12주차 발표

딥러닝프레임워크

Attention code 분석

202184050 정가현 | 202184031 송민아 | Date. 2024.05.20

TABLE OF CONTENTS

목차 소개

01 | Seq2Seq

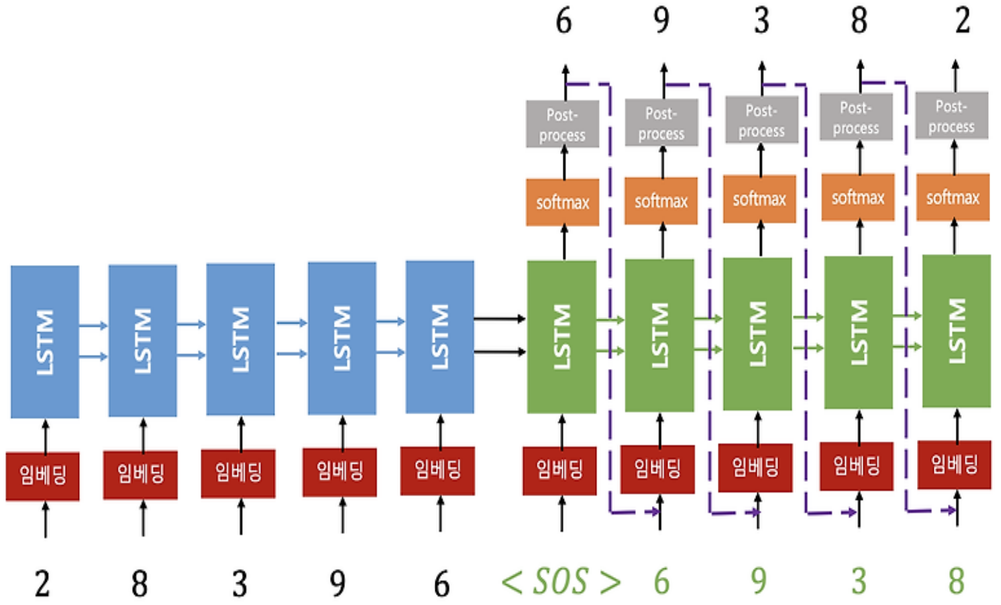
01 | Seq2Seq 란?

Seq : 시계열 데이터를 입력으로 받아서
2 : to
Seq : 시계열 데이터를 반환하는 것

02 | RNN (Recurrent Neural Networks)

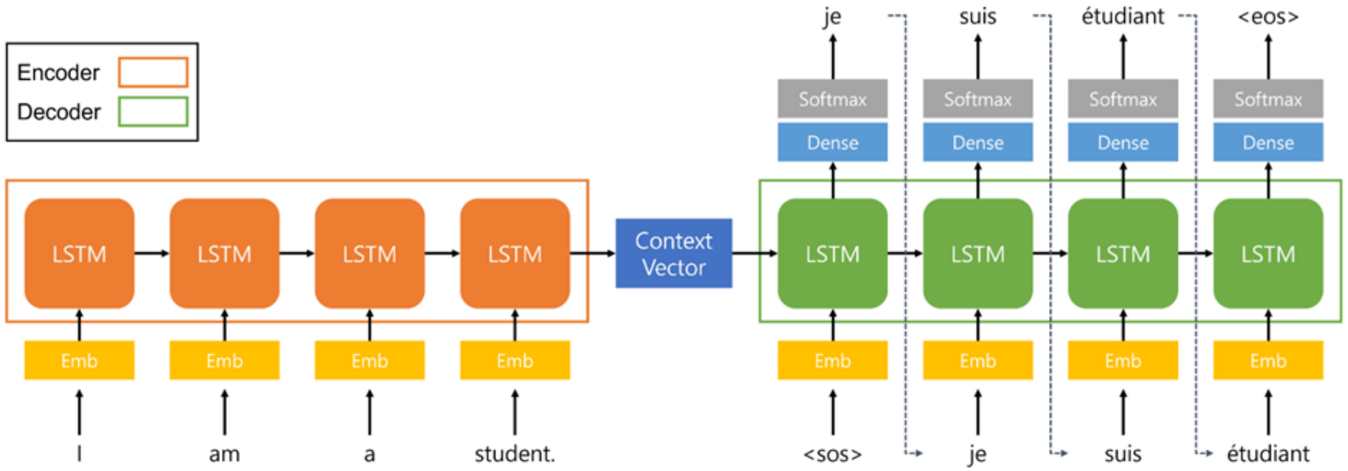
- 순환신경망은 입력층에서 출력층으로의 입력값 전달과 동시에 은닉층의 정보가 다음 은닉층으 로 이어지는 구조를 가진 신경망

Seq2Seq



02 | Attention이 사용된 이유

01 | seq2seq 한계점



현황에서 주목해야 할 중요한 특징이나 문제점을 간단히 설명합니다. 해당 페이지의 내용만 보고도 현황을 간단히 파악할 수 있도록요. 참고 자료에 적합한 자세한 내용을 적어주세요.

02 | Attention의 아이디어

Decoder에서 각 단어를 생성할 때, Context vector뿐만 아니라 은닉층의 모든 hidden state값과 현재 Decoder 셀에서 생성되는 단어와의 관계를 고려하자 !

03 | Attention code analysis

```python
# 시계열 데이터에 대해 어텐션 메커니즘을 구현한 코드
#Encoder가 출력하는 hs와 각 단어의 가중치 a를 작성한 후 가중합을 구하는것을 구현

import numpy as np

T, H = 5, 4  # 시계열 길이(T)를 5, 은닉 상태 벡터의 원소 수(H)를 4로 설정
hs = np.random.randn(T, H)  # (5, 4) 크기의 랜덤 행렬 생성, 인코더의 출력
a = np.array([0.8, 0.1, 0.03, 0.05, 0.02])  # 각 시계열 데이터의 가중치 (5,)

# 가중치를 (5, 1) 크기로 변경한 후, H=4번 반복하여 (5, 4) 크기의 가중치 행렬 생성
ar = a.reshape(5, 1).repeat(4, axis=1)
print(ar.shape)  # 출력: (5, 4)

# hs와 ar의 원소별 곱셈 (element-wise multiplication)
t = hs * ar
print(t.shape)  # 출력: (5, 4)

# t의 각 열에 대해 합계를 계산하여 컨텍스트 벡터 c를 생성 (각 열의 합계를 구함)
c = np.sum(t, axis=0)
print(c.shape)  # 출력: (4,)
```

✓ 0.1s Python

```
(5, 4)
(5, 4)
(4,)
```

01 | 시계열 데이터에 대해 어텐션 메커니즘을 구현한 코드

- T는 시계열 데이터의 길이(5)
- H는 은닉 상태 벡터의 차원(4)
- hs는 (5, 4) 크기의 랜덤 행렬, 인코더의 출력
- a는 (5,) 크기의 배열, 각 시계열 데이터의 가중치

03 | Attention code analysis

```python
#  N개의 시계열 데이터에 대해 어텐션 메커니즘을 적용하는 코드
import numpy as np

N, T, H = 10, 5, 4  # 배치 크기(N), 시계열 길이(T), 은닉 상태 벡터의 차원(H)
hs = np.random.randn(N, T, H)  # (10, 5, 4) 크기의 랜덤 행렬, 인코더의 출력
a = np.random.randn(N, T)  # (10, 5) 크기의 랜덤 행렬, 각 시계열 데이터의 가중치

# 가중치를 (N, T, 1)로 확장하고 H번 반복하여 (N, T, H) 크기의 가중치 행렬 생성
ar = a.reshape(N, T, 1).repeat(H, axis=2)
# ar = a.reshape(N, T, 1)  # 브로드캐스트를 사용하는 경우

t = hs * ar  # 인코더의 출력에 가중치를 적용하여 각 시계열 데이터의 컨텍스트 벡터를 계산
print(t.shape)  # 출력: (10, 5, 4)

c = np.sum(t, axis=1)  # 각 시계열 데이터에 대해 컨텍스트 벡터의 합을 계산
print(c.shape)  # 출력: (10, 4)
```

✓ 0.0s Python

```
(10, 5, 4)
(10, 4)
```

02 | N개의 시계열 데이터에 대해 어텐션 메커니즘을 적용하는 코드

- N은 배치 크기(10)로, 한 번에 처리할 시계열 데이터의 개수
- T는 시계열 데이터의 길이(5)
- H는 은닉 상태 벡터의 차원(4)
- hs는 (10, 5, 4) 크기의 랜덤 행렬, N개의 시계열 데이터 각각에 대한 인코더의 출력
- a는 (10, 5) 크기의 랜덤 행렬, N개의 시계열 데이터 각각에 대한 가중치

03 | Attention code analysis

```python
#위 그래프의 역전파를 구현
#'Repeat의 역전파는 Sum'이고 'Sum의 역전파는 Repeat'

import numpy as np

class WeightSum:
    def __init__(self):
        self.params, self.grads = [], [] # 학습하는 매개변수가 없어서 self.paramas=[]
        self.cache = None

    def forward(self, hs, a):
        N, T, H = hs.shape

        ar = a.reshape(N, T, 1).repeat(H, axis=2)  # 가중치를 (N, T, 1)로 reshape 후 H번 반복하
        t = hs * ar  # 원소별 곱셈
        c = np.sum(t, axis=1)  # 시계열 길이(T) 차원에 대해 합산하여 각 데이터에 대한 가중 평균 

        self.cache = (hs, ar)  # 순전파 계산에 사용된 값 저장
        return c

    def backward(self, dc):
        hs, ar = self.cache  # 순전파 때 계산했던 값 로드
        N, T, H = hs.shape

        dt = dc.reshape(N, 1, H).repeat(T, axis=1)  # Sum 노드의 역전파
        dar = dt * hs  # 원소별 곱셈
        dhs = dt * ar  # 원소별 곱셈
        da = np.sum(dar, axis=2)  # Repeat 노드의 역전파

        return dhs, da
```

✓ 0.0s Python

03 | WeightSum 클래스

- forward 메서드는 주어진 시계열 데이터(hs)와 가중치(a)를 사용하여 가중 평균을 계산
- 입력된 가중치 a를 (N, T, 1) 형태로 reshape하고, H 차원으로 repeat하여 (N, T, H)의 형태인 ar을 생성
- backward 메서드는 출력에 대한 미분값(dc)을 입력으로 받아 역전파를 수행
- Sum 노드의 역전파를 수행하기 위해 dc를 reshape하여 (N, 1, H)의 형태인 dt를 생성

03 | Attention code analysis

```python
import sys
sys.path.append('..')  # 상위 디렉토리를 파이썬 모듈 검색 경로에 추가

from common.layers import Softmax  # Softmax 레이어 임포트
import numpy as np  # NumPy 임포트

# 시계열 데이터의 크기와 은닉 상태의 크기 설정
N, T, H = 10, 5, 4

# 시계열 데이터의 형상을 가진 랜덤 행렬 생성
hs = np.random.randn(N, T, H)
# 현재 시점의 은닉 상태를 가진 랜덤 행렬 생성
h = np.random.randn(N, H)

# 브로드캐스트를 사용하여 현재 시점의 은닉 상태를 적절한 형상으로 변환
hr = h.reshape(N, 1, H).repeat(T, axis=1)
# hr = h.reshape(N, 1, H)  # 브로드캐스트를 사용하는 경우

# 각 시계열 데이터의 현재 시점의 은닉 상태와의 원소별 곱셈
t = hs * hr
print(t.shape)

# 각 시계열 데이터에 대한 가중 합을 계산하여 (N, T) 형태의 행렬 생성
s = np.sum(t, axis=2)
print(s.shape)

# Softmax 레이어를 이용하여 각 시계열 데이터에 대한 확률 분포 계산
softmax = Softmax()
a = softmax.forward(s)
print(a.shape)
```

✓ 0.0s Python

```
(10, 5, 4)
(10, 5)
(10, 5)
```

04 | 각 시계열 데이터의 중요도를 나타내는 가중치를 계산

- np.random.randn() 함수를 사용하여 시계열 데이터(hs)와 현재 시점의 은닉 상태(h)를 랜덤하게 생성
- h를 reshape하여 (N, 1, H)의 형태로 변환한 후 repeat() 메서드를 사용하여 시계열의 길이(T)만큼 반복하여 (N, T, H)의 형태인 hr을 생성
- 각 시계열 데이터의 현재 시점의 은닉 상태와의 원소별 곱셈을 수행하여 (N, T, H)의 형태인 t를 계산
- 각 시계열 데이터에 대한 가중 합을 계산하여 (N, T)의 형태인 s를 생성
- Softmax 레이어를 이용하여 각 시계열 데이터에 대한 확률 분포를 계산합니다. 결과적으로 (N, T)의 형태인 a를 얻음

03 | Attention code analysis

```python
import sys
sys.path.append('..')
from common.np import *  # NumPy 임포트
from common.layers import Softmax  # Softmax 레이어 임포트

class AttentionWeight:
    def __init__(self):
        self.params, self.grads = [], []  # 학습하는 매개변수가 없음
        self.softmax = Softmax()  # Softmax 레이어 초기화
        self.cache = None  # 순전파 시 사용되는 중간 결과 저장

    def forward(self, hs, h):
        N, T, H = hs.shape

        hr = h.reshape(N, 1, H).repeat(T, axis=1)  # 현재 시점의 은닉 상태를 적절한 형상으로 변환
        t = hs * hr  # 각 시계열 데이터의 현재 시점의 은닉 상태와의 원소별 곱셈
        s = np.sum(t, axis=2)  # 각 시계열 데이터에 대한 가중 합 계산
        a = self.softmax.forward(s)  # Softmax 레이어를 통해 각 시계열 데이터에 대한 확률 분포 계산

        self.cache = (hs, hr)  # 순전파 계산에 사용된 값 저장
        return a

    def backward(self, da):
        hs, hr = self.cache  # 순전파 때 계산했던 값 로드
        N, T, H = hs.shape

        ds = self.softmax.backward(da)  # Softmax 레이어의 역전파를 통해 출력에 대한 미분값 계산
        dt = ds.reshape(N, T, 1).repeat(H, axis=2)  # 각 시계열 데이터에 대한 미분값 계산
        dhs = dt * hr  # hs에 대한 미분값 계산
        dhr = dt * hs  # hr에 대한 미분값 계산
        dh = np.sum(dhr, axis=1)  # h에 대한 미분값 계산

        return dhs, dh
```

✓ 0.0s Python

05 | AttentionWeight 클래스

- forward 메서드는 주어진 시계열 데이터(hs)와 현재 시점의 은닉 상태(h)를 사용하여 각 시계열 데이터의 중요도를 나타내는 어텐션 가중치를 계산
- 현재 시점의 은닉 상태를 적절한 형상으로 변환한 후 각 시계열 데이터와의 원소별 곱셈을 수행하여 가중 합을 계산
- Softmax 레이어를 통해 각 시계열 데이터에 대한 확률 분포를 계산하고, 결과를 반환
- backward 메서드는 출력에 대한 미분값(da)을 입력으로 받아 역전파를 수행
- Softmax 레이어의 역전파를 통해 출력에 대한 미분값을 계산하고, 각 시계열 데이터에 대한 미분값을 구함

03 | Attention code analysis

```python
class Attention:
    def __init__(self):
        self.params, self.grads = [], []  # 학습하는 매개변수가 없음
        self.attention_weight_layer = AttentionWeight()  # AttentionWeight 계층 초기화
        self.weight_sum_layer = WeightSum()  # WeightSum 계층 초기화
        self.attention_weight = None  # 각 단어의 가중치를 저장하기 위한 변수

    def forward(self, hs, h):
        a = self.attention_weight_layer.forward(hs, h)  # AttentionWeight 계층에 의한 순전파 수행
        out = self.weight_sum_layer.forward(hs, a)  # WeightSum 계층에 의한 순전파 수행

        # 각 단어의 가중치를 나중에 참조하도록 attention_weight 변수에 저장
        self.attention_weight = a
        return out

    def backward(self, dout):
        dhs0, da = self.weight_sum_layer.backward(dout)  # WeightSum 계층에 의한 역전파 수행

        # AttentionWeight 계층에 의한 역전파 수행
        dhs1, dh = self.attention_weight_layer.backward(da)
        dhs = dhs0 + dhs1  # WeightSum 계층과 AttentionWeight 계층의 역전파 결과를 합산
        return dhs, dh
```

✓ 0.0s Python

06 | Attention 클래스

- AttentionWeight 계층을 통해 어텐션 가중치를 계산
- WeightSum 계층을 통해 시계열 데이터에 어텐션 가중치를 적용하여 가중 평균을 계산
- WeightSum 계층에 대한 역전파를 수행하여 시계열 데이터에 대한 미분값과 어텐션 가중치에 대한 미분값을 얻음
- AttentionWeight 계층에 대한 역전파를 수행하여 시계열 데이터에 대한 미분값과 현재 시점의 은닉 상태에 대한 미분값을 얻음
- 얻은 두 미분값을 합산하여 반환

03 | Attention code analysis

```
lass TimeAttention:
    def __init__(self):
        self.params, self.grads = [], []  # 학습하는 매개변수가 없음
        self.layers = None  # Attention 계층을 담을 리스트
        self.attention_weights = None  # 각 Attention 계층에서 계산된 가중치를 담을 리스트

    def forward(self, hs_enc, hs_dec):
        N, T, H = hs_dec.shape  # 디코더의 형상 정보를 획득
        out = np.empty_like(hs_dec)  # 디코더의 출력을 저장할 배열 생성
        self.layers = []  # 사용된 Attention 계층을 저장할 리스트 초기화
        self.attention_weights = []  # 각 Attention 계층에서 계산된 가중치를 저장할 리스트 초기화

        for t in range(T):  # 디코더의 시간 방향으로 반복
            layer = Attention()  # Attention 계층 생성

            # 해당 시점의 Attention 계층을 사용하여 출력 계산
            out[:, t, :] = layer.forward(hs_enc, hs_dec[:, t, :])
            self.layers.append(layer)  # 사용된 Attention 계층을 리스트에 추가

            # 해당 시점의 Attention 계층에서 계산된 가중치를 리스트에 추가
            self.attention_weights.append(layer.attention_weight)

        return out

    def backward(self, dout):
        N, T, H = dout.shape  # 출력의 형상 정보를 획득
        dhs_enc = 0  # 인코더의 역전파 결과 초기화
        dhs_dec = np.empty_like(dout)  # 디코더의 역전파 결과를 저장할 배열 생성

        for t in range(T):  # 디코더의 시간 방향으로 역순으로 반복
            layer = self.layers[t]  # 해당 시점의 Attention 계층을 가져옴
            dhs, dh = layer.backward(dout[:, t, :])  # 해당 시점의 Attention 계층에 대한 역전파 수행
            dhs_enc += dhs  # 인코더의 역전파 결과 누적
            dhs_dec[:, t, :] = dh  # 디코더의 역전파 결과 저장

        return dhs_enc, dhs_dec
```

✓ 0.0s Python

07 | TimeAttention 클래스

- 디코더의 시간 방향으로 반복하면서 각 시점에 대해 Attention 계층을 생성하고, 해당 시점의 Attention 계층을 사용하여 출력을 계산
- 사용된 Attention 계층과 해당 시점에서 계산된 가중치를 리스트에 저장
- 디코더의 시간 방향으로 역순으로 반복하면서 각 시점에 대해 해당 시점의 Attention 계층에 대한 역전파 수행
- 각 시점에서의 역전파 결과를 누적하여 인코더의 역전파 결과와 디코더의 역전파 결과를 반환

03 | Attention code analysis

```python
#8.2.1 Encoder구현

import sys
sys.path.append('..')
from common.time_layers import*
from ch07.seq2seq import Encoder, Seq2seq
from ch08.attention_layer import TimeAttention
class AttentionEncoder(Encoder):
    def forward(self, xs):
        xs = self.embed.forward(xs)  # Embedding 계층을 통해 입력을 단어 벡터로 변환

        # LSTM 계층을 통해 시계열 데이터를 처리하여 은닉 상태를 반환
        hs = self.lstm.forward(xs)
        return hs  # 시계열 방향으로 펼쳐진 은닉 상태를 반환

    def backward(self, dhs):
        dout = self.lstm.backward(dhs)  # LSTM 계층에 대한 역전파 수행
        dout = self.embed.backward(dout)  # Embedding 계층에 대한 역전파 수행
        return dout
```

✓ 0.0s Python

08 | AttentionEncoder 클래스

- 입력 데이터를 Embedding 계층을 통해 단어 벡터로 변환 후 LSTM 계층을 통해 시계열 데이터를 처리하고, 모든 시점의 은닉 상태를 반환
- 반환된 은닉 상태는 seq2seq 모델에서는 마지막 시점의 은닉 상태였지만, Attention을 추가한 경우에는 모든 시점의 은닉 상태를 그대로 반환
- STM 계층에 대한 역전파를 수행하여 LSTM 계층의 가중치 및 편향에 대한 미분값을 계산
- Embedding 계층에 대한 역전파를 수행하여 Embedding 계층의 가중치에 대한 미분값을 계산

03 | Attention code analysis

09 | AttentionDecoder 클래스

```
ss AttentionDecoder:
 def __init__(self, vocab_size, wordvec_size, hidden_size): # 어텐션 디코더 계층의 초기화
    V, D, H = vocab_size, wordvec_size, hidden_size
    rn = np.random.randn
    # 임베딩 가중치 초기화
    embed_W = (rn(V, D) / 100).astype('f')
    # LSTM 가중치 초기화
    lstm_Wx = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
    lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
    lstm_b = np.zeros(4 * H).astype('f')
    # 어텐션 가중치 초기화
    # 기존 디코더와 달리 어텐션 계층을 추가하여 2*H로 변경
    affine_W = (rn(2 * H, V) / np.sqrt(2 * H)).astype('f')
    affine_b = np.zeros(V).astype('f')

    # 계층 초기화
    self.embed = TimeEmbedding(embed_W)
    self.lstm = TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=True)
    self.attention = TimeAttention()  # 어텐션 레이어 초기화
    self.affine = TimeAffine(affine_W, affine_b)
    layers = [self.embed, self.lstm, self.attention, self.affine]

    # 매개변수 및 기울기 초기화
    self.params, self.grads = [], []
    for layer in layers:
        self.params += layer.params
        self.grads += layer.grads

 def forward(self, xs, enc_hs):
    h = enc_hs[:, -1]
    self.lstm.set_state(h)
    out = self.embed.forward(xs)
    dec_hs = self.lstm.forward(out)
    c = self.attention.forward(enc_hs, dec_hs)  # 어텐션 메커니즘 적용

    # Time Attention 계층의 출력과 LSTM 계층의 출력을 연결
    out = np.concatenate((c, dec_hs), axis=2)
    score = self.affine.forward(out)

    return score
```

```
def backward(self, dscore):
    # 합쳐진 출력을 역전파하기 위해 H=H2//2로 나눈다.
    dout = self.affine.backward(dscore)
    N, T, H2 = dout.shape
    H = H2 // 2

    dc, ddec_hs0 = dout[:, :, :H], dout[:, :, H:]
    denc_hs, ddec_hs1 = self.attention.backward(dc)
    # Affine에 들어간 hs와 Attention에 들어간 hs를 더하여 lstm.backward()를 진행
    ddec_hs = ddec_hs0 + ddec_hs1
    dout = self.lstm.backward(ddec_hs)
    dh = self.lstm.dh
    denc_hs[:, -1] += dh
    self.embed.backward(dout)

    return denc_hs

# 새로운 단어열(혹은 문자열)을 생성하는 generate() 메서드
def generate(self, enc_hs, start_id, sample_size):
    sampled = []
    sample_id = start_id
    h = enc_hs[:, -1]
    self.lstm.set_state(h)

    for _ in range(sample_size):
        x = np.array([sample_id]).reshape((1, 1))

        out = self.embed.forward(x)
        dec_hs = self.lstm.forward(out)
        c = self.attention.forward(enc_hs, dec_hs)
        out = np.concatenate((c, dec_hs), axis=2)
        score = self.affine.forward(out)

        sample_id = np.argmax(score.flatten())
        sampled.append(sample_id)

    return sampled
```

✓ 0.0s Python

03 | Attention code analysis

```python
from ch07.seq2seq import Encoder, Seq2seq

class AttentionSeq2seq(Seq2seq): # Seq2seq 클래스를 상속하여 AttentionSeq2seq 클래스 정의
    def __init__(self, vocab_size, wordvec_size, hidden_size):

        # vocab_size, wordvec_size, hidden_size를 args 변수에 할당
        agrs = vocab_size, wordvec_size, hidden_size
        self.encoder = AttentionEncoder(*args) # AttentionEncoder 초기화
        self.decoder = AttentionDecoder(*args) # AttentionDecoder 초기화
        self.softmax = TimeSoftmaxWithLoss()  # Softmax Loss 계층 초기화

        # 매개변수와 기울기 초기화

        # Encoder와 Decoder의 매개변수 결합
        self.params = self.encoder.params + self.decoder.params

        # Encoder와 Decoder의 기울기 결합
        self.grads = self.encoder.grads + self.decoder.grads
```

✓ 0.0s Python

10 | AttentionSeq2seq 클래스

- AttentionSeq2seq 클래스의 초기화를 수행
- 주어진 vocab_size, wordvec_size, hidden_size를 이용하여 AttentionEncoder와 AttentionDecoder를 초기화
- 시간 차원의 Softmax 계층인 TimeSoftmaxWithLoss를 초기화
- 각 계층의 매개변수와 기울기를 합쳐서 모델의 매개변수(params)와 기울기(grads)로 설정

03 | Attention code analysis

11 |
seq2seq 모델과 어텐션 메커니즘을 사용한 모델을 훈련하고 평가하는 과정

```
port sys
s.path.append('..')
s.path.append('../ch07')
port numpy as np
om dataset import sequence
om common.optimizer import Adam
om common.trainer import Trainer
om common.util import eval_seq2seq
om attention_seq2seq import AttentionSeq2seq
om ch07.seq2seq import Seq2seq
om ch07.peeky_seq2seq import PeekySeq2seq

데이터 읽기
_train, t_train), (x_test, t_test) = sequence.load_data('date.txt')  # 'date.txt' 데이터셋 로드
ar_to_id, id_to_char = sequence.get_vocab()  # 어휘 사전 생성

입력 문장 반전(Reverse)
입력 문장을 반전시켜 모델이 역순으로 입력을 받도록 함
train, x_test = x_train[:, ::-1], x_test[:, ::-1]

하이퍼파라미터 설정
cab_size = len(char_to_id)  # 어휘 사전 크기
rdvec_size = 16  # 단어 벡터의 차원 수
dden_size = 256  # 은닉 상태의 차원 수
tch_size = 128  # 미니배치 크기
x_epoch = 10  # 최대 에폭 수
x_grad = 5.0  # 기울기 클리핑을 위한 임계값
```

```
# 어텐션을 적용한 시퀀스-투-시퀀스 모델 생성
model = AttentionSeq2seq(vocab_size, wordvec_size, hidden_size)
optimizer = Adam()  # Adam 옵티마이저 생성
trainer = Trainer(model, optimizer)  # 훈련을 위한 Trainer 객체 생성

# 에폭마다 테스트 데이터를 사용하여 정답률 측정
acc_list = []  # 정확도 저장을 위한 리스트 초기화
for epoch in range(max_epoch):
    # 모델 학습
    trainer.fit(x_train, t_train, max_epoch=1, batch_size=batch_size, max_grad=max_grad)

    # 정답률 측정
    correct_num = 0
    for i in range(len(x_test)):
        question, correct = x_test[[i]], t_test[[i]]  # 테스트용 데이터 선택
        verbose = i < 10  # 상세한 출력 여부 결정

        # 평가 수행
        correct_num += eval_seq2seq(model, question, correct, id_to_char, verbose, is_reverse=True)

    acc = float(correct_num) / len(x_test)  # 정확도 계산
    acc_list.append(acc)  # 정확도 리스트에 추가
    print('val acc %.3f%%' % (acc * 100))  # 정확도 출력

model.save_params()  # 학습된 모델 파라미터 저장
```

2m 27.1s Python

```
| 에폭 1 |  반복 1 / 351 | 시간 1[s] | 손실 4.08
| 에폭 1 |  반복 21 / 351 | 시간 13[s] | 손실 3.09
```

감사합니다 !

202184050 정가현 | 202184031 송민아 | Date. 2024.05.20