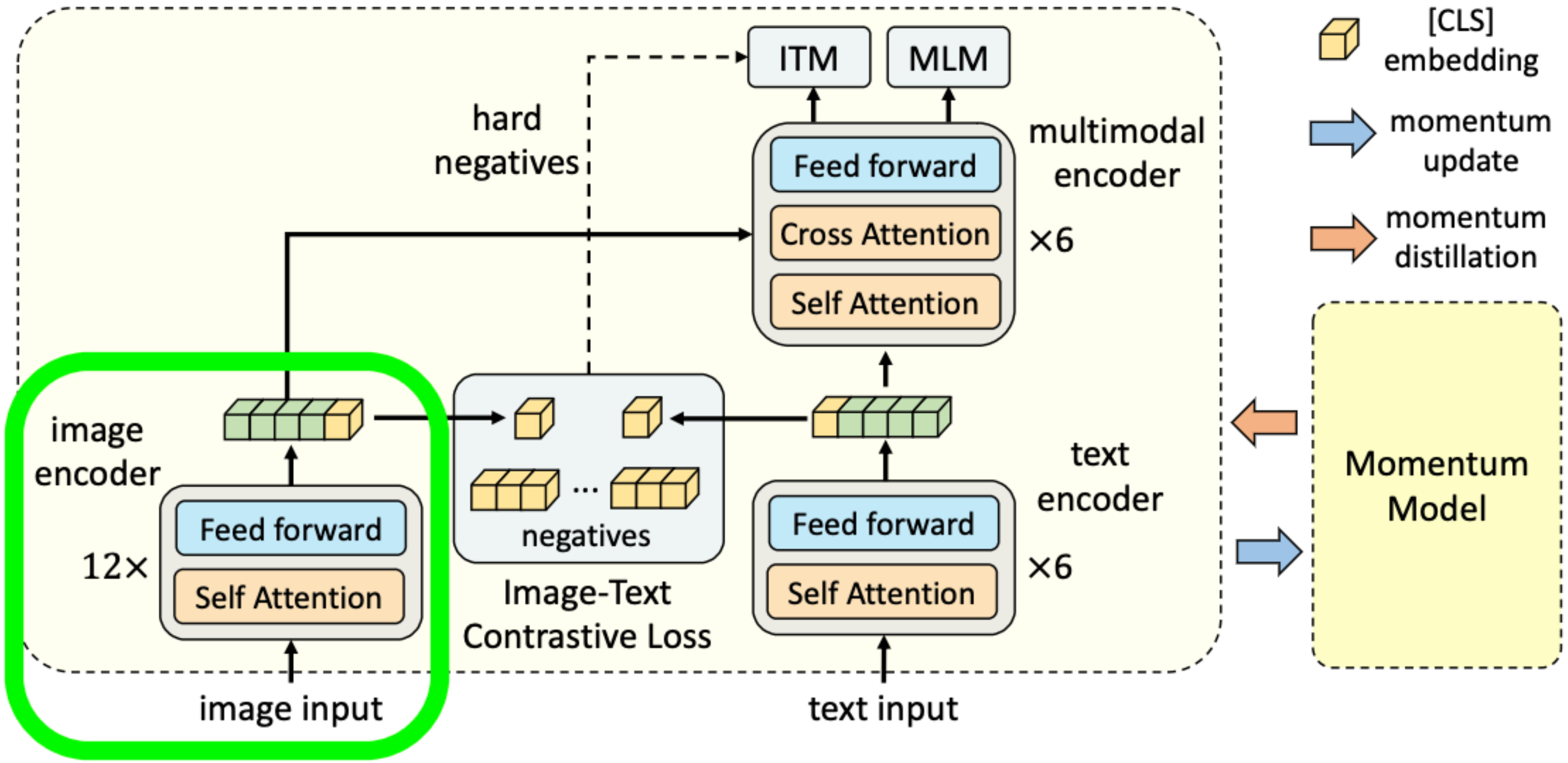


초록색 : β값이 높음, smooth
노란색 : β이 작음, 새로운 데이터에 adaptive해짐

2. Code

Code

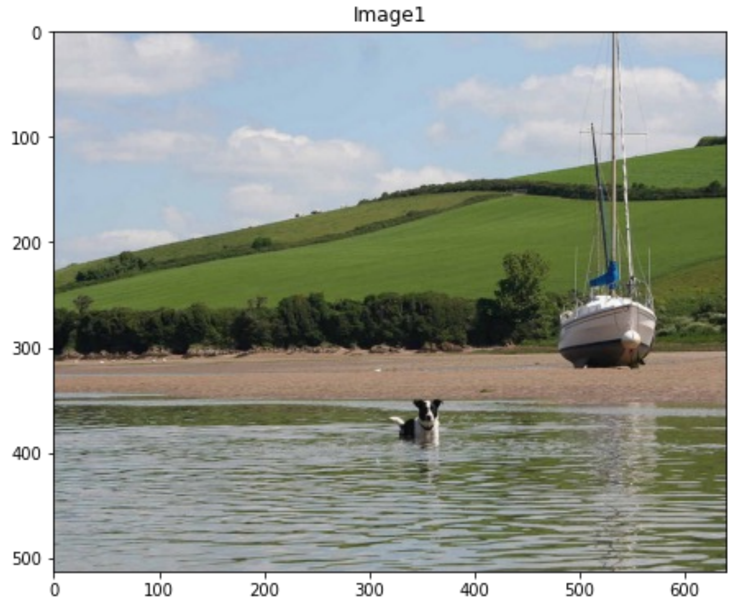
Image encoder – Image data



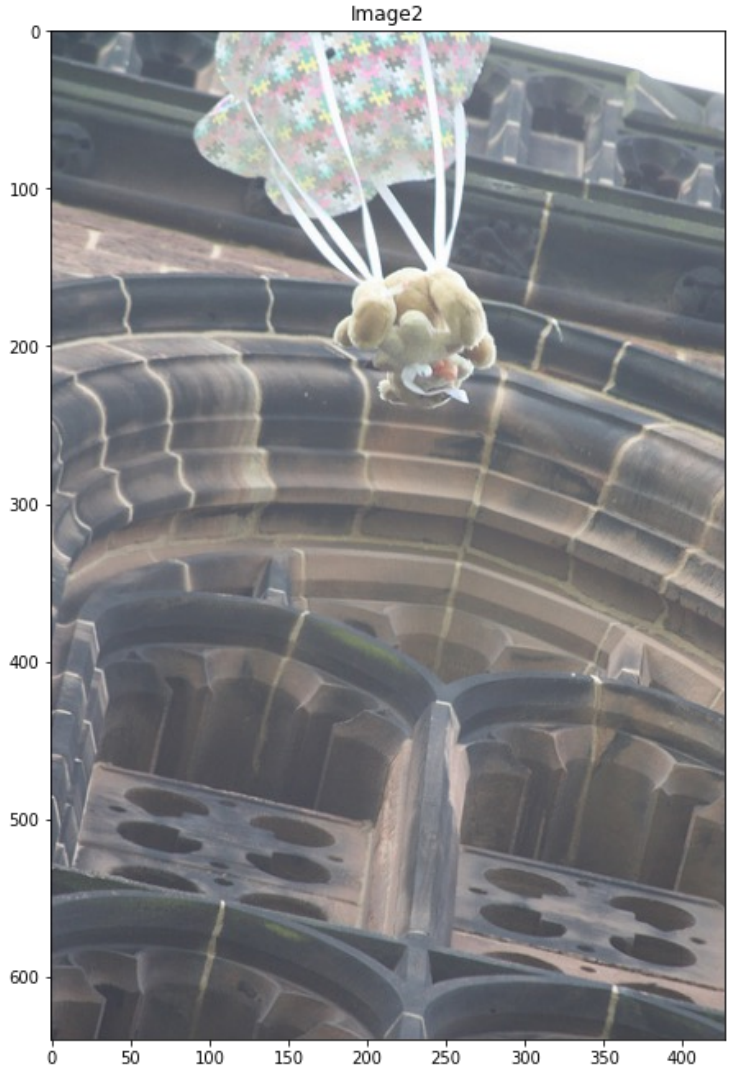
Code

Image encoder – Image data

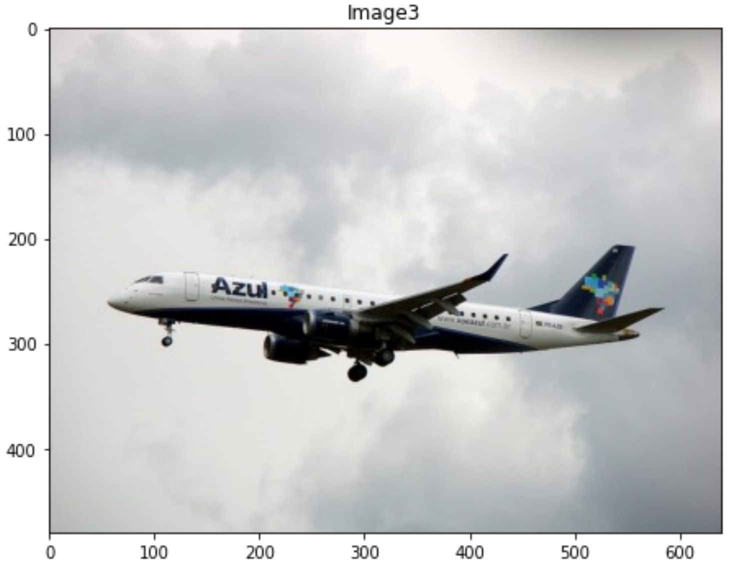
"The dog is swimming in the lake on a sunny day"



"a stuffed animal with a parachute falling from above"

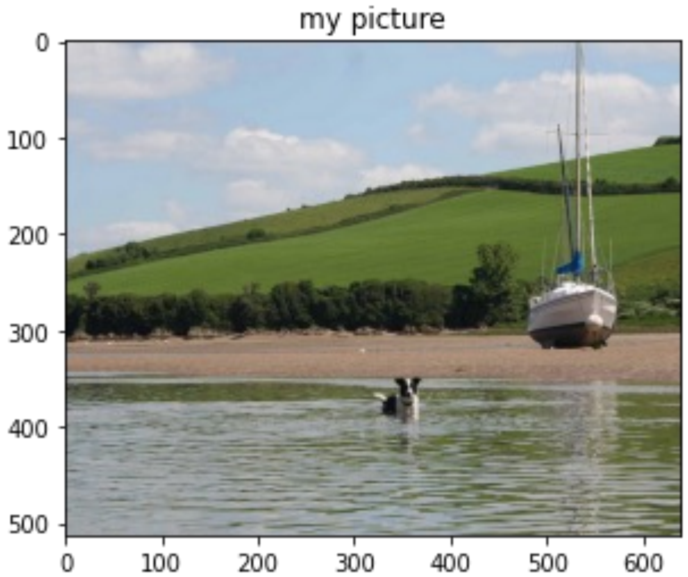


"A larger commerical jet is flying in the air"

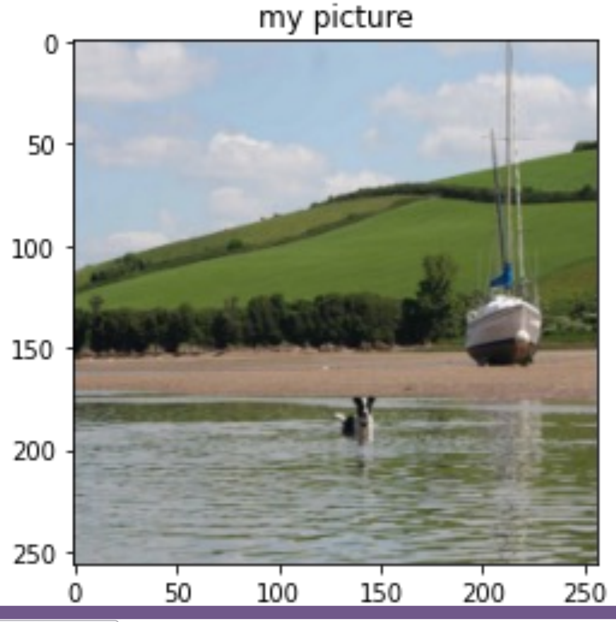


Code

Image encoder – Image data




```
(513, 640, 3)
```




```
torch.Size([3, 256, 256])
torch.Size([1, 3, 256, 256])
```

```python
image_one = "./data/caption_image1.jpg"
plt.imshow(plt.imread(image_one))
plt.title('my picture')
plt.show()
print(plt.imread(image_one).shape)

image_one = Image.open(image_one)
transform = Compose([Resize((256, 256))])
image_one = transform(image_one)
plt.imshow(image_one)
plt.title('my picture')
plt.show()


transform = Compose([ToTensor()])
image_one = transform(image_one)
print(image_one.shape)

image_one = image_one.unsqueeze(0)
print(image_one.shape)
```

Code

Image encoder – Image data(3개의 그림에 적용)

```python
Image_list = []
transform = Compose([Resize((256, 256)), ToTensor()])
for i in range(len(img_list)):
    image = Image.open(os.path.join(img_dir, img_list[i]))
    image = transform(image)
    image = image.unsqueeze(0)
    Image_list.append(image)

image_input = torch.concat(Image_list, dim = 0)
print(image_input.shape)
```

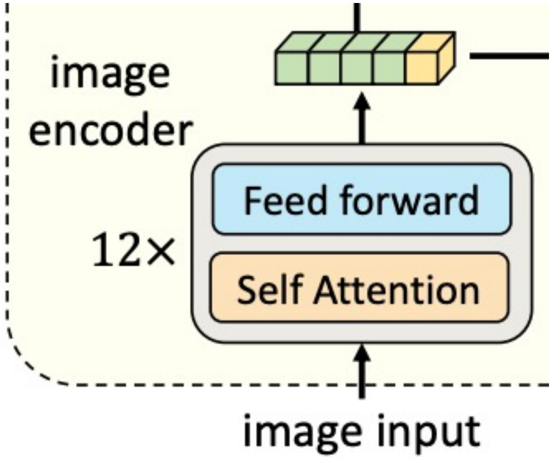
```
torch.Size([3, 3, 256, 256])
```

Code

Image encoder – input data of Image encoder

As illustrated in Figure 1, ALBEF contains an image encoder, a text encoder, and a multimodal encoder. We use a 12-layer visual transformer ViT-B/16 [38] as the image encoder, and initialize it with weights pre-trained on ImageNet-1k from [31]. An input image I is encoded into a sequence of embeddings: $\{\boldsymbol{v}_{\mathrm{cls}}, \boldsymbol{v}_1, ..., \boldsymbol{v}_N\}$, where v_{cls} is the embedding of the `[CLS]` token. We use a 6-layer transformer [39] for both the text encoder and the multimodal encoder. The text encoder is initialized using the first 6 layers of the BERT$_{\mathrm{base}}$ [40] model, and the multimodal encoder is initialized using the last 6 layers of the BERT$_{\mathrm{base}}$. The text encoder transforms an input text T into a sequence of embeddings $\{\boldsymbol{w}_{\mathrm{cls}}, \boldsymbol{w}_1, ..., \boldsymbol{w}_N\}$, which is fed to the multimodal encoder. The image features are fused with the text features through cross attention at each layer of the multimodal encoder.

```
torch.Size([3, 3, 256, 256])
```



Code

Image encoder – ViT model & Initializing projection layer

```
embed_dim = 256
vision_width = 768
image_res = 256
visual_encoder = VisionTransformer(
    img_size=image_res, patch_size=16, embed_dim=768, depth=12, num_heads=12,
    mlp_ratio=4, qkv_bias=True, norm_layer=partial(nn.LayerNorm, eps=1e-6))

vision_proj = nn.Linear(vision_width, embed_dim) # [768, 256]
```

Code

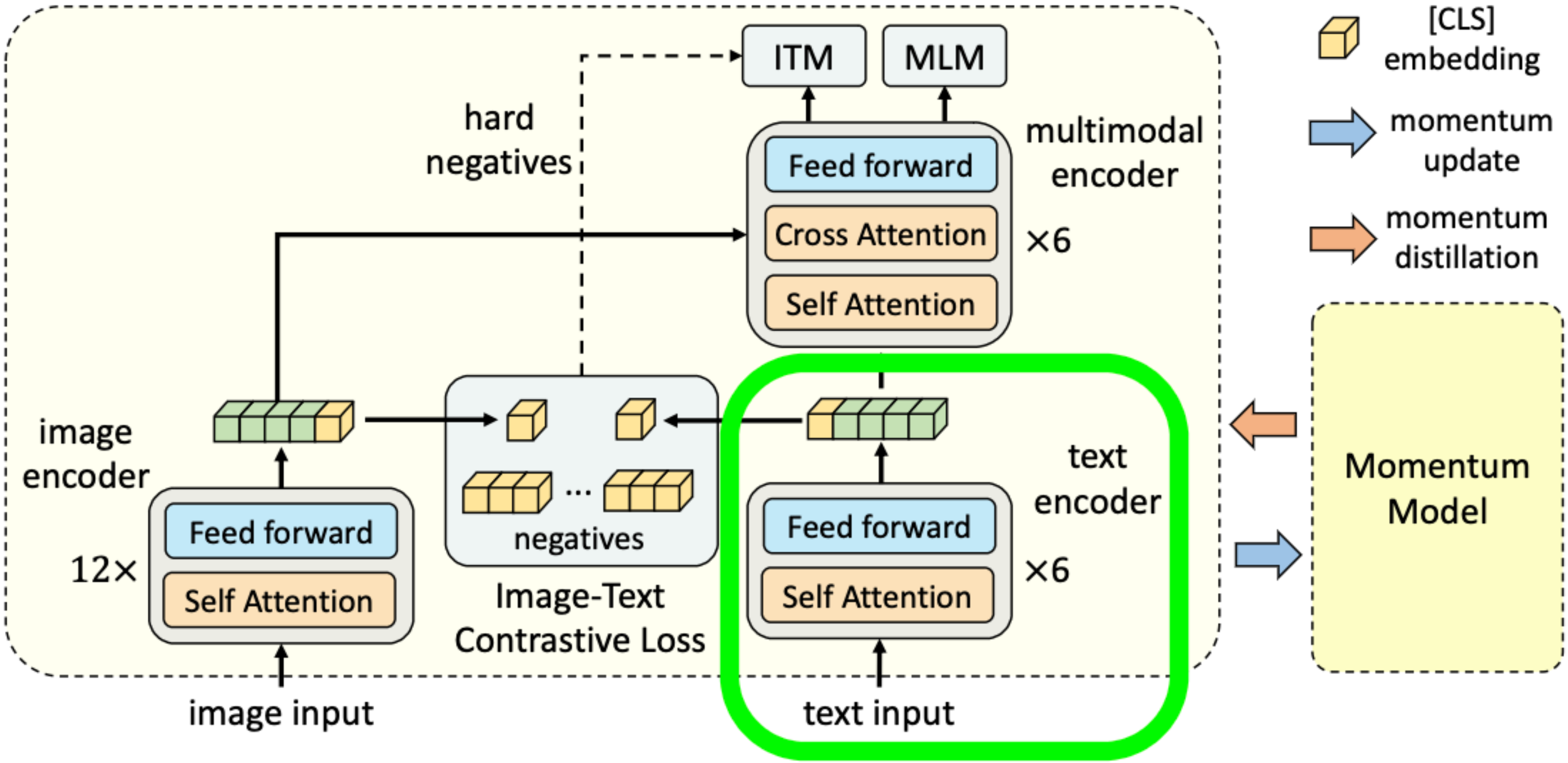
Image encoder – Output of ViT model

$$\text{입력이미지 } I \rightarrow \{\overrightarrow{v_{cls}}, \overrightarrow{v_1}, \ldots, \overrightarrow{v_N}\}$$

```
image_embeds = visual_encoder(image_input) [3, 257, 768]
print(image_embeds[:,0,:].shape) [3, 768]
image_feat = F.normalize(vision_proj(image_embeds[:,0,:]),dim=-1)[3, 256]

[3, 257]
image_atts = torch.ones(image_embeds.size()[:-1],dtype=torch.long).to(image.device)
```

Code

Text encoder



Code

Text encoder $BERT_{base}$ model & Initializing projection layer

As illustrated in Figure 1, ALBEF contains an image encoder, a text encoder, and a multimodal encoder. We use a 12-layer visual transformer ViT-B/16 [38] as the image encoder, and initialize it with weights pre-trained on ImageNet-1k from [31]. An input image I is encoded into a sequence of embeddings: $\{\boldsymbol{v}_{\text{cls}}, \boldsymbol{v}_1, ..., \boldsymbol{v}_N\}$, where v_{cls} is the embedding of the [CLS] token. We use a 6-layer transformer [39] for both the text encoder and the multimodal encoder. The text encoder is initialized using the first 6 layers of the $\text{BERT}_{\text{base}}$ [40] model, and the multimodal encoder is initialized using the last 6 layers of the $\text{BERT}_{\text{base}}$. The text encoder transforms an input text T into a sequence of embeddings $\{\boldsymbol{w}_{\text{cls}}, \boldsymbol{w}_1, ..., \boldsymbol{w}_N\}$, which is fed to the multimodal encoder. The image features are fused with the text features through cross attention at each layer of the multimodal encoder.

```
embed_dim = 256
text_width = 768
bert_config = BertConfig.from_json_file('../configs/config_bert.json')
text_encoder = BertForMaskedLM.from_pretrained('bert-base-uncased', config=bert_config)
text_proj = nn.Linear(text_width, embed_dim)
```

Code

Tokenizing text data

```
from transformers import BertTokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-cased')
text = tokenizer("The dog is swimming in the lake on a sunny day")
print("text:", text)
```

```
text: {'input_ids': [101, 1109, 3676, 1110, 5947, 1107, 1103, 3521, 1113,
170, 21162, 1285, 102],
        'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

Code

Tokenized text data into model

```
text.input_ids = torch.Tensor(text.input_ids).to(torch.long)
text.input_ids = text.input_ids.unsqueeze(dim=0)
print("text.input_id:", text.input_ids.shape) torch.Size([1, 13]

text.attention_mask = torch.Tensor(text.attention_mask)
text.attention_mask = text.attention_mask.unsqueeze(dim=0)
print("text.attention_mask:", text.attention_mask.shape) torch.Size([1, 13])

text_output = text_encoder.bert(torch.Tensor(text.input_ids), attention_mask = torch.Tensor(text.attention_mask),
                                return_dict = True, mode = 'text')

text_embeds = text_output.last_hidden_state
print("text_embeds", text_embeds.shape) torch.Size([1, 13, 768])
text_feat = F.normalize(text_proj(text_embeds[:,0,:]),dim=-1)
print("text_feat", text_feat.shape) torch.Size([1, 256])
```

Code

Tokenized text data into model

$$\{\overrightarrow{w_{cls}}, \overrightarrow{w_1}, \ldots, \overrightarrow{w_N}\}$$

```
torch.Size([3, 13])
torch.Size([3, 13])
```

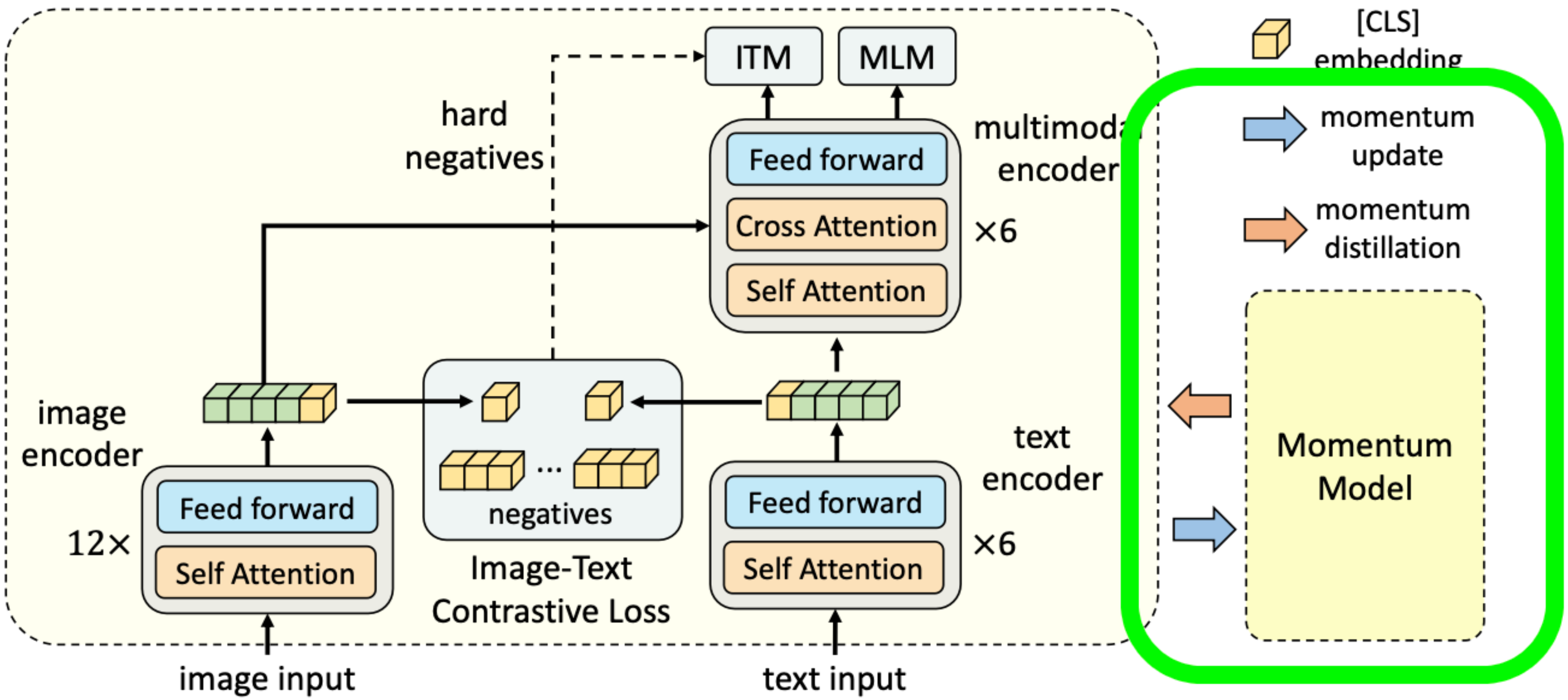
```
torch.Size([3, 13, 768])
```

```
torch.Size([3, 256])
```

```python
text1 = "The dog is swimming in the lake on a sunny day"
text2 = "a stuffed animal with a parachute falling from above"
text3 = "A larger commerical jet is flying in the air"

from transformers import BertTokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-cased')
text = tokenizer([text1, text2, text3], padding=True)

text.input_ids = torch.Tensor(text.input_ids).to(torch.long)
text.attention_mask = torch.Tensor(text.attention_mask)

print("text.input_id", text.input_ids.shape)
print("text.attention_mask", text.attention_mask.shape)

text_output = text_encoder.bert(text.input_ids, text.attention_mask,
                                return_dict = True, mode = 'text')

text_embeds = text_output.last_hidden_state
print("text_embeds.shape", text_embeds.shape)
text_feat = F.normalize(text_proj(text_embeds[:,0,:]),dim=-1)
print("text_feat", text_feat.shape)
```

Code

Momentum Model Initialization




```python
# create momentum models
visual_encoder_m = VisionTransformer(
    img_size=image_res, patch_size=16, embed_dim=768, depth=12, num_heads=12,
    mlp_ratio=4, qkv_bias=True, norm_layer=partial(nn.LayerNorm, eps=1e-6))
vision_proj_m = nn.Linear(vision_width, embed_dim)
text_encoder_m = BertForMaskedLM.from_pretrained('bert-base-uncased', config=bert_config)
text_proj_m = nn.Linear(text_width, embed_dim)
```

Code

Momentum Model update

```
model_pairs =  [[visual_encoder, visual_encoder_m],
                [vision_proj, vision_proj_m],
                [text_encoder, text_encoder_m],
                [text_proj,text_proj_m],
               ]

momentum= 0.995
def momentum_update(model_pairs):
    for model_pair in model_pairs:
        for param, param_m in zip(model_pair[0].parameters(), model_pair[1].parameters()):
            param_m.data = param_m.data * momentum + param.data * (1. - momentum)
```

$$V_t = \beta \times V_{t-1} + (1-\beta) \times \Theta_t$$

Code

Queue

Image-Text Contrastive Learning aims to learn better unimodal representations before fusion. It learns a similarity function $s = g_v(\boldsymbol{v}_{\text{cls}})^\top g_w(\boldsymbol{w}_{\text{cls}})$, such that parallel image-text pairs have higher similarity scores. g_v and g_w are linear transformations that map the `[CLS]` embeddings to normalized lower-dimensional (256-d) representations. Inspired by MoCo [24], we maintain two queues to store the most recent M image-text representations from the momentum unimodal encoders. The normalized features from the momentum encoders are denoted as $g'_v(\boldsymbol{v}'_{\text{cls}})$ and $g'_w(\boldsymbol{w}'_{\text{cls}})$. We define $s(I,T) = g_v(\boldsymbol{v}_{\text{cls}})^\top g'_w(\boldsymbol{w}'_{\text{cls}})$ and $s(T,I) = g_w(\boldsymbol{w}_{\text{cls}})^\top g'_v(\boldsymbol{v}'_{\text{cls}})$.

Code

Dequeue and enqueue

```python
@torch.no_grad()
def _dequeue_and_enqueue(self, image_feat, text_feat):
    # gather keys before updating queue
    image_feats = concat_all_gather(image_feat)
    text_feats = concat_all_gather(text_feat)

    batch_size = image_feats.shape[0]

    ptr = int(self.queue_ptr)
    assert self.queue_size % batch_size == 0  # for simplicity

    # replace the keys at ptr (dequeue and enqueue)
    self.image_queue[:, ptr:ptr + batch_size] = image_feats.T
    self.text_queue[:, ptr:ptr + batch_size] = text_feats.T
    ptr = (ptr + batch_size) % self.queue_size  # move pointer

    self.queue_ptr[0] = ptr
```

Code

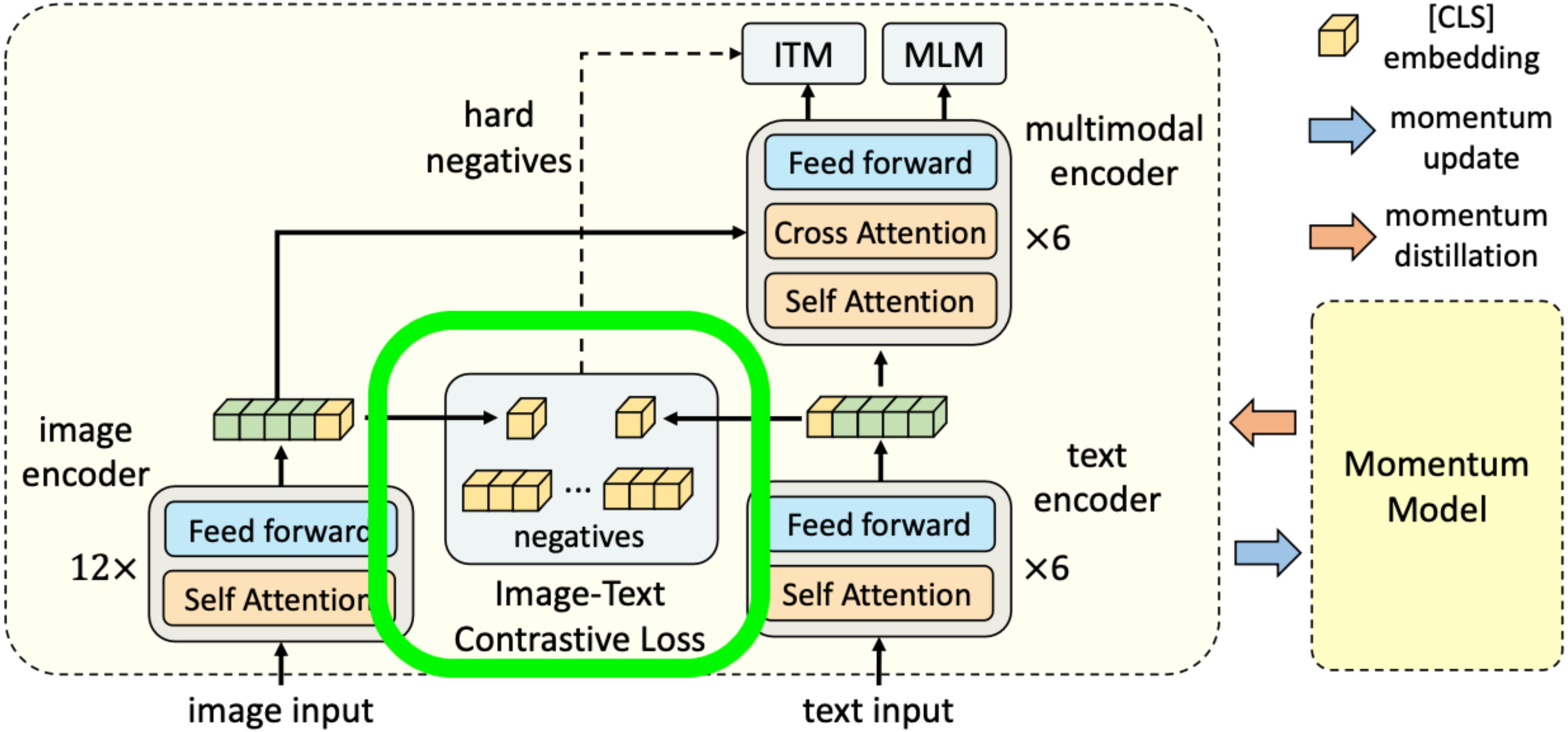
Momentum Model update

```
queue_size = 65536
randn = 256
embed_dim = 256

image_queue = torch.randn(randn, queue_size)
text_queue = torch.randn(embed_dim, queue_size)
image_queue = nn.functional.normalize(image_queue, dim=0)
text_queue = nn.functional.normalize(text_queue, dim=0)
print('image_queue, text_queue:', image_queue.shape, text_queue.shape)
```

```
image_queue, text_queue: torch.Size([256, 65536]) torch.Size([256, 65536])
```

Code

ITC


ITM
MLM
hard negatives
multimodal encoder
Feed forward
Cross Attention
Self Attention
×6
image encoder
12×
Feed forward
Self Attention
negatives
Image-Text Contrastive Loss
text encoder
Feed forward
Self Attention
×6
image input
text input
[CLS] embedding
momentum update
momentum distillation
Momentum Model

Code

ITC Loss

$$p_m^{\mathrm{i2t}}(I) = \frac{\exp(s(I, T_m)/\tau)}{\sum_{m=1}^{M} \exp(s(I, T_m)/\tau)}, \quad p_m^{\mathrm{t2i}}(T) = \frac{\exp(s(T, I_m)/\tau)}{\sum_{m=1}^{M} \exp(s(T, I_m)/\tau)}$$

$$\mathcal{L}_{\mathrm{itc}} = \frac{1}{2}\mathbb{E}_{(I,T)\sim D}\big[\mathrm{H}(\boldsymbol{y}^{\mathrm{i2t}}(I), \boldsymbol{p}^{\mathrm{i2t}}(I)) + \mathrm{H}(\boldsymbol{y}^{\mathrm{t2i}}(T), \boldsymbol{p}^{\mathrm{t2i}}(T))\big]$$

$$\mathcal{L}_{\mathrm{itc}}^{\mathrm{mod}} = (1-\alpha)\mathcal{L}_{\mathrm{itc}} + \frac{\alpha}{2}\mathbb{E}_{(I,T)\sim D}\big[\mathrm{KL}(\boldsymbol{q}^{\mathrm{i2t}}(I) \parallel \boldsymbol{p}^{\mathrm{i2t}}(I)) + \mathrm{KL}(\boldsymbol{q}^{\mathrm{t2i}}(T) \parallel \boldsymbol{p}^{\mathrm{t2i}}(T))\big]$$

Code

ITC Loss

```
temp = 0.07
alpha = 0.4

# get momentum features
with torch.no_grad():
    momentum_update(model_pairs)
    image_embeds_m = visual_encoder_m(image_input)  # [3, 257, 768]
    image_feat_m = F.normalize(vision_proj_m(image_embeds_m[:,0,:]),dim=-1) # [3, 256]
    image_feat_all = torch.cat([image_feat_m.t(), image_queue.clone().detach()],dim=1) # cat([256, 3], [256, 65536])=[256, 65539]


    text_output_m = text_encoder_m.bert(text.input_ids, attention_mask = text.attention_mask,
                                        return_dict = True, mode = 'text')  # [3, 13, 768]

    text_feat_m = F.normalize(text_proj_m(text_output_m.last_hidden_state[:,0,:]),dim=-1) # [3, 256]
    text_feat_all = torch.cat([text_feat_m.t(), text_queue.clone().detach()],dim=1) # [256, 65539]

    sim_i2t_m = image_feat_m @ text_feat_all / temp # [3, 256] @ [256, 65539] = [3, 65539]
    sim_t2i_m = text_feat_m @ image_feat_all / temp # [3, 65539]

    sim_targets = torch.zeros(sim_i2t_m.size()).to(image.device) # [3, 65539]
    sim_targets.fill_diagonal_(1)

    sim_i2t_targets = alpha * F.softmax(sim_i2t_m, dim=1) + (1 - alpha) * sim_targets # [3, 65539]
    sim_t2i_targets = alpha * F.softmax(sim_t2i_m, dim=1) + (1 - alpha) * sim_targets # [3, 65539]

    print_size(image_embeds_m.shape, image_feat_m.shape, image_feat_all.shape, text_output_m.last_hidden_state.shape,
               text_feat_m.shape, text_feat_all.shape, sim_i2t_m.shape, sim_t2i_m.shape,
               sim_targets, sim_i2t_targets.shape, sim_t2i_targets.shape)
################
```

Code

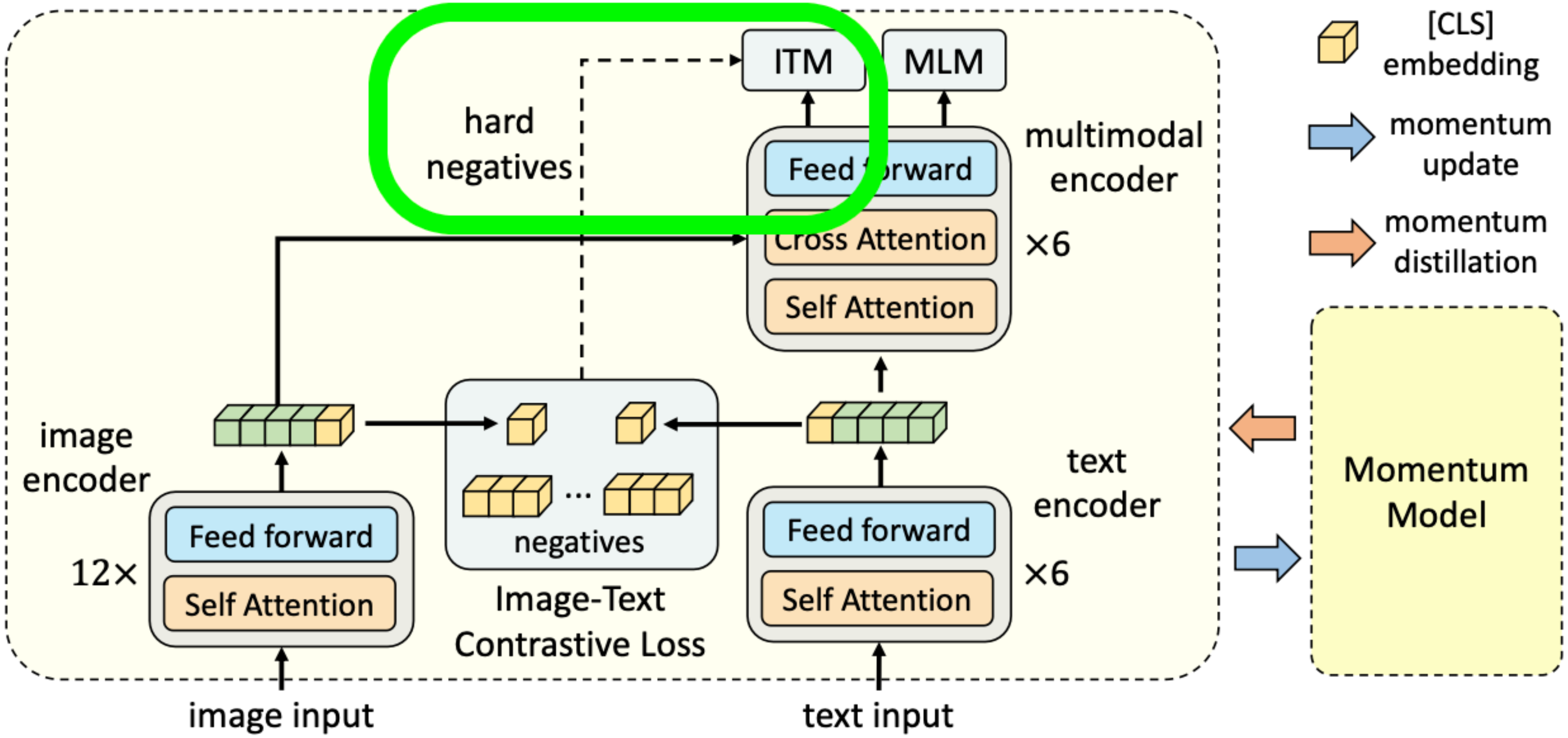
ITC Loss

```python
sim_i2t = image_feat @ text_feat_all / temp # [3, 65539]
sim_t2i = text_feat @ image_feat_all / temp # [3, 65539]
print('sim_i2t:', sim_i2t.shape)
print('sim_t2i:', sim_t2i.shape)

loss_i2t = -torch.sum(F.log_softmax(sim_i2t, dim=1)*sim_i2t_targets,dim=1).mean()
loss_t2i = -torch.sum(F.log_softmax(sim_t2i, dim=1)*sim_t2i_targets,dim=1).mean()
print('loss_i2t:', loss_i2t)
print('loss_t2i:', loss_t2i)

loss_ita = (loss_i2t+loss_t2i)/2
print("loss_ita:", loss_ita)
# dequeue_and_enqueue(image_feat_m, text_feat_m, image_queue, text_queue)
```

$$\mathcal{L}_{\text{itc}} = \frac{1}{2}\mathbb{E}_{(I,T)\sim D}\left[\mathrm{H}(\boldsymbol{y}^{\text{i2t}}(I), \boldsymbol{p}^{\text{i2t}}(I)) + \mathrm{H}(\boldsymbol{y}^{\text{t2i}}(T), \boldsymbol{p}^{\text{t2i}}(T))\right]$$

Code

ITM



We propose a strategy to sample hard negatives for the ITM task with zero computational overhead. A negative image-text pair is hard if they share similar semantics but differ in fine-grained details. We use the contrastive similarity from Equation 1 to find in-batch hard negatives. For each image in a mini-batch, we sample one negative text from the same batch following the contrastive similarity distribution, where texts that are more similar to the image have a higher chance to be sampled. Likewise, we also sample one hard negative image for each text.

$$\mathcal{L}_{\text{itm}} = \mathbb{E}_{(I,T)\sim D}\text{H}(\boldsymbol{y}^{\text{itm}}, \boldsymbol{p}^{\text{itm}}(I,T))$$

Code

ITM Loss

```
###==================================###
# forward the positve image-text pair
with torch.no_grad():
    bs = image_input.size(0)
    weights_i2t = F.softmax(sim_i2t[:,:bs],dim=1)
    weights_t2i = F.softmax(sim_t2i[:,:bs],dim=1)

    weights_i2t.fill_diagonal_(0)
    weights_t2i.fill_diagonal_(0)
```

```
weights_i2t:
tensor([[0.2300, 0.4070, 0.3630],
        [0.2286, 0.4069, 0.3645],
        [0.2177, 0.4233, 0.3590]])
weights_t2i:
tensor([[0.2785, 0.3038, 0.4177],
        [0.2698, 0.2980, 0.4321],
        [0.2718, 0.2924, 0.4358]])
```

```
tensor([[0.0000, 0.3038, 0.4177],
        [0.2698, 0.0000, 0.4321],
        [0.2718, 0.2924, 0.0000]])
```

Code

ITM Loss

```
# select a negative image for each text
image_embeds_neg = []
for b in range(bs):
    neg_idx = torch.multinomial(weights_t2i[b], 1).item()
    image_embeds_neg.append(image_embeds[neg_idx])
image_embeds_neg = torch.stack(image_embeds_neg,dim=0)
```

```
torch.Size([3, 257, 768])
```

```
tensor([[0.0000, 0.3038, 0.4177],
        [0.2698, 0.0000, 0.4321],
        [0.2718, 0.2924, 0.0000]])

neg_idx_t2i 1
neg_idx_t2i 2
neg_idx_t2i 1
```

```
# select a negative text for each image
text_embeds_neg = []
text_atts_neg = []
for b in range(bs):
    neg_idx = torch.multinomial(weights_i2t[b], 1).item()
    text_embeds_neg.append(text_embeds[neg_idx])
    text_atts_neg.append(text.attention_mask[neg_idx])
text_embeds_neg = torch.stack(text_embeds_neg,dim=0)
text_atts_neg = torch.stack(text_atts_neg,dim=0)
```

```
torch.Size([3, 13, 768])
```

```
weights_i2t:
tensor([[0.0000, 0.4070, 0.3630],
        [0.2286, 0.0000, 0.3645],
        [0.2177, 0.4233, 0.0000]])

neg_idx_i2t 2
neg_idx_i2t 2
neg_idx_i2t 1
```

Code

ITM Loss

```
text_embeds_all = torch.cat([text_embeds, text_embeds_neg],dim=0)
text_atts_all = torch.cat([text.attention_mask, text_atts_neg],dim=0)

image_embeds_all = torch.cat([image_embeds_neg,image_embeds],dim=0)
image_atts_all = torch.cat([image_atts,image_atts],dim=0)

output_pos = text_encoder.bert(encoder_embeds = text_embeds,
                               attention_mask = text.attention_mask,
                               encoder_hidden_states = image_embeds,
                               encoder_attention_mask = image_atts,
                               return_dict = True,
                               mode = 'fusion',
                               )

output_neg = text_encoder.bert(encoder_embeds = text_embeds_all,
                               attention_mask = text_atts_all,
                               encoder_hidden_states = image_embeds_all,
                               encoder_attention_mask = image_atts_all,
                               return_dict = True,
                               mode = 'fusion',
                               )
```

```
torch.Size([6, 13, 768])
torch.Size([6, 13])
torch.Size([6, 257, 768])
torch.Size([6, 257])
torch.Size([3, 13, 768])
torch.Size([6, 13, 768])
```

Code

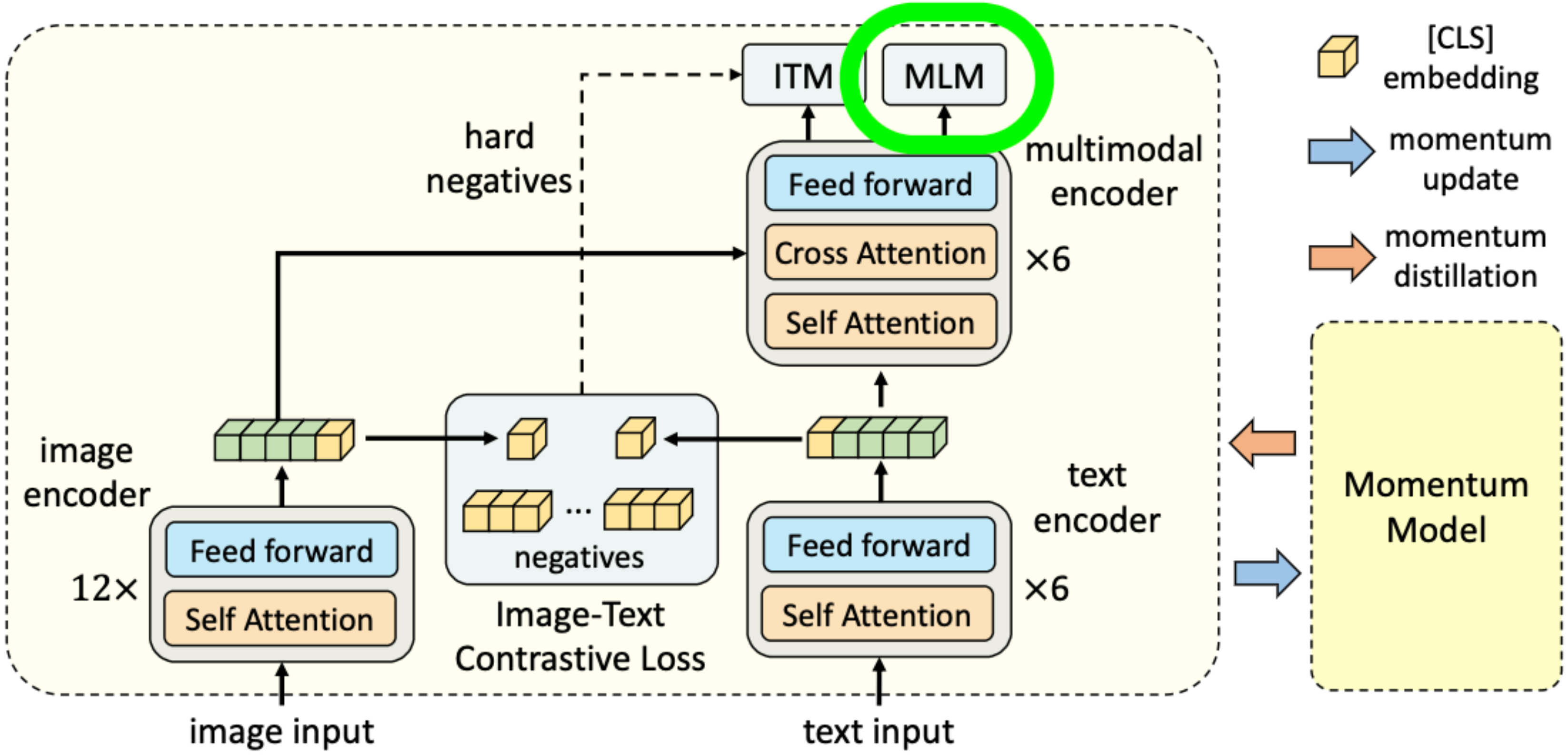
ITM Loss

```
vl_embeddings = torch.cat([output_pos.last_hidden_state[:,0,:], output_neg.last_hidden_state[:,0,:]],dim=0)
itm_head = nn.Linear(text_width, 2)
vl_output = itm_head(vl_embeddings)

itm_labels = torch.cat([torch.ones(bs,dtype=torch.long),torch.zeros(2*bs,dtype=torch.long)],
                       dim=0).to(image.device)
loss_itm = F.cross_entropy(vl_output, itm_labels)
```

```
vl_embeddings: torch.Size([9, 768])
vl_output tensor([[-0.1870, -0.3388],
        [ 0.1340,  0.0165],
        [ 0.2440, -0.0590],
        [-0.2321, -0.3385],
        [ 0.2185, -0.0172],
        [ 0.0896, -0.0488],
        [ 0.1489,  0.0021],
        [-0.2321, -0.3385],
        [-0.0964, -0.3057]], grad_fn=<AddmmBackward0>)
itm_labels tensor([1, 1, 1, 0, 0, 0, 0, 0, 0])
loss_itm: tensor(0.6862, grad_fn=<NllLossBackward0>)
```

Code

MLM



Masked Language Modeling utilizes both the image and the contextual text to predict the masked words. We randomly mask out the input tokens with a probability of 15% and replace them with the special token `[MASK]`[1]. Let $\hat{T}$ denote a masked text, and $\boldsymbol{p}^{\text{msk}}(I, \hat{T})$ denote the model's predicted probability for a masked token. MLM minimizes a cross-entropy loss:

$$\mathcal{L}_{\text{mlm}} = \mathbb{E}_{(I,\hat{T})\sim D} \mathrm{H}(\boldsymbol{y}^{\text{msk}}, \boldsymbol{p}^{\text{msk}}(I, \hat{T})) \quad (3)$$

Code

MLM

```python
def mask(input_ids, vocab_size, device, targets=None, masked_indices=None, probability_matrix=None):
    if masked_indices is None:
        masked_indices = torch.bernoulli(probability_matrix).bool()

    masked_indices[input_ids == tokenizer.pad_token_id] = False
    masked_indices[input_ids == tokenizer.cls_token_id] = False

    if targets is not None:
        targets[~masked_indices] = -100 # We only compute loss on masked tokens

    # 80% of the time, we replace masked input tokens with tokenizer.mask_token ([MASK])
    indices_replaced = torch.bernoulli(torch.full(input_ids.shape, 0.8)).bool() & masked_indices
    input_ids[indices_replaced] = tokenizer.mask_token_id

    # 10% of the time, we replace masked input tokens with random word
    indices_random = torch.bernoulli(torch.full(input_ids.shape, 0.5)).bool() & masked_indices & ~indices_replaced
    random_words = torch.randint(vocab_size, input_ids.shape, dtype=torch.long).to(device)
    input_ids[indices_random] = random_words[indices_random]
    # The rest of the time (10% of the time) we keep the masked input tokens unchanged

    if targets is not None:
        return input_ids, targets
    else:
        return input_ids
```

Code

MLM

```
##================== MLM =========================##
input_ids = text.input_ids.clone()
labels = input_ids.clone()
mlm_probability = 0.15

probability_matrix = torch.full(labels.shape, mlm_probability)
input_ids, labels = mask(input_ids, text_encoder.config.vocab_size, image.device, targets=labels,
                         probability_matrix = probability_matrix)

with torch.no_grad():
    logits_m = text_encoder_m(input_ids,
                              attention_mask = text.attention_mask,
                              encoder_hidden_states = image_embeds_m,
                              encoder_attention_mask = image_atts,
                              return_dict = True,
                              return_logits = True,
                              )
mlm_output = text_encoder(input_ids,
                          attention_mask = text.attention_mask,
                          encoder_hidden_states = image_embeds,
                          encoder_attention_mask = image_atts,
                          return_dict = True,
                          labels = labels,
                          soft_labels = F.softmax(logits_m,dim=-1),
                          alpha = alpha
                          )
loss_mlm = mlm_output.loss

loss_total = loss_ita + loss_mlm + loss_itm
```

$$\mathcal{L} = \mathcal{L}_{itc} + \mathcal{L}_{mlm} + \mathcal{L}_{itm}$$

```
torch.Size([3, 13])
torch.Size([3, 13])
torch.Size([3, 13])
```

```
probability_matrix:
tensor([[0.1500, 0.1500, 0.1500, 0.1500, 0.1500, 0.1500, 0.1500, 0.1500, 0.1500,
         0.1500, 0.1500, 0.1500, 0.1500],
        [0.1500, 0.1500, 0.1500, 0.1500, 0.1500, 0.1500, 0.1500, 0.1500, 0.1500,
         0.1500, 0.1500, 0.1500, 0.1500],
        [0.1500, 0.1500, 0.1500, 0.1500, 0.1500, 0.1500, 0.1500, 0.1500, 0.1500,
         0.1500, 0.1500, 0.1500, 0.1500]])
```

Thank you