VTT 모델 구현 가이드라인

이 문서는 VTT pipeline의 모델 구현 가이드라인입니다. VTT pipeline의 통일성과 성능 개선에 필요한 내용이 포함되어 있습니다. 아래 모델 구현 관련 추천 내용은 R0, R1, ..., R5으로 표시했습니다.

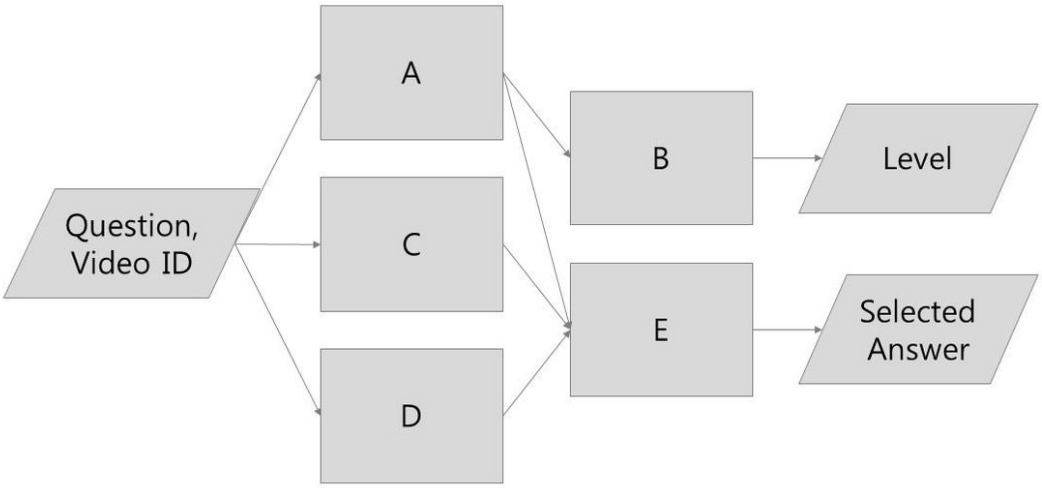
- 작성: 서울대학교 소프트웨어 플랫폼 랩
- 날짜: 2020.04.02
- 버전: v1.0

1. 현재 구축된 VTT Pipeline의 구조

아래는 VTT Pipeline의 구성 모델들입니다. 코드 수합 당시(2019년 12월) 실행이 안되는 모델은 포함되어 있지 않습니다.

- 모델 A - 비디오 스토리 구축 모델
- 모델 B - Level Classification 모델 (모델 B에서 결정한 Level은 현재 pipeline에서 사용되지 않고 있습니다.)
- 모델 C - High-Level QA 모델
- 모델 D - Knowledge-Based QA 모델
- 모델 E - 답안 선택 모델

모델 A~E 간의 dependency는 아래와 같습니다:



2. 개발 container 환경 가이드라인

R0. 개발 환경을 통일합니다.

VTT pipeline 전체는 최종적으로 하나의 docker container를 사용합니다. 하나의 docker container 환경에서 전체 모델을 수행할 수 있도록 연구실 및 기관에서는 논의를 통해 구체적인 개발 환경을 통일하기를 권장합니다.

아래는 2019년 12월에 만든 통합 container 환경 예시입니다:

- OS: Ubuntu 18.04.3 LTS
- Python: 3.6.8

- Deep Learning Framework
  - TensorFlow: 1.15.0
  - PyTorch: 1.2.0
- CUDA Toolkit: 10.0

3. 모델 코드 작성 가이드라인

R1. 모델 실행 함수는 초기화하는 부분과 추론하는 부분으로 나눕니다.

모델 실행 함수는 `init`과 `predict` 함수로 명확히 분리합니다. `init` 함수는 모델을 처음 실행할 때만 불리고, 실제 추론 시에는 `predict` 함수만 불리는 구조로 만듭니다.

- **class Model**
  - **init()**: 모델을 초기화합니다.
    - 모델 load, QA data 및 tokenizer load 등 초기화 작업을 실행합니다.
    - 초기화한 모델과 데이터는 클래스의 멤버로 갖습니다.
  - **predict(input)**: 모델 추론 작업을 합니다.
    - Parameter:
      - input: 추론 input (질문, 비디오 스토리 등)
    - Return:
      - output: 추론 output (비디오 스토리, 답안 등)

예로 현재 VTT 모델 중에 초기화 부분을 분리하지 않은 코드를 분리한 코드로 바꾼 경우입니다:

- 변경전

```
## File: LevelClassificationModel.py
def printPredictionLevel(input_a, input_b):
  memory_output = MemoryLevel.Memory_level_model(input_a, input_b)
  ...
  return memory_output, ...
```

```
## File: MemoryLevel.py
def Memory_level_model(input_a, input_b):
  ...
  # data load
  tokenizer = tokenization.FullTokenizer(vocab_file=...)
  # model definition
  model = BertForSequenceClassification(bert_config,len(label_list))
  # model load
  model.to(device)
  ...
  # prediction
  logits = model(input_ids, segment_ids, input_mask, label_ids)
  ...
  return output
```

- 변경후

```
## File: LevelClassificationModel.py
class LevelClassificationModel():
  def __init__(self):
```

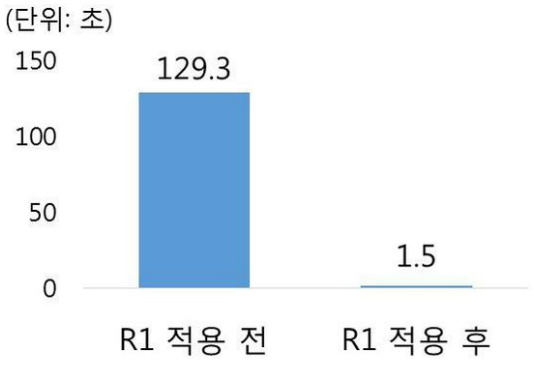
```
    self.memory_level = MemoryLevel.MemoryLevelModel()
    ...
  def predict(self, input_a, input_b):
    memory_output = self.memory_level.predict(input_a, input_b)
    ...
    return memory_output, ...
```

```
## File: MemoryLevel.py
class MemoryLevelModel():
  def __init__(self):
    # data load
    self.tokenizer = tokenization.FullTokenizer(vocab_file=...)
    # model definition
    self.model = BertForSequenceClassification(bert_config,len(label_list))
    # model load
    self.model.to(self.device)
    ...
  def predict(self, input_a, input_b):
    ...
    # prediction
    logits = model(input_ids, segment_ids, input_mask, label_ids)
    ...
    return output
```

R1 적용에 따른 Pipeline 수행 시간 변화와 각 모델의 초기화 시간은 아래 그래프, 표와 같습니다:

- R1 부터 나오는 수행 시간 측정은 다음 환경에서 진행하였습니다.
  - CPU: 2 18-core Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
  - GPU: 1 NVIDIA Titan Xp
  - Memory: 256 GB

(A): 초기화 분리 후 VTT Pipeline 수행 시간 변화 (단위: 초)



(B): 각 모델 별 초기화에 걸리는 시간 (단위: 초)

모델	초기화 시간 (s)
A	57.25
B	2.95
C	10.02
D	DNN 사용 X
E	57.58
합	**127.8**

R2. Input을 이용하여 추론 모델을 생성하지 않습니다.

Input을 이용하여 모델을 생성하는 경우는 input을 사용하지 않고 모델을 생성할 수 있도록 변경합니다. R2를 따라야 R1을 따를 수 있습니다.

- 아래는 input이 들어온 후 모델을 만드는 예시와 이를 수정한 예시입니다:
  - 변경전

- 붉은 영역은 `input`에 dependent한 변수들입니다.
- `predict()`의 `Model_Selector`에서는 해당 변수들을 이용하여 모델을 만들고 있습니다.

```
## File: predict.py
def main():
  input = clientSock.recv(4096)
  ...
  data_list = _text2data(input)
  data = VTTExample(data_list, 'prediction', model_params, dicts, False)
  final_answer = predict(data, model_params)

def predict(data, model_params, ...):
  # generate model
  with tf.variable_scope('model') as scope:
    model = Model_Selector (data.emb_mat_token, data.emb_mat_glove,
                            len(data.dicts['token']), len(data.dicts['char']),
                            data.max_token_size, data.max_ans_size, ...)
  # predict answer
  ...
```

```
## File: dataset.py
class VTTExample(object):
  def __init__(self, data_list, data_type, model_params, dicts, is_binary):
    max_lens = self.find_data_max_length(data_list)
    self.max_token_size = min(max_lens['token'], model_params.max_token_size)
    self.max_ans_size = min(max_lens['answer'], model_params.max_answer_num)
```

- 변경후
  - `Model_Selector`의 파라미터는 `model_params`를 이용하여 미리 고정한 뒤, input과 모델 생성을 분리합니다.

```
## File: predict.py
class AnswerSelectionModel():
  def __init__(self):
    data = VTTExample('prediction', model_params, dicts, False)
    # generate model
    with tf.variable_scope('model') as scope:
      model = Model_Selector(data.emb_mat_token, data.emb_mat_glove,
                             len(data.dicts['token']), len(data.dicts['char']),
                             data.max_token_size, data.max_ans_size, ...)
```

```
## File: dataset.py
class VTTExample(object):
  def __init__(self, data_type, model_params, dicts, is_binary):
    self.max_token_size = model_params.max_token_size
    self.max_ans_size = model_params.max_answer_num
```

R3. predict 함수에서 불필요하게 data loader worker를 사용하지 않습니다.

PyTorch의 경우 data loader에서 worker를 사용하면 subprocess를 생성하게 됩니다. Data loader의 본래 목적은 학습 중에 데이터를 읽어오는 subprocess를 두어 data loading과 학습을 병렬로 수행하는 것입니다. 추론 시 data loader worker를 사용하면 병렬화로 인한 성능 이득은 없고 subprocess를 돌리는 overhead만 있습니다. 따라서 추론 모델에서는 worker를 사용하지 않도록 합니다.

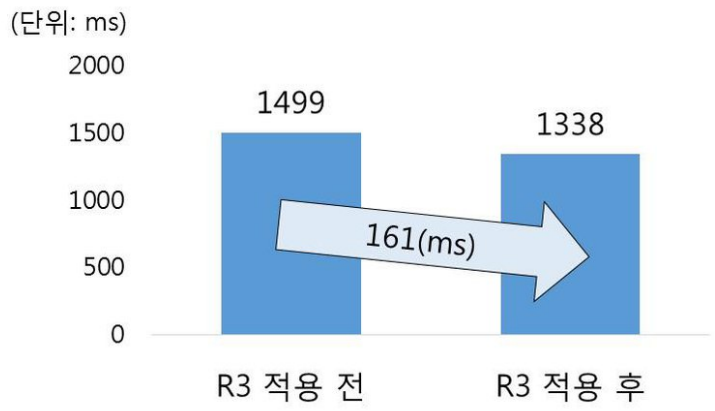
- data loader의 worker를 사용하지 않도록 변경한 예시입니다:
  - 변경전

```
data_loader = FriendsBertDataLoader(samples, num_samples, num_workers=1, ...)
```

  - 변경후

```
data_loader = FriendsBertDataLoader(samples, num_samples, num_workers=0, ...)
```

- 아래는 R3 적용에 따른 Pipeline 수행 시간 변화입니다 (단위: ms).



R4. 추론에 불필요한 input은 predict 함수에서 제거합니다.

학습 모델의 forward 함수를 그대로 추론 모델에 사용하는 경우, 일부 input이 추론 연산에 사용되지 않는 경우가 있습니다. 사용하지 않는 input은 predict 함수에서 제거합니다.

- 아래는 `forward` 함수의 일부 input이 output 계산에 사용되지 않는 예시입니다:
  - 변경전
    - `forward` 함수는 negative sample(`input3`, `input4`)을 input으로 받고 있습니다.
    - `y1`만 추론 결과에 사용되며, `y2`는 사용되지 않습니다. 붉은 영역은 `y2`를 연산하는데 사용된 연산입니다.

```
## File: infer.py
def main():
...
output = model(input[0],input[1],input[2],input[3])
L.append(output[0][0][1].item())
...
```

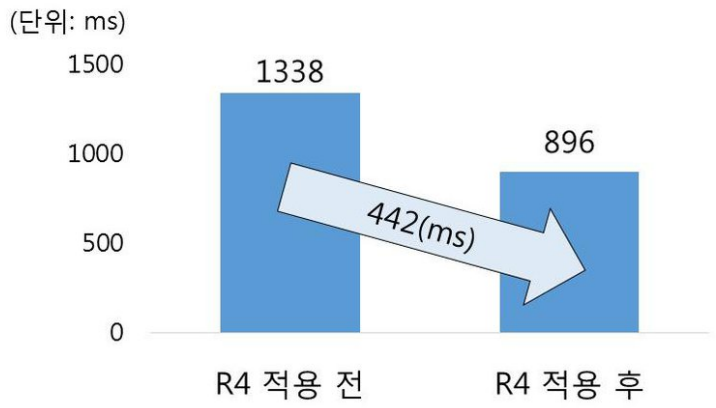
```
## File: model.py
def forward(self, input1,input2,input3,input4):
  x1,_ = self.bert1(input1[0],input1[1])
  x2,_ = self.bert2(input2[0],input2[1])
  x3,_ = self.bert1(input3[0],input3[1])
  x4,_ = self.bert2(input4[0],input4[1])
  y1 = torch.cat((x1[:,0],x2[:,0]),1)
  y2 = torch.cat((x3[:,0],x4[:,0]),1)
  y1 = self.fc(y1)
  y2 = self.fc(y2)
  y1 = F.softmax(y1,dim=1)
  y2 = F.softmax(y2,dim=1)
  return (y1,y2)
```

- 변경후
  - `forward` 함수에서 `y2`를 계산하는 항목을 지운 코드입니다.

```
## File: infer.py
def predict(input):
  ...
  output = model(input[0],input[1])
  L.append(output[0][1].item())
  ...
```

```
## File: model.py
def forward(self, input1,input2):
  x1,_ = self.bert1(input1[0],input1[1])
  x2,_ = self.bert2(input2[0],input2[1])
  y1 = torch.cat((x1[:,0],x2[:,0]),1)
  y1 = self.fc(y1)
  y1 = F.softmax(y1,dim=1)
  return y1
```

- 아래는 R4 적용에 따른 Pipeline 수행 시간 변화입니다 (단위: ms).



R5. predict 함수에서 최대한 효율적인 연산을 사용합니다.

- 아래는 비효율적인 연산과 이를 개선한 예시입니다:
  - (A): dataset에서 사용할 데이터를 csv 파일에 쓰고, 작성한 csv 파일을 다시 읽는 경우
    - 변경전

```
# Write test.csv file
writefile(dataset,'dataset/test.csv', episode, question)
# Read test.csv file
samples = pd.read_csv('dataset/test.csv')
# Read test.csv file in "FriendsBertDataLoader"
data_loader = module_data.FriendsBertDataLoader('dataset/test.csv', ...)
```

    - 변경후: csv 파일로 write/read하지 않고, 바로 data를 읽도록 변경

```
# Hold data in 'samples'
# 'generate_samples' function is added to generate data frame 'samples'
samples = generate_samples(dataset, episode, question)
data_loader = module_data.FriendsBertDataLoader(samples, ...)
```

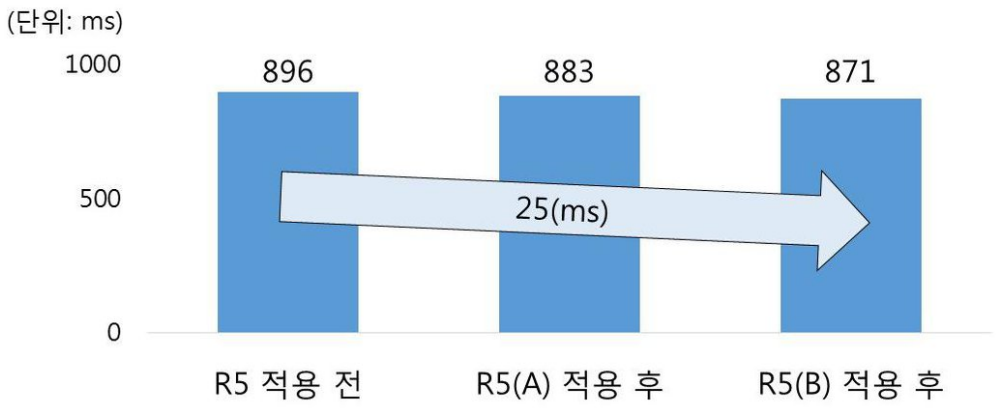
- (B): `rank` 연산을 이용하여 list의 최소값과 index를 구하는 경우
  - 변경전

```
# Convert list 'L' to series 'obj'
obj = pd.Series(L)
# List 'D' holds rank of each element in 'obj'
ranking = obj.rank(method='max').astype(int)
for idx in range(len(ranking)):
    D[samples['shot_id'].values[idx]]=str(ranking[idx])
# Find rank == "1" in list 'D'
for s_id, rank in D.items():
    if rank == str(1):
        scene = s_id
        break
```

  - 변경후: list의 `index`와 `min` 함수 이용

```
# Find index of minimum element in list 'L'
min_idx = L.index(min(L))
# Find scene id from dictionary 'samples' with index found above
scene = samples['shot_id'].values[min_idx]
```

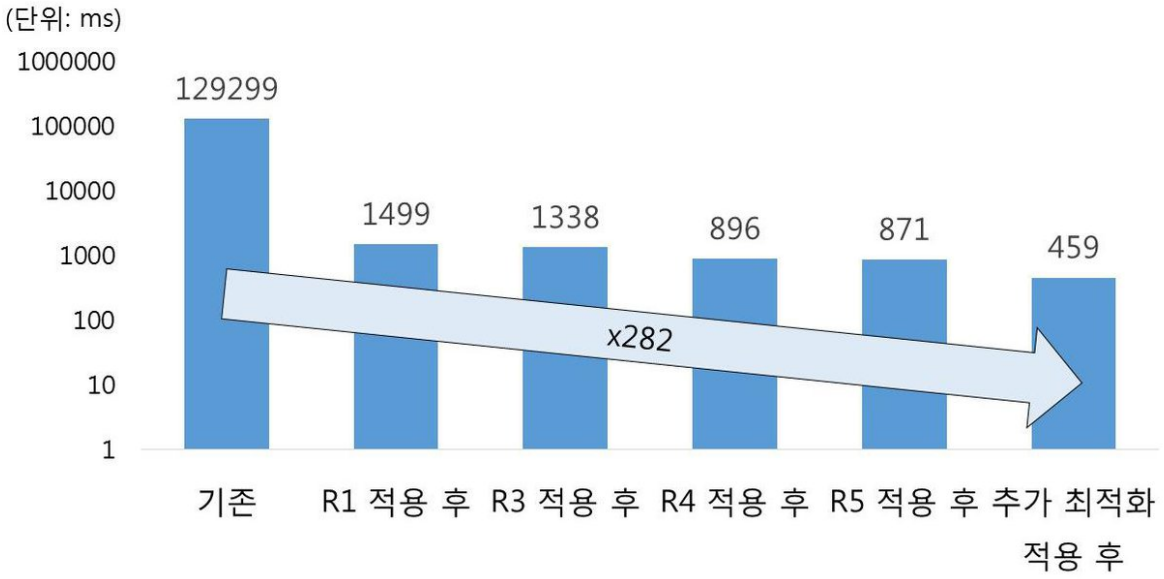
- 아래는 (A)와 (B)를 적용한 Pipeline 수행 시간입니다 (단위: ms).



4. Recommendation 적용에 따른 Pipeline 수행 시간 변화

아래는 각 Recommendation 적용에 따른 전체 Pipeline 수행 시간 로그 스케일 그래프입니다 (단위: ms).

- R5 적용 후, 추가로 프로세스 기반 병렬화와 input batching을 적용한 최적화 결과가 포함되어 있습니다. 프로세스 기반 병렬화를 통해 서로 dependency가 없는 모델을 같이 처리할 수 있습니다. (e.g. 모델 A, C, D)
- 최종적으로 원래 VTT pipeline 구현 대비 282배의 성능 향상이 있습니다.


(단위: ms)
1000000
100000
10000
1000
100
10
1
129299
1499
1338
896
871
459
x282
기존
R1 적용 후
R3 적용 후
R4 적용 후
R5 적용 후
추가 최적화 적용 후