exam 개인학습

Lec 3.1

AI summary Try on this page

Type something...

01) Image Captioning with Vanilla RNNS

<START> a clock surrounded by trees with no leaves <END>

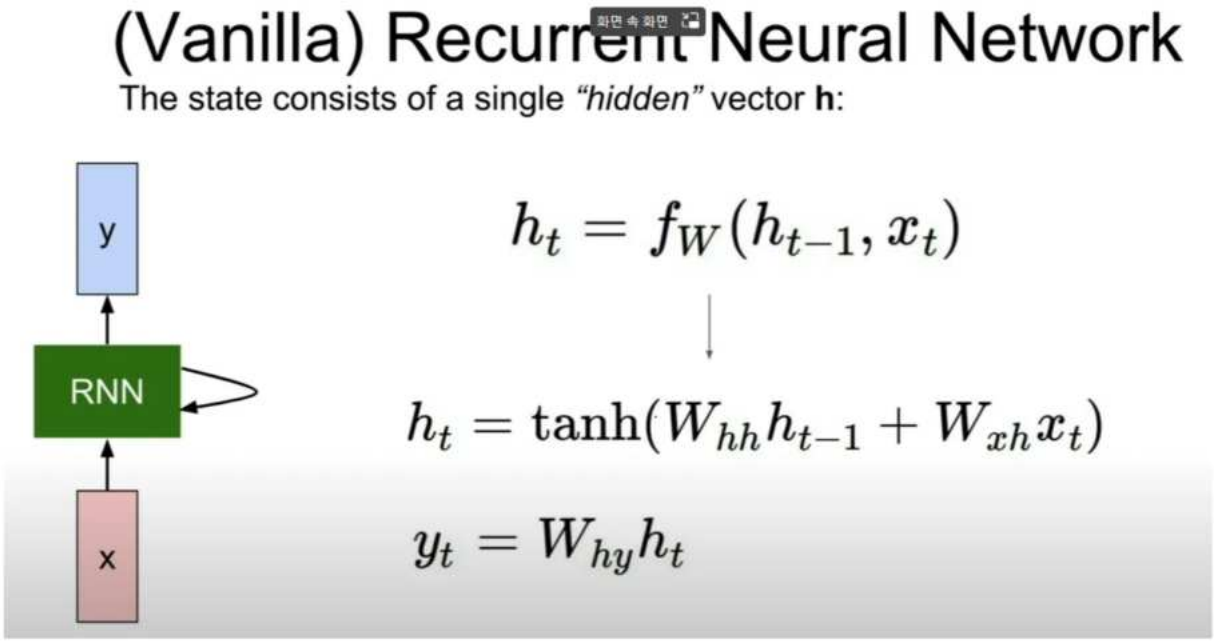
- 지금부터 이미지를 보고 문장을 생성하는 Image Captioning을 할 것이다.
- <START>, <END> 토큰들이 문장의 시작과 끝을 담당한다.
- 한 이미지당 5개의 설명을 가지고 있다.

```
[8]  for x in range(5):
       print(data['train_image_idxs'][x])
       print(data['train_captions'][x])
       print(decode_captions(data['train_captions'][x], data['idx_to_word']))
```

```
53314
[  1   4 142 510  10 667 415 277  58   2   0   0   0   0   0   0]
<START> a very clean and well decorated empty bathroom <END>
21548
[  1   4   3 172   6   4  62  10 317   6 114 612   2   0   0   0   0]
<START> a <UNK> view of a kitchen and all of its appliances <END>
53314
[  1   4  60  10  22  58   9   3   3 137   3   2   0   0   0   0   0]
<START> a blue and white bathroom with <UNK> <UNK> wall <UNK> <END>
21548
[  1   4   3 162   6   4  62  10 462  44   2   0   0   0   0   0   0]
<START> a <UNK> photo of a kitchen and dining room <END>
43077
[  1   4   3 149  59 242   7  25  97   4  48 129   2   0   0   0   0]
<START> a <UNK> stop sign across the street from a red car <END>
```

- 이미지당 인덱스가 존재하고 caption을 decode해서 영어 문장을 얻을 수 있다.
- 1 = `<START>` , 2 = `<END>` , 3 = `<UNK>` 로 할당되어 있고, 0은 할당되어 있는 값이 없다는 것을 확인할 수 있다.
- 최대 길이가 17인 것을 확인할 수 있다.

1_rnn_step_forward(x, prev_h, Wx, Wh, b):



$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$y_t = W_{hy}h_t$$

- 이전 hidden state를 받아 `affine_foward` 를 통과한 값과 Input이 `affine_foward` 를 통과한 값을 더한 후 `tanh` 을 씌워준다.
- output은 `FC-layer` 를 연결해서 출력한다.

```
next_h, cache = None, None
##############################################################################
# TODO: Implement a single forward step for the vanilla RNN. Store the next  #
# hidden state and any values you need for the backward pass in the next_h   #
# and cache variables respectively.                                          #
##############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

out1 = prev_h.dot(Wh) # (N, H)
out2 = x.dot(Wx)  # (N, H)
out = out1 + out2 + b
next_h = np.tanh(out)
cache = (x, prev_h, Wx, Wh, next_h)
```

- bias를 잊지말고 꼭 더해주자.
- `cache` 에는 x, prev_h, Wx, Wh, next_h를 저장해주었다.

2_rnn_step_backward(dnext_h, cache):

5. tanh 함수 미분

tanh함수의 미분은 다음의 분수 미분 공식을 이용하여 계산할 수 있다.

$$y = \left\{\frac{f(x)}{g(x)}\right\}' = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$$

$\frac{\partial e^x}{\partial x} = e^x$, $\frac{\partial e^x}{\partial x} = e^x$ 이것을 이용하면 다음과 같이 $tanh$ 함수를 미분할 수 있다.

$$\frac{\partial \tanh(x)}{\partial x} = \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2}$$

$$= 1 - \frac{(e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2}$$

$$= 1 - \left\{\frac{(e^x - e^{-x})}{(e^x + e^{-x})}\right\}^2$$

$$= 1 - \tanh(x)^2$$

$$= 1 - y^2$$

$$tanh' = 1 - (tanh)^2$$

```
x, prev_h, Wx, Wh, next_h = cache
dxhout = dnext_h * (1 - np.square(next_h)) # (4, 6)
dx = np.dot(dxhout, Wx.T)
dprev_h = np.dot(dxhout, Wh.T)
dWx = np.dot(x.T, dxhout)
dWh = np.dot(prev_h.T, dxhout)
db = np.sum(dxhout, axis = 0) # (6, )
```

- 사이즈를 맞춰주기 위해 전치행렬을 이용하여 사이즈를 맞춰주었다.
- bias는 열들의 합을 구해주기 위해 `axis = 0` 을 사용하였다.
- one_step을 구현했으니 이를 반복하여 rnn 전체 forward/backward를 구해보자.

행렬미분법칙 1: 선형 모형

선형 모형을 미분하면 그레디언트 벡터는 가중치 벡터이다.

$$f(x) = w^T x \tag{4.4.25}$$

$$\nabla f = \frac{\partial w^T x}{\partial x} = \frac{\partial x^T w}{\partial x} = w \tag{4.4.26}$$

(증명)

$$\frac{\partial(w^Tx)}{\partial x} = \begin{bmatrix} \frac{\partial(w^Tx)}{\partial x_1} \\ \frac{\partial(w^Tx)}{\partial x_2} \\ \vdots \\ \frac{\partial(w^Tx)}{\partial x_N} \end{bmatrix} = \begin{bmatrix} \frac{\partial(w_1x_1 + \cancel{w_2x_2 + \cdots + w_Nx_N})}{\partial x_1} \\ \frac{\partial(\cancel{w_1x_1} + w_2x_2 + \cancel{\cdots + w_Nx_N})}{\partial x_2} \\ \vdots \\ \frac{\partial(\cancel{w_1x_1 + w_2x_2 + \cdots} + w_Nx_N)}{\partial x_N} \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = w$$

$$\frac{\partial(w^Tx)}{\partial x} = w$$

3_rnn_forward(x, h0, Wx, Wh, b):

```
h = np.zeros((x.shape[0], x.shape[1], Wh.shape[0])) # (N, T, H)
cache = [0 for _ in range(x.shape[1])]
prev_h = h0
for t in range(x.shape[1]):
  next_h, cache_t = rnn_step_forward(x[:,t,:], prev_h, Wx, Wh, b)
  h[:,t,:] = next_h
  cache[t] = cache_t
  prev_h = next_h
```

- x의 모양이 `(N, T, D)` 이고 t개의 vector 가져서 t를 기준으로 반복문을 사용한다.
- 3차원과 2차원은 broadcast가 안되기 때문에 부분으로 잘라서 `rnn_step_forward` 에 넣는다.
- 그 후 h와 cache에 각각 저장해준다.

4_rnn_backward(dh, cache):

```
N, T, H = dh.shape
D = cache[0][0].shape[1] # x의 dim = 1
dx = np.zeros((N, T, D))
dh0 = np.zeros((N, H))
dWx = np.zeros((D, H))
dWh = np.zeros((H, H))
db = np.zeros((H, ))
dprev_h = np.zeros((N, H))

for t in range(T - 1, -1, -1): # T - 1 - 0
  cache_t = cache[t]
  dx_t, dprev_h_t, dWx_t, dWh_t, db_t = rnn_step_backward(dh[:,t,:]+dprev_h, cache_t)
  dx[:,t,:] = dx_t
  dWx  += dWx_t
  dWh += dWh_t
  db += db_t
  dprev_h = dprev_h_t
dh0 = dprev_h
```

- 각각 사이즈별로 초기화를 해주고 x만 concate시켜준다.
- **나머지는 gradient를 모두 합한다.**
- h gradient를 더해줘서 완전한 gradient를 만든다. (정확히 이해가지는 않는다.)

5_word_embedding_forward(x, W)

- 워드 임베딩(Word Embedding)은 단어를 벡터로 표현하는 것을 말한다. 즉 밀집 표현으로 변환하는 방법이다.

2. 밀집 표현(Dense Representation)

이러한 희소 표현과 반대되는 표현이 있으니, 이를 밀집 표현(dense representation)이라고 합니다. 밀집 표현은 벡터의 차원을 단어 집합의 크기로 상정하지 않습니다. 사용자가 설정한 값으로 모든 단어의 벡터 표현의 차원을 맞춥니다. 또한, 이 과정에서 더 이상 0과 1만 가진 값이 아니라 실수값을 가지게 됩니다. 다시 희소 표현의 예를 가져와봅시다.

Ex) 강아지 = [0 0 0 0 1 0 0 0 0 0 0 0 ... 중략 ... 0] # 이 때 1 뒤의 0의 수는 9995개. 차원은 10,000

예를 들어 10,000개의 단어가 있을 때 강아지란 단어를 표현하기 위해서는 위와 같은 표현을 사용했습니다. 하지만 밀집 표현을 사용하고, 사용자가 밀집 표현의 차원을 128로 설정한다면, 모든 단어의 벡터 표현의 차원은 128로 바뀌면서 모든 값이 실수가 됩니다.

Ex) 강아지 = [0.2 1.8 1.1 -2.1 1.1 2.8 ... 중략 ...] # 이 벡터의 차원은 128

이 경우 벡터의 차원이 조밀해졌다고 하여 밀집 벡터(dense vector)라고 합니다.

- `one-hot encoding` 을 사용하게 되면 벡터의 차원이 너무 커지게 되기 때문에 벡터의 차원을 줄이고 값을 실수로 바꾼다.

```
out, cache = None, None
##############################################################################
# TODO: Implement the forward pass for word embeddings.                      #
#                                                                            #
# HINT: This can be done in one line using NumPy's array indexing.           #
##############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
out = W[x]
cache = (x, W)
```

- x: `(N, T)` , W: `(V, D)` 로 이루어져 있다.
- x는 N개의 데이터, T 단어 caption으로 이루어져 있고 ,W는 vocabulary 종류, RNN에 입력할 dim으로 이루어져 있다.
- x각각의 원소가 idx이기 때문에 W에 집어넣어서 word_embedding으로 만들 수 있다. (생각하기 어려웠던 문제였다.)

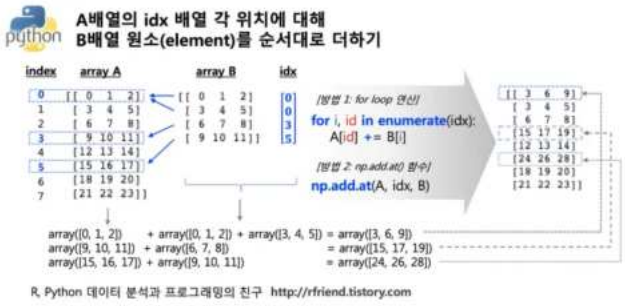
```
weight_matrix = np.array([
    [0.1, 0.2, 0.3],  # 'I'
    [0.4, 0.5, 0.6],  # 'love'
    [0.7, 0.8, 0.9],  # 'learning'
    [1.0, 1.1, 1.2],  # 'NLP'
    [1.3, 1.4, 1.5]   # 'ChatGPT'
])

# Define the input matrix (N, T)
input_matrix = np.array([
    [0, 2, 1, 3],  # ['I', 'learning', 'love', 'NLP']
    [3, 1, 4, 0]   # ['NLP', 'love', 'ChatGPT', 'I']
])

# Generate the output matrix (N, T, D)
output_matrix = weight_matrix[input_matrix]
```

- 행의 개수가 단어의 개수이고 각각의 행이 단어를 의미하기 때문에 `W[x]` 를 통해서 저장해줄 수 있었다.

6_word_embedding_backward(dout, cache):



- `np.add.at` 을 이용해서 해당 인덱스 위치에서 덧셈을 진행할 수 있다.

```
x, W = cache
dW = np.zeros_like(W)
np.add.at(dW, x, dout)
```

- dW에 해당하는 인덱스에 dout을 더해주는 방식으로 gradient를 구할 수 있었다.

```python
def temporal_affine_forward(x, w, b):
    """Forward pass for a temporal affine layer.

    The input is a set of D-dimensional
    vectors arranged into a minibatch of N timeseries, each of length T. We use
    an affine function to transform each of those vectors into a new vector of
    dimension M.

    Inputs:
    - x: Input data of shape (N, T, D)
    - w: Weights of shape (D, M)
    - b: Biases of shape (M,)

    Returns a tuple of:
    - out: Output data of shape (N, T, M)
    - cache: Values needed for the backward pass
    """
    N, T, D = x.shape
    M = b.shape[0]
    out = x.reshape(N * T, D).dot(w).reshape(N, T, M) + b
    cache = x, w, b, out
    return out, cache
```

- temporal_affine_layer 가 하는 일은 RNN을 통해 나온 output을 vocabulary size에 맞게 벡터화 시켜주는 것이다.
- FC layer들 통과하기 위해 `x.reshape(N * T, D)` 를 사용해서 w와 연산시켜준 후 새로운 M vector로 만들어 준다.
- 아까 word_embedding함수는 **x raw data -> x_ vocab -> RNN dim** 에 맞추어 주었다면, temporal affine layer함수는 반대로 **RNN dim -> vocab dim으로 벡터화 시켜준다.**
- 위에서는 M이 vocabulary 종류와 같을 것이고, 각 (N, T)에는 각 M vocab 종류마다의 점수가 저장될 것이다.

```python
        self.cell_type = cell_type
        self.dtype = dtype
        self.word_to_idx = word_to_idx
        self.idx_to_word = {i: w for w, i in word_to_idx.items()}
        self.params = {}

        vocab_size = len(word_to_idx)

        self._null = word_to_idx["<NULL>"]
        self._start = word_to_idx.get("<START>", None)
        self._end = word_to_idx.get("<END>", None)

        # Initialize word vectors
        self.params["W_embed"] = np.random.randn(vocab_size, wordvec_dim)
        self.params["W_embed"] /= 100

        # Initialize CNN -> hidden state projection parameters
        self.params["W_proj"] = np.random.randn(input_dim, hidden_dim)
        self.params["W_proj"] /= np.sqrt(input_dim)
        self.params["b_proj"] = np.zeros(hidden_dim)

        # Initialize parameters for the RNN
        dim_mul = {"lstm": 4, "rnn": 1}[cell_type]
        self.params["Wx"] = np.random.randn(wordvec_dim, dim_mul * hidden_dim)
        self.params["Wx"] /= np.sqrt(wordvec_dim)
        self.params["Wh"] = np.random.randn(hidden_dim, dim_mul * hidden_dim)
        self.params["Wh"] /= np.sqrt(hidden_dim)
        self.params["b"] = np.zeros(dim_mul * hidden_dim)

        # Initialize output to vocab weights
        self.params["W_vocab"] = np.random.randn(hidden_dim, vocab_size)
        self.params["W_vocab"] /= np.sqrt(hidden_dim)
        self.params["b_vocab"] = np.zeros(vocab_size)

        # Cast parameters to correct dtype
        for k, v in self.params.items():
            self.params[k] = v.astype(self.dtype)
```

- word vector 초기화 부분
- CNN 초기화 부분 → hidden을 전달
- lstm / rnn 초기화 하는 부분
- output 초기화 하는 부분

7_loss(self, features, captions):

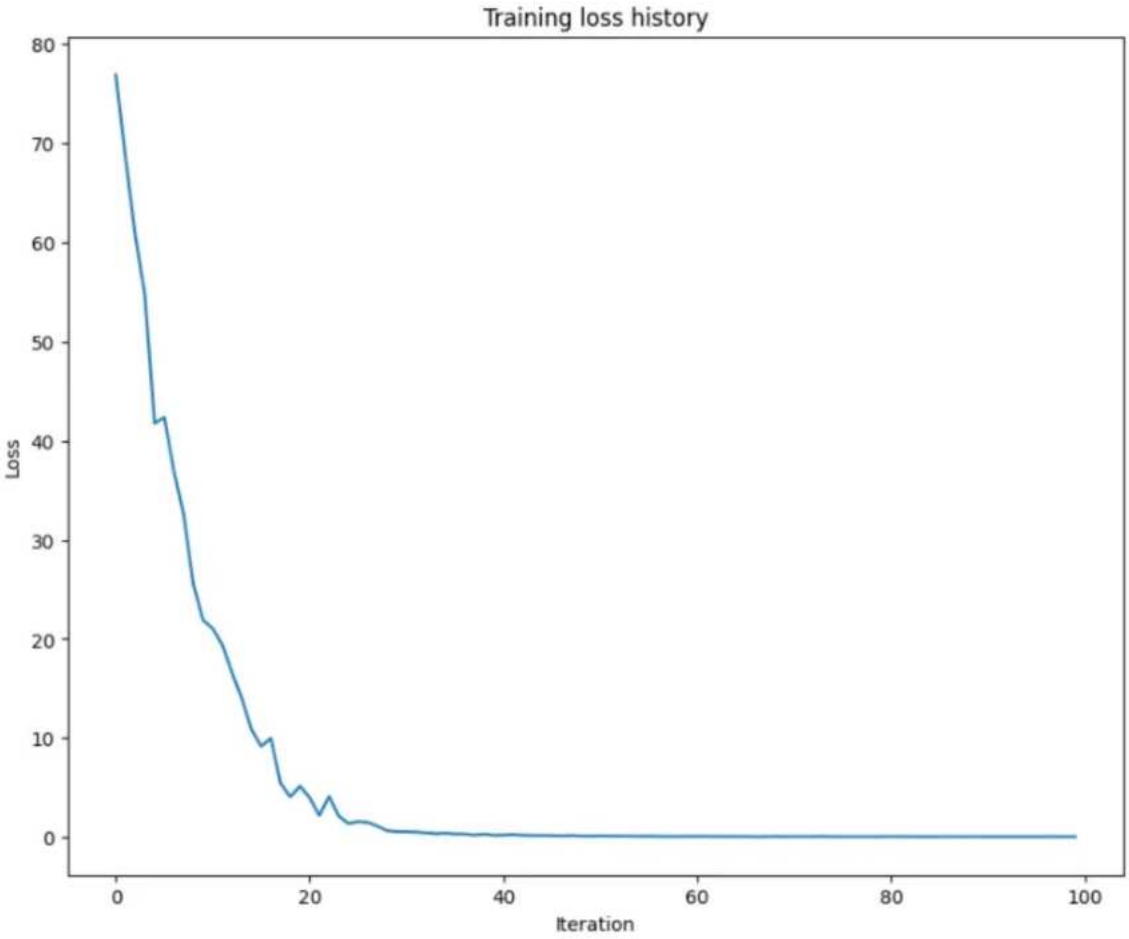
- 앞서 초기화 한 parameter들을 바탕으로 loss를 구현해보자.
- loss는 이전 모델들을 참고하여 약간 변형만 하였다.

1. affine transformation: image features 넣기: hidden state 생성
2. word embedding layer: captions_in → vector로 바꿔주기
3. vanilla RNN 사용하기: vector, hidden state 넣어주기
4. temporal affine transformation: score 계산하기
5. temporal softmax 사용해서 loss 구하기

```
# forward part
h0, cache1 = affine_forward(features, W_proj, b_proj)
cout, cache2 = word_embedding_forward(captions_in, W_embed)
h, cache3 = rnn_forward(cout, h0, Wx, Wh, b)
scores, cache4 = temporal_affine_forward(h, W_vocab, b_vocab)
loss, dx = temporal_softmax_loss(scores, captions_out,mask)

# backward pass
dx4, dw4, db4 = temporal_affine_backward(dx, cache4)
grads['W_vocab'] = dw4
grads['b_vocab'] = db4
dx3, dh0, dWx3, dWh3, db3 = rnn_backward(dx4, cache3)
grads['Wx'] = dWx3
grads['Wh'] = dWh3
grads['b'] = db3
dW2 = word_embedding_backward(dx3, cache2)
grads['W_embed'] = dW2
dx1, dW1, db1 = affine_backward(dh0, cache1)
grads['W_proj'] = dW1
grads['b_proj'] = db1
```

- forward을 부분을 작성한 후 반대로 돌아가서 backward를 작성하였다.
- dx, dh0값 혼동에 주의하자.



- loss값이 급격하게 떨어지는 것을 확인할 수 있다.

8_sample(self, features, max_length=30):

RNN Sampling at Test Time

Unlike classification models, image captioning models behave very differently at training time vs. at test time. At training time, we have access to the ground-truth caption, so we feed ground-truth words as input to the RNN at each timestep. At test time, we sample from the distribution over the vocabulary at each timestep and feed the sample as input to the RNN at the next timestep.

In the file `cs231n/classifiers/rnn.py`, implement the `sample` method for test-time sampling. After doing so, run the following to sample from your overfitted model on both training and validation data. The samples on training data should be very good. The samples on validation data, however, probably won't make sense.

- sample 함수는 test time 때 caption을 sampling을 하는 방식을 구현하는 것이다.

- 각 time step에서 다음 hidden state를 계산하고, hidden state를 이용해서 모든 단어들의 score를 얻는다.
- 그 중에서 가장 큰 score를 가진 단어를 다음 단어로 출력하는 것이다. 이렇게 구현을 하긴 하지만, test time 때의 성능이 그닥 좋진 않다.

```
# forward part
h,cache1=affine_forward(features,W_proj,b_proj) # initializing hidden state
prev = self._start*np.ones((N,),dtype=np.int32) # feed start token to the rnn.
for i in range(max_length):
  prev_embed,cache2 = word_embedding_forward(prev, W_embed) #embed the input.
  h, cache3 = rnn_step_forward(prev_embed, h, Wx, Wh, b)
  scores,cache4 = affine_forward(h,W_vocab ,b_vocab) #get the scores-> use the max index to predict the next word.
  prev = np.argmax(scores,axis=1) # update prev to the highest score index.
  captions[:,i] = prev
```

- 각 단어마다 score의 최대값을 caption에 저장해주었다.

1. Inline Question 1

Inline Question 1

In our current image captioning setup, our RNN language model produces a word at every timestep as its output. However, an alternate way to pose the problem is to train the network to operate over *characters* (e.g. 'a', 'b', etc.) as opposed to words, so that at it every timestep, it receives the previous character as input and tries to predict the next character in the sequence. For example, the network might generate a caption like

'A', ' ', 'c', 'a', 't', ' ', 'o', 'n', ' ', 'a', ' ', 'b', 'e', 'd'

Can you describe one advantage of an image-captioning model that uses a character-level RNN? Can you also describe one disadvantage? HINT: there are several valid answers, but it might be useful to compare the parameter space of word-level and character-level models.

Your Answer:

- 단어 단위가 아닌 글자 단위로 caption을 처리하면 어떻게 달라질 까?
- 단어 단위의 인덱싱을 하면 parameter의 수가 매우 커져서 모델의 속도가 느리게 될 것이다.
- 하지만 이전 단어를 문맥으로 삼아 추론할 수 있기 때문에 정확도는 더 높을 것이다.