HW3

2022019734 김민서

1. Code Implementation

1) RNN

먼저 RNN의 step_forward와 step_backward는 구현되어 있는 affine_forward와 affine_backward를 사용해서 구현했다. 이때, backward는 chain rule을 활용해 out에 대한 gradient를 구한 뒤 이를 활용하는 식으로 접근했다.

```
X_concat = np.concatenate((prev_h, x), axis=1)
W_concat = np.concatenate((Wh, Wx), axis=0)
next_h, cache = affine_forward(X_concat, W_concat, b)
next_h = np.tanh(next_h)
```

```
X_concat, W_concat, b = cache
H = b.shape[0]
out, _ = affine_forward(X_concat, W_concat, b)
dout = dnext_h * (1 - np.tanh(out) ** 2)
dX_concat, dW_concat, db = affine_backward(dout, cache)
dprev_h, dx = dX_concat[:, :H], dX_concat[:, H:]
dWh, dWx = dW_concat[:H], dW_concat[H:]
```

```
next_h error:  6.292421426471037e-09
```

```
dx error:  2.886660575545105e-10
dprev_h error:  2.7376198521247043e-10
dWx error:  1.0572704842803103e-09
dWh error:  1.208567265765649e-09
db error:  1.708752322503098e-11
```

Forward와 backward는 step_forward와 step_backward를 반복해서 적용했으며, backward 과정에서 upstream에서 온 gradient와 현재 gradient를 합쳐서 backward를 진행해야 함에 유의해야 했다.

```
N, T, D = x.shape
prev_h = h0
hList = []
cache = []
for i in range(T):
    x_curr = x[:,i,:]
    prev_h, cache_curr = rnn_step_forward(x_curr, prev_h, Wx, Wh, b)
    hList.append(prev_h)
    cache.append(cache_curr)
h = np.stack(hList, axis=1)
```

```
N, T, H = dh.shape
dnext_h = dh[:,-1,:]
D = cache[0][0].shape[1] - H
dWx = np.zeros((D, H))
dWh = np.zeros((H, H))
db = np.zeros(H)
dxList = []
for i in reversed(range(T)):
    ingrad = dnext_h + dh[:,i,:] if i != T - 1 else dnext_h
    dx_curr, dnext_h, dWx_curr, dWh_curr, db_curr = rnn_step_backward(ingrad, ca
    dxList.append(dx_curr)
    dWx += dWx_curr
    dWh += dWh_curr
    db += db_curr
dh0 = dnext_h
dx = np.stack(dxList[::-1], axis=1)
```

```
h error:  7.728466180186066e-08
```

```
dx error:  1.5378207243060017e-09
dh0 error:  3.3807584058812853e-09
dWx error:  7.240201246394462e-09
dWh error:  1.2948851548797392e-07
db error:  2.2649434379099313e-10
```

Word embedding은 각각 인덱싱과 np.add.at 메서드를 활용해서 구현할 수 있었다.

```
out = W[x]
cache = (x, W)
```

```
x, W = cache
V, D = W.shape
dW = np.zeros((V, D))
np.add.at(dW, x, dout)
```

```
dW error:  3.2774595693100364e-12
```

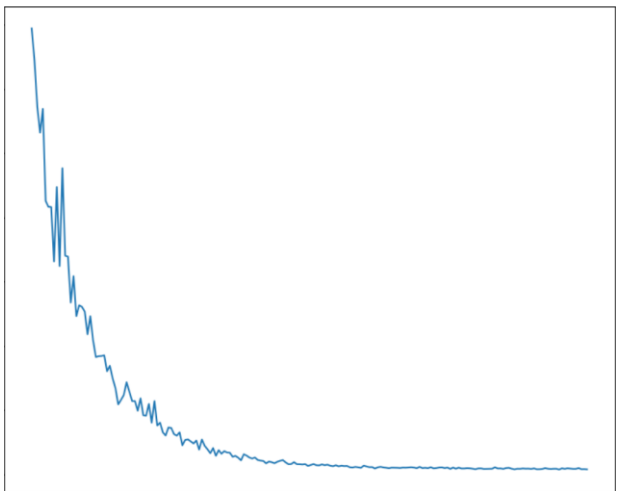
```
dx error:  2.9215945034030545e-10
dw error:  1.5772088618663602e-10
db error:  3.252200556967514e-11
```

위에서 만든 모듈을 이용해서 RNN의 loss를 구현하면 다음과 같다. 이미지 feature을 affine_forward를 통해 project하고, caption은 word embedding을 한다. 이후, 두 결과물을 바탕으로 forward를 진행한 후, 해당 결과를 vocab에 대한 점수들로 바꿔주기 위해 affine_forward를 한 뒤 softmax를 취해서 loss를 계산한다. Backward 단계에서는 이렇게 해서 얻어진 loss를 chain rule을 이용해 역으로 propagate하면 된다.

```
h0, cache_feat = affine_forward(features, W_proj, b_proj)
x, cache_embed = word_embedding_forward(captions_in, W_embed)
h, cache = rnn_forward(x, h0, Wx, Wh, b)
out_vocab, cache_vocab = temporal_affine_forward(h, W_vocab, b_vocab)
loss, dout = temporal_softmax_loss(out_vocab, captions_out, mask)
dh, grads["W_vocab"], grads["b_vocab"] = temporal_affine_backward(dout, cache_vocab)
dembed, dh0, grads["Wx"], grads["Wh"], grads["b"] = rnn_backward(dh, cache)
grads["W_embed"] = word_embedding_backward(dembed, cache_embed)
_, grads["W_proj"], grads["b_proj"] = affine_backward(dh0, cache_feat)
```

```
loss:  9.832355910027385
expected loss:  9.83235591003
difference:  2.6147972675971687e-12
```

위의 코드를 바탕으로 작은 데이터셋에 학습을 시키면 아래와 같은 loss graph를 얻을 수 있다.



inference를 위해서 sample 부분을 구현하면 다음과 같다. Loss 함수에서 forward 과정과 비슷한 방식으로 score을 얻어준 뒤, 가장 높은 점수를 가진 word를 현재 caption으로

예측하는 방식이다.

```
if self.cell_type == "rnn":
    h0, _ = affine_forward(features, W_proj, b_proj)
    prev_word = self._start * np.ones(N, dtype=np.int32)
    prev_h = h0
    for i in range(max_length):
        embed, _ = word_embedding_forward(prev_word, W_embed)
        curr_h, _ = rnn_step_forward(embed, prev_h, Wx, Wh, b)
        score, _ = affine_forward(curr_h, W_vocab, b_vocab)
        curr_word = np.argmax(score, axis=1)
        captions[:,i] = curr_word
        prev_word = curr_word
        prev_h = curr_h
```

해당 코드를 바탕으로 inference를 해주면 train set에 대해서 상당히 overfitting되어 ground truth 자체를 output으로 내놓은 것을 알 수 있다.

train
a boy sitting with <UNK> on with a donut in his hand <END>
GT:<START> a boy sitting with <UNK> on with a donut in his hand <END>

train
a man <UNK> with a bright colorful kite <END>
GT:<START> a man <UNK> with a bright colorful kite <END>

반면, validation set에 대해서는 아예 맞지 않는 이야기를 하고 있음을 알 수 있다.

2) LSTM

LSTM의 step_forward와 step_backward는 rnn에서의 수식이 lstm에서의 수식으로 대체되었다는 것 외에는 큰 구조적인 변경점은 없다. dnext_c를 구할 때 short cut connection으로 인한 gradient와 원래 흐름으로 인한 gradient를 모두 고려함에 유의해야 했다.

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$

```
N, H = prev_h.shape
X_concat = np.concatenate((prev_h, x), axis=1)
W_concat = np.concatenate((Wh, Wx), axis=0)
a, _ = affine_forward(X_concat, W_concat, b)
i, f, o, g = sigmoid(a[:,:H]), sigmoid(a[:,H:2*H]), sigmoid(a[:,2*H:3*H]), np.tanh(a[:,3*H:])
next_c = f * prev_c + i * g
next_h = o * np.tanh(next_c)
cache = (i, f, o, g, prev_c, W_concat, X_concat, b)
```

```
i, f, o, g, prev_c, W_concat, X_concat, b = cache
next_c = f * prev_c + i * g
N, H = dnext_h.shape
do = dnext_h * np.tanh(next_c)
dnext_c = dnext_c + dnext_h * o * (1 - np.tanh(next_c) ** 2)
dprev_c = dnext_c * f
df = dnext_c * prev_c
di = dnext_c * g
dg = dnext_c * i
dai = di * i * (1 - i) # i = sigmoid(ai) / g = tanh(ag)
daf = df * f * (1 - f)
dao = do * o * (1 - o)
dag = dg * (1 - g ** 2)
da = np.concatenate((dai, daf, dao, dag), axis=1)
dX_concat, dW_concat, db = affine_backward(da, (X_concat, W_concat, b))
dprev_h, dx = dX_concat[:, :H], dX_concat[:, H:]
dWh, dWx = dW_concat[:H], dW_concat[H:]
```

```
dx error:  3.261398431623834e-09
dh error:  2.7327025072558407e-10
dc error:  1.5221723979041107e-10
dWx error:  1.6989913110863295e-09
dWh error:  2.7311398322845992e-08
db error:  2.5828274262391835e-10
next_h error:  5.7054131967097955e-09
next_c error:  5.8143123088804145e-09
```

lstm에서의 forward와 backward 역시 step_forward와 step_backward를 반복해주는 것을 통해 구현할 수 있다.

```
    N, T, D = x.shape
    _, H = h0.shape
    prev_h = h0
    prev_c = np.zeros((N, H))
    hList = []
    cache = []
    for i in range(T):
        x_curr = x[:,i,:]
        curr_h, curr_c, cache_curr = lstm_step_forward(x_curr, prev_h, prev_c, Wx, Wh, b)
        hList.append(curr_h)
        cache.append(cache_curr)
        prev_h = curr_h
        prev_c = curr_c
    h = np.stack(hList, axis=1)
```

```
N, T, H = dh.shape
dnext_h = dh[:,-1,:]
dnext_c = np.zeros((N, H))
D = cache[0][6].shape[1] - H
dWx = np.zeros((D, 4 * H))
dWh = np.zeros((H, 4 * H))
db = np.zeros(4 * H)
dxList = []
for i in reversed(range(T)):
    ingrad = dnext_h + dh[:,i,:] if i != T - 1 else dnext_h
    dx_curr, dnext_h, dnext_c, dWx_curr, dWh_curr, db_curr = lstm_step_backward(ingrad, dnext_c, cache[i])
    dxList.append(dx_curr)
    dWx += dWx_curr
    dWh += dWh_curr
    db += db_curr
dh0 = dnext_h
dx = np.stack(dxList[::-1], axis=1)
```

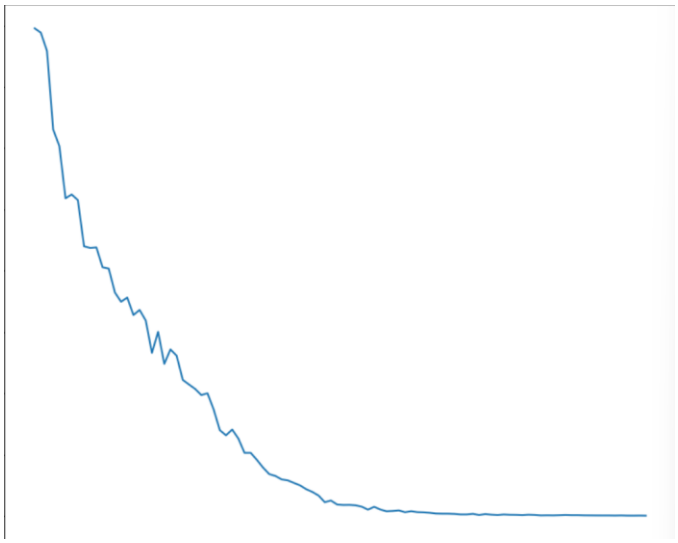
```
dx error:  4.282359190539485e-09
dh0 error:  1.8820763975745557e-08
dWx error:  1.4475881990556828e-09
dWh error:  6.258723210101113e-07
db error:  2.676008503610984e-10
h error:  8.610537442272635e-08
```

Loss를 구하는 과정 역시 rnn과 비슷하되 lstm_forward를 적용해주면 된다.

```
h0, cache_feat = affine_forward(features, W_proj, b_proj)
x, cache_embed = word_embedding_forward(captions_in, W_embed)
h, cache = lstm_forward(x, h0, Wx, Wh, b)
out_vocab, cache_vocab = temporal_affine_forward(h, W_vocab, b_vocab)
loss, dout = temporal_softmax_loss(out_vocab, captions_out, mask)
dh, grads["W_vocab"], grads["b_vocab"] = temporal_affine_backward(dout, cache_vocab)
dembed, dh0, grads["Wx"], grads["Wh"], grads["b"] = lstm_backward(dh, cache)
grads["W_embed"] = word_embedding_backward(dembed, cache_embed)
_, grads["W_proj"], grads["b_proj"] = affine_backward(dh0, cache_feat)
```

```
loss:  9.82445935443226
expected loss:  9.82445935443
difference:  2.261302256556519e-12
```

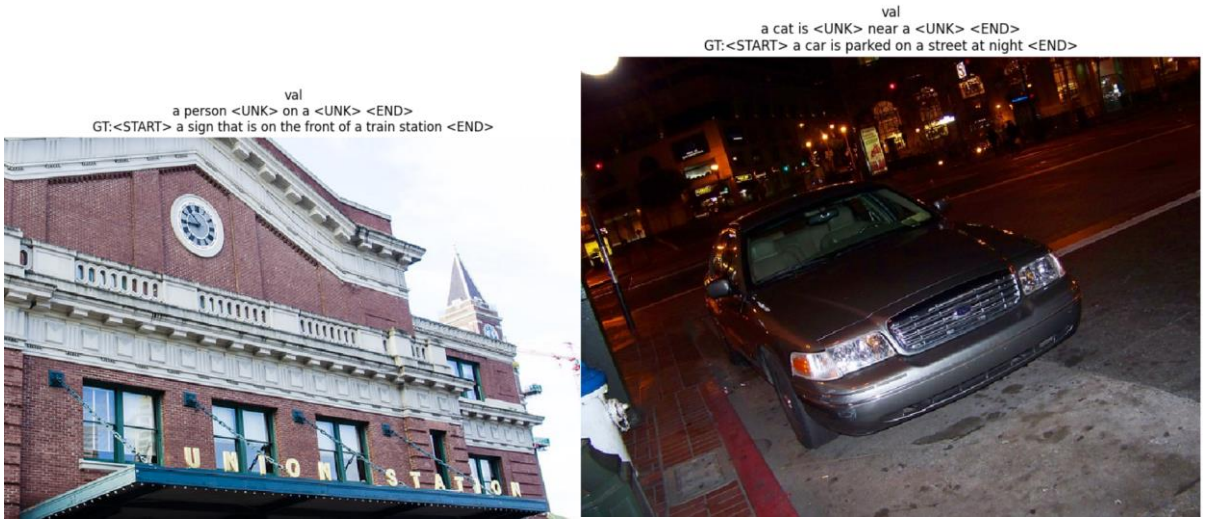
이때 loss curve는 아래와 같다.



Sampling 역시 lstm_step_forward를 적용해서 rnn과 비슷하게 진행하면 된다.

```
h0, _ = affine_forward(features, W_proj, b_proj)
prev_word = self._start * np.ones(N, dtype=np.int32)
prev_h = h0
prev_c = np.zeros_like(h0)
for i in range(max_length):
    embed, _ = word_embedding_forward(prev_word, W_embed)
    curr_h, curr_c, _ = lstm_step_forward(embed, prev_h, prev_c, Wx, Wh, b)
    score, _ = affine_forward(curr_h, W_vocab, b_vocab)
    curr_word = np.argmax(score, axis=1)
    captions[:,i] = curr_word
    prev_word = curr_word
    prev_h = curr_h
    prev_c = curr_c
```

작은 데이터셋에 대해서 학습을 진행했기에 마찬가지로 overfitting이 많이 되어, train에는 ground truth를, test에는 완전히 이상한 caption을 만들어내는 것을 알 수 있다.



3) Transformer

Multi head attetion의 forward의 경우, key, query, value를 뽑아낸 뒤, 이를 N, H, T, E//H 모양으로 만든 후, query와 key를 곱해서 attention score를 얻는다. 이때, mask가 필요한 경우, attention score를 논문에 나와 있듯이 -inf로 세팅해준다. 그 후 attention score에 softmax와 dropout를 거친 후, value와 곱하면 된다.

```
query = self.query(query).view(N, S, self.n_head, self.head_dim).transpose(1, 2)
key = self.key(key).view(N, T, self.n_head, self.head_dim).transpose(1, 2)
value = self.value(value).view(N, T, self.n_head, self.head_dim).transpose(1, 2)
attn_score = torch.matmul(query, key.transpose(2, 3)) / math.sqrt(self.head_dim)
if attn_mask is not None:
    attn_score = attn_score.masked_fill(attn_mask==0, -float("inf"))
attn_score = F.softmax(attn_score, dim=-1)
attn_score = self.attn_drop(attn_score)
output = torch.matmul(attn_score, value)
output = self.proj(output.transpose(1, 2).contiguous().view(N, S, E))
```

```
self_attn_output error:  0.0003775124598178026
masked_self_attn_output error:  0.0001526367643724865
attn_output error:  0.0003527921483788199
```

아래의 공식에 따라 positional encoding을 만든 뒤, 이를 원래 결과에 합쳐주면 된다.

$$\begin{cases} \sin\left(i \cdot 10000^{-\frac{j}{d}}\right) & \text{if j is even} \\ \cos\left(i \cdot 10000^{-\frac{(j-1)}{d}}\right) & \text{otherwise} \end{cases}$$

```
ivals = torch.arange(0, max_len).unsqueeze(1)
jvals = torch.arange(0, embed_dim, 2).unsqueeze(0)
pe[:,:,0::2] = torch.sin(ivals * 10000 ** -(jvals / embed_dim))
pe[:,:,1::2] = torch.cos(ivals * 10000 ** -(jvals / embed_dim))
```

```
output = x + self.pe[:,:x.shape[1]]
output = self.dropout(output)
```

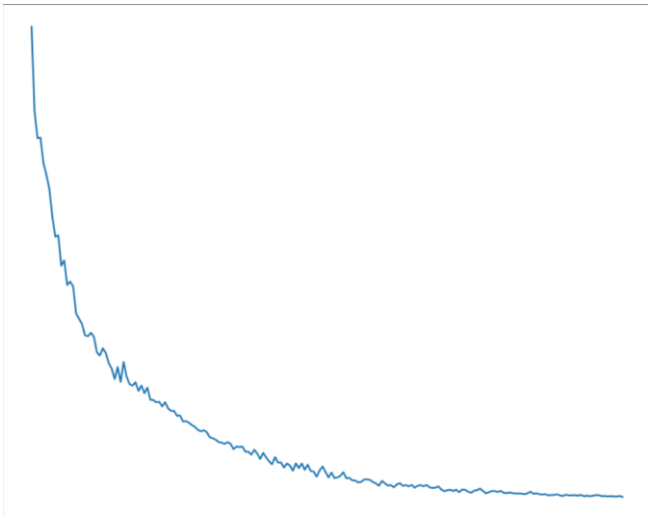
```
pe_output error:  0.00010421011374914356
```

Captioning transformer의 경우, image feature를 projection하고, caption을 embedding 한 뒤, 마스크와 함께 transformer decoder에 넣어주면 된다.

```
embed_img = self.visual_projection(features).unsqueeze(1)
embed_cap = self.positional_encoding(self.embedding(captions))
tgt_mask = torch.tril(torch.ones(T, T)).to(device)
output = self.transformer(embed_cap, embed_img, tgt_mask)
scores = self.output(output)
```

```
scores error:  5.056720613955726e-06
```

이때, loss curve는 아래와 같다.



훈련된 transformer를 가지고 inference를 해보면, 마찬가지로 train set에 overfitting되었음을 알 수 있다.

2. Implementation of CIDEr

CIDEr 구현의 경우 spec 문서에 나와있는 https://github.com/vrama91/cider를 참고해서 구현했다. tf-idf를 이용해서 코사인 유사도를 계산하고 있음을 알 수 있다.

```
for test, refs in zip(self.ctest, self.crefs):
    # compute vector for test captions
    vec, norm, length = counts2vec(test)
    # compute vector for ref captions
    score = np.array([0.0 for _ in range(self.n)])
    for ref in refs:
        vec_ref, norm_ref, length_ref = counts2vec(ref)
        score += sim(vec, vec_ref, norm, norm_ref, length, length_ref)
```

```
# measure consine similarity
val = np.array([0.0 for _ in range(self.n)])
for n in range(self.n):
    # ngram
    for (ngram,count) in vec_hyp[n].items():
        val[n] += vec_hyp[n][ngram] * vec_ref[n][ngram]

    if (norm_hyp[n] != 0) and (norm_ref[n] != 0):
        val[n] /= (norm_hyp[n]*norm_ref[n])

    assert(not math.isnan(val[n]))
return val
```

Rnn, lstm, transformer을 일부 train set에 대해서만 훈련시킨 뒤 CIDEr 점수를 구하면 다음과 같다.

	RNN	LSTM	Transformer
Train	7.2	9	12.9
Val	6.8	8.7	12.6

RNN의 경우 vanishing gradient 문제가 도드라지기에 가장 낮은 성능을 보인 것으로 추정되며, LSTM이 RNN보다 나은 모습을, Transformer가 제일 좋은 모습을 보여줬다. 따라서 4에서 NICE 데이터셋에 inference로 사용할 모델로는 transformer가 적절해보인다.

3. Visualize attention map

attention map은 transformer_drophead에 구현되어 있음

Multihead attention에 각 head의 attention score을 평균을 낸 뒤, 이를 반환해주었다.

```
attn_score = F.softmax(attn_score, dim=-1)
output = torch.matmul(attn_score, value)
output = self._drophead(output, mode)
output = self.proj(output.transpose(1, 2).contiguous().view(N, S, E))
```

```
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                              END OF YOUR CODE
################################################################################
return output, attn_score.mean(dim=1)
```

Transformer decoder layer에서 image-caption cross attention에 활용된 attention score를 반환해주었고, decoder에서 마지막 layer에서의 attention score을 최종적으로 visualization에 사용하였다.

```
# Attend to both the target sequence and the sequence from the last
# encoder layer.
tgt2, attns = self.multihead_attn(query=tgt, key=memory, value=memory, mode=mode)
tgt = tgt + self.dropout2(tgt2)
tgt = self.norm2(tgt)

# Pass
tgt2 = self.linear2(self.dropout(self.activation(self.linear1(tgt))))
tgt = tgt + self.dropout3(tgt2)
tgt = self.norm3(tgt)
return tgt, attns
```

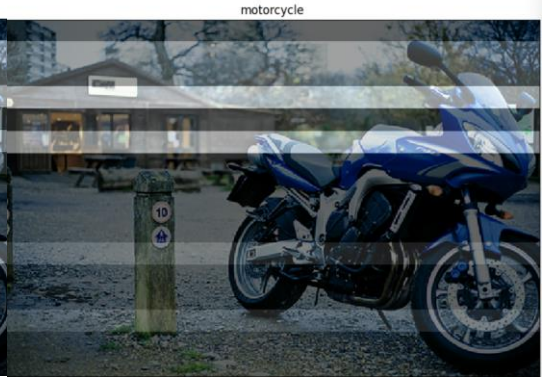
```
def forward(self, tgt, memory, mode, tgt_mask=None):
    output = tgt
    attns = None
    for mod in self.layers:
        output, attns = mod(output, memory, mode, tgt_mask=tgt_mask)
    return output, attns #return last layer attention
```

아래 이미지에 대한 각 단어별 attention map을 표현하면 다음과 같다. 각 단어별로 이미지에 어떤 파트에 주목할지가 조금씩 바뀌는 모습을 확인할 수 있다.

A motorcycle parked parked on to a street .
GT:<START> A motorcycle parked in a rocky lot by a wooden post .

A

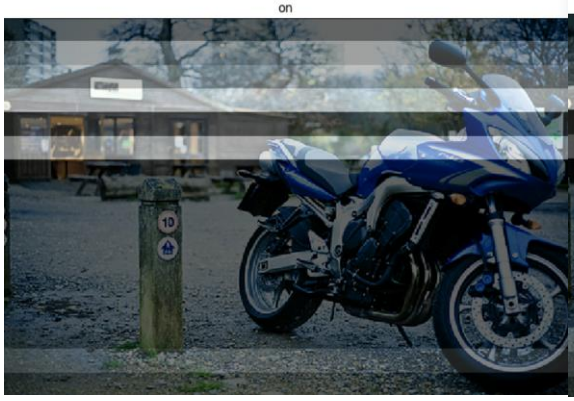
motorcycle



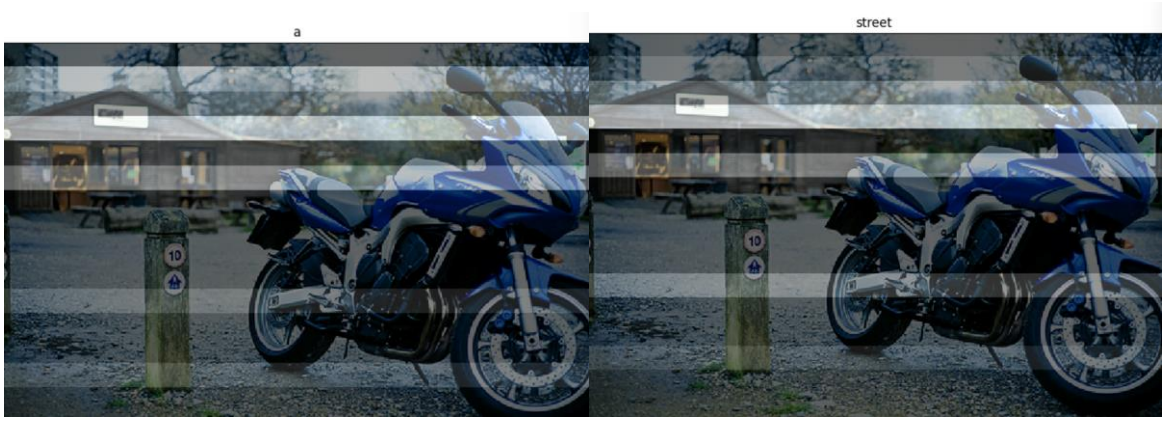
parked

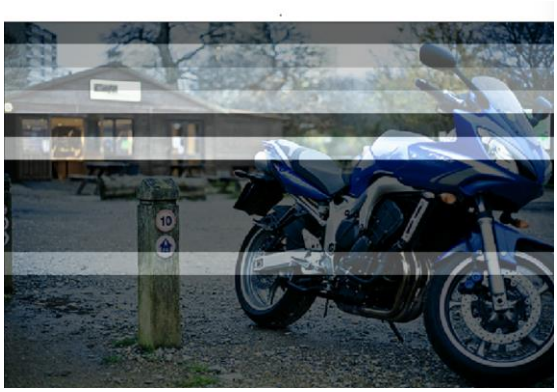
parked

on



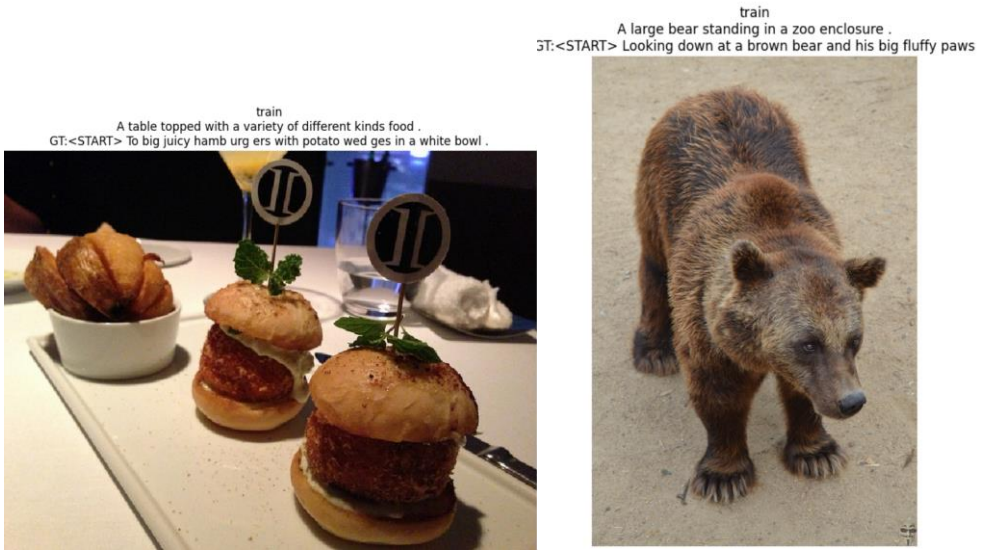
to





4. Inference on NICE challenge dataset

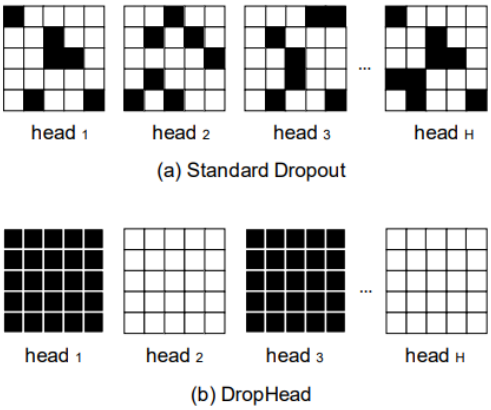
2에서 가장 좋은 모습을 보인 transformer를 전체 데이터셋에 대해서 train시킨 결과 CIDEr 점수도 train set과 validation set에서 각각 83과 74를 기록할 만큼 올랐다. Train과 validation set에 대한 정성적인 결과를 보더라도 상당히 그럴듯한 결과를 보임을 확인할 수 있었다.



이제 훈련시킨 모델을 nice dataset에서 확인해보니 아래와 같은 점수를 얻었다.

Bleu@1	Bleu@2	Bleu@3	Bleu@4	METEOR	CIDEr
47.45	31.69	20.81	13.99	18.18	38

그러던 중 "Scheduled DropHead: A Regularization Method for Transformer Models"라는 논문을 읽게 되었고, 일반적인 dropout 방식보다 각 head별 dropout 상태를 동일하게 하는 drophead 메서드가 더 좋은 성능을 보일 수 있음을 알게 되었다. 따라서 논문에 있는 pseudo code를 바탕으로 drophead 방법을 구현한 후 적용해봤다. 유일하게 달라진 부분은 count_ones가 0일 때 에러가 발생할 확률이 있어서 count_ones가 0이면 1로 바꾸어주었다. Count_ones가 0이라는 이야기는 모든 head에 대해 masking이 됐다는 의미이므로 나누어지는 상수를 바꾸어주어도 되기 때문이다.



Algorithm 1 DropHead

Require: Dropout rate p; outputs from H attention heads $O = [\text{head}_1; \ldots; \text{head}_H]$; mode

if mode == inference **then**
  return O
end if
Randomly sample ξ: $\xi_i \sim \text{Bernoulli}(p)$.
For each element ξ_i, create a mask vector $\mathbf{M}_i$ of the same shape as head_i with all elements equal to ξ_i.
Apply the mask: $O = O \times M$
Normalization: $O = O \times H / \textbf{count\_ones}(\xi)$

```python
def _drophead(self, out, mode):
    #out shape : (N, H, S, E//H)
    #mode : ("test" or "train")
    if mode == "test":
        return out
    N, H, _, _ = out.shape
    dist = torch.distributions.Bernoulli(torch.tensor([1 - 0.1]))
    mask = dist.sample((N, H)).unsqueeze(-1).cuda()
    count_ones = mask.sum(dim=1).unsqueeze(-1).cuda()
    count_ones[count_ones == 0] = 1
    out = out * mask * H / count_ones
    return out
```

그 결과 아래와 같이 성능이 오히려 안 좋아지는 것을 확인할 수 있었다. 해당 데이터셋에 경우 random하게 dropout을 하는 것이 조금 더 generalizability를 높일 수 있었기 때문이라고 생각된다.

Bleu@1	Bleu@2	Bleu@3	Bleu@4	METEOR	CIDEr
47.7	31.38	19.95	13.19	16.28	33.47