[Vehicle Future Position Prediction]

수행 과제 보고서

- 서 형 진 -

안녕하십니까, ML 파트 과제를 수행하게 된 서형진 학생입니다.

이 문서에는 "Vehicle Future Position Prediction" 과제를 풀기 위한 간단한 과정과 그 결과의 내용을 적었습니다.

제출한 파일 중 최상위 폴더에 위치한 **"단계.txt"**를 보시면 시간 순서대로 어떤 작업을 하였고, 이를 위해 어떤 파일을 작성했는지 기록한 것을 보실 수 있으시지만 단순히 파일 주석만으로는 설명이 힘들거나 주석으로 담기에는 애매했던 것, 그리고 최종 결과 정리와 같은 내용은 따로 보고서 형태로 작성하여 드리는 것이 좋을 것 같아 판단하여 이 보고서를 작성하게 되었습니다.

"단계.txt" 문서와 코드, 그리고 이 보고서를 함께 참조하시면서 보시면 더욱 편하게 채점하실 수 있을 것이라 생각합니다.

감사합니다!

[Index]

0. 폴더 및 파일 구조

단계.txt

시간 순서대로 작업 내용을 정리한 txt 파일입니다

memo.txt

data

- 1222
  - kart_data_for_analyze.p
  - test_data.p
  - train_data.p
  - val_data.p
- 1224
  - test_data.p
  - train_data.p
  - val_data.p
  - test_30step_data.p
  - train_30step_data.p
  - val_30step_data.p
- 1224_img
  - scaler
    - posX_scaler.pt
    - posY_scaler.pt
  - test_data.p
  - train_data.p
  - val_data.p

preprocess 과정을 거친 데이터를 저장한 폴더입니다

각각 train, test, validation 데이터로 나뉘어 있습니다.

kart_data_for_analyze.p는 전처리 이전 데이터 분석을 위해 CSV 파일을 pickle로 저장한 파일입니다.

1224 폴더의 *_30step_*은,
본래 100step (0.01x100=1초) 단위로 끊었던 데이터를 30step (0.01x30=0.3초)로 다시 끊어 저장한 것입니다.

1224_img 폴더는 데이터를 시계열 형식이 아닌 이미지 형식으로 바꾸어 저장한 데이터가 들어있습니다.
주어진 데이터를 어떻게 이미지 형태로 바꾸었는지는 이후 전처리 챕터에서 자세히 설명하겠습니다.

scaler 폴더는 position을 0~1로 normalization한 sklearn의 MinMaxScaler을 저장한 모델 파일을 담고 있습니다.
이후에 스케일링한 데이터를 원래대로 복원할 때 사용하기 위해 저장했습니다.

models

- model_1223
  - model_dX, model_dY
    - parm
  - final_dX.pt, final_dY.pt
- model_1224_dsa, model_1224_img ,model_1226_tf, model_1226_tfCnn
  - model_d/parm
    - #.pt
  - final_d.pt

모델이 훈련하면서 각 epoch마다의 모델의 파라미터를 저장한 폴더입니다.

공통으로 model_d/parm/#.pt 구조를 가지며,
#은 epoch에 해당하는 숫자가 적혀있습니다.
단, 모든 epoch의 모델의 용량이 280GB 정도이기에, 제출 파일에는 가장 좋은 성능을 보인 epoch의 모델만 남겨 제출했습니다.

model_1223은 LSTM Attention model을,
model_1224는 DSA를,
model_1224_img는 SimpleCNN을,
model_1226_tf는 Transformer Base를,
model_1226_tfCnn은 Transformer with CNN을
저장하고 있습니다.

model_1223의 경우, LSTM Attention model이 실패하였기 때문에 폴더가 비어있습니다.

preprocess
- data_analyze
  - 1222_correlation.csv
  - 1222_covariance.csv
  - 1222_pValue.csv
  - check_covariance.py
  - cor_and_pValue_analysis.png
  - preprocess_for_analyze.py
- preprocess_img.py
- preprocess.py

전처리 코드와 데이터 분석 결과가 담긴 폴더입니다.

data_analyze 폴더에는 공분산, 상관관계, p-value를 계산하는 코드와 그 결과를 기록한 csv, png 파일이 들어있습니다.

preprocess_img.py는 데이터를 이미지 형식으로, preprocess.py는 데이터를 시계열 형식으로 전처리하는 코드입니다.

train_logs
- 1223_trainLog_lstmAttention.txt
- 1224_trainLog_CNN.txt
- 1224_trainLog_DSA.txt
- 1224_trainLog_DSA_30step.txt
- 1226_trainLog_TF.txt
- 1226_trainLog_TF_with_CNN.txt

각 모델의 train log를 저장한 폴더입니다.

각 파일 이름은 "훈련날짜_trainLog_모델type.txt" 으로 구성되어 있습니다.

main
- 1222_train.py
- 1224_train.py
- 1224v2_train_img.py
- 1226_train_tf.py
- data_manager.py
- loss.py
- model_deprecated.py
- model.py
- train.sh
- utils.py
- test_data
  - img
    - t.pt, x.pt
  - tf
    - c.pt, dirx.pt, diry.pt ,dx.pt, dy.pt, linx.pt, liny.pt ,n.pt, t.pt
  - p.pt
  - x.pt

훈련, 모델, loss, 유틸 함수, 테스트 등의 파일과 폴더들이 들어있는 main 폴더입니다.

1222_train.py는 LSTM Attention Model을, 1224_train.py는 DSA를, 1224_v2_train_img.py는 SimpleCNN을, 1226_train_tf.py는 transformer와 with CNN을 훈련할 때 사용하는 코드입니다.

data_manager.py는 훈련/테스트를 위한 데이터를 샘플링 및 로드하는 함수가 구현되어 있습니다.

loss.py는 loss계산과 관련된 함수가 구현돼 있습니다.

model.py는 모든 모델 코드가 적혀있으며, deprecated는 중간에 버려진 모델을 기록해놓은 파일입니다.

utils.py는 모델 파라미터 업데이트 함수, string을 bool로 바꿔주는 함수 등 유틸 함수가 구현되어 있습니다.

train.sh는 각 train 코드를 적절한 옵션으로 편하게 실행하기 위해 작성한 bash 파일입니다.

test_data 폴더에는 data_manager.py와 model.py에서 구현한 함수와 모듈이 정상적으로 작동하는지 확인하기 위한 torch tensor가 저장되어 있습니다.

- test
  - calculate_attention.py
  - calculate_cnn.py
  - calculate_tf.py
  - calculate.sh
  - plot_attention.py
  - plot_cnn.py
  - plot_tf.py
  - plot.sh
  - extrach_loss.py
  - test_logs
    - json
      - result_*.json
    - *.txt
  - plots
    - CNN, DSA, TF, TFwithCNN
      - #.png
  - extracted_loss_log
    - CNN, DSA, TF, TFwithCNN
      - train_loss.txt
      - dev_loss.txt

test 폴더는 log와 test data set을 이용하여 모델의 마지막 성능 점검을 진행한 내용이 저장되어 있습니다

calculate_*.py는 주어진 모델이 어떤 epoch일 때 test set에 대하여 가장 좋은 성능을 가지는지 알아내기 위해 test set에 대한 loss를 계산합니다. 이 과정은 calculate.sh로 실행하며, 그 결과는 "test_logs/json/result_*.json" 파일에 기록됩니다.

extracted_loss_log 폴더에는 extrach_loss.py를 통해 학습 과정에서 발생한 train loss와 dev loss를 학습 로그 파일에서 추출하여 저장합니다. 이후 loss 변화 그래프를 그리기 위해 제작했습니다.

plot_*.py 에서는 위의 calculate를 통해 알아낸 test set에서 가장 좋은 성능을 가지는 epoch의 모델을 로드하여 '현재 위치/경로', '실제 다음 위치', '예측 다음 위치'를 plot.scatter로 그린 후 png 파일로 저장합니다. 이 과정은 plot.sh를 통해 진행되며, png 파일은 "plots/[model]/#.png"에 저장됩니다.

1. 전처리

I. 데이터 분석 및 분석 결과

주어진 과제의 목표는 0.01초 이후의 X와 Y 위치였습니다. 따라서 저는 가장 먼저 X position과 Y position에 영향을 주는 데이터가 무엇인지 알아내야 할 필요가 있다고 생각했습니다.

이 작업을 위해 저는 공분산과 상관계수, 그리고 이에 대한 신뢰도를 측정하기 위한 p-value를 계산하기로 했습니다.

이를 위해 사용한 코드와 분석 결과는 **"preprocess/data_analyze"** 폴더에 담겨있으며 이 과정은 12월 22일에 진행했습니다.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1		posX	posY	posZ	**dirX**	**dirY**	dirZ	**linX**	**linY**	linZ	angX	angY	angZ
2	posX	0	-0.084040336	-0.009181364	0.002790434	0.376731452	0.00127461	0.00341844	0.398991151	-0.001185723	0.008907316	-0.02657691	-0.000485286
3	posY	-0.084040336	0	0.177789829	-0.667635163	-0.05956689	0.00431122	-0.71118693	0.000220805	-0.001206346	-0.071158414	0.114781001	-0.138911927
4	posZ	-0.009181364	0.177789829	0	0.010833069	0.000727217	0.01454857	0.004727906	0.009155066	0.01063711	0.150015975	-0.003027834	-0.011403656
5	dirX	0.002790434	-0.667635163	0.010833069	0	-0.00524112	-0.01249288	0.98185262	-0.005287729	-0.003959596	-0.009164524	0.024254512	-0.040871093
6	dirY	0.376731452	-0.059566891	0.000727217	-0.005241121	0	-0.00413145	-0.00596656	0.982326919	-0.004043008	-0.102479933	-0.044116918	-0.283156767
7	dirZ	0.001274605	0.004311216	0.01454857	-0.012492875	-0.00413145	0	-0.00461057	-0.004323268	0.973456939	-0.051535964	0.0306102	0.064884615
8	linX	0.00341844	-0.71118693	0.004727906	0.98185262	-0.00596656	-0.00461057	0	-0.00888807	-0.001251609	0.001914762	0.038681028	-0.040766431
9	linY	0.398991151	0.000220805	0.009155066	-0.005287729	0.982326919	-0.00432327	-0.00888807	0	-0.002789189	-0.117776875	-0.025710248	-0.283680231
10	linZ	-0.001185723	-0.001206346	0.01063711	-0.003959596	-0.00404301	0.97345694	-0.00125161	-0.002789189	0	-0.042237049	0.02159326	0.059922597
11	angX	0.008907316	-0.071158414	0.150015975	-0.009164524	-0.10247993	-0.05153596	0.001914762	-0.117776875	-0.042237049	0	-0.106832005	0.122616086
12	angY	-0.02657691	0.114781001	-0.003027834	0.024254512	-0.04411692	0.0306102	0.038681028	-0.025710248	0.02159326	-0.106832005	0	-0.088893915
13	angZ	-0.000485286	-0.138911927	-0.011403656	-0.040871093	-0.28315677	0.06488462	-0.04076643	-0.283680231	0.059922597	0.122616086	-0.088893915	0
14													
15	[p_value]												
16		0	1.43E-25	0.254327685	0.729014984	0	0.87426478	0.671270896	0	0.882967106	0.268780579	0.00096638	0.951957502
17		1.43E-25	0	1.12E-109	0	1.34E-13	0.59248057	0	0.978130156	0.880946692	9.10E-19	2.27E-46	2.79E-67
18													

위 이미지는 correlation과 p-value를 각 데이터 항목별로 적어놓은 엑셀 파일입니다.

분석 결과, X position의 경우는 [Y direction], [Y linear velocity] 데이터와 Y position의 경우는 [X direction], [X linear velocity] 데이터와 높은 correlation을 가지며 동시에 신뢰할만한 p-value를 가짐을 알 수 있었습니다.

Z 축 정보도 필요할지, 그리고 각속도 정보도 필요할지 고민했었지만,
분석 결과, 모델에게는 방향과 선형 속도 정보만 주어도 충분히 좋은 결과를 계산할 것이라는 결론을 내릴 수 있었고 데이터 전처리 방향을 잡을 수 있었습니다.

II. 전처리 방향 선택

```
######################
#      설  정        #
######################
# 1) time step length는 0.01s의 100배인 1s으로 설정하여 신경망 모델이 지난 1s의 데이터를 보고 앞의 0.01s 의 결과를 예측하도록 한다
# 2) dirX, dirY, linX, linZ, angX, angY, angZ 각각 0 ~ 1의 값을 가지도록 normalize 한다
#    맵의 크기는 정해져있으므로 posX와 posY를 normalize해도 실제 주행에 사용할 수 있도록 변환할 수 있지만, 일단 여기서는 제외하고 해보자.
#    우선은 모델의 input으로 들어가지도 않고, 더 세밀하거나 극적인 변화에 대응할 수 있도록 유도할 수 있을 것이라고 생각한다.
# 3) target 값은 현재(1s의 데이터 중 가장 마지막) posX, posY와 0.01s 이후(1.01s)의 posX, posY의 차이가 된다.
#    최종 결과가 좀 더 물리적으로 옳바를 수 있도록 모델이 최종 위치를 계산하는 그래프를 탐색하는 것이 아니라, 주어진 물리량에 대한 '변화값'을 계산하는 그래프를 탐색하도록 유도하기 위함이다
```

주어진 데이터는 0.01초 단위로 기록되어 있지 않았습니다. 그래서 저는 이 데이터를 동일한 time step (0.01s)를 가지도록 변경해야 했습니다.

가장 간단한 방법인 평균으로 이 문제를 해결했습니다. 각 데이터가 기록된 시간에서 소수점 3번째 자리에서 반올림하여 시간 간격을 0.01s로 만든 후, 동일한 시간의 데이터끼리 평균을 내었습니다.

그 결과 15,416개의 데이터가 나왔고, 저는 이 데이터를 14,000개는 train data로, 1,000개는 validation data로, 그리고 416개의 데이터는 test data로 이용하기로 했습니다.

그리고 저는 기본적으로 이 데이터를 시계열 형식으로 활용하기로 하였기에 일정한 time step 만큼 자를 필요가 있었고, 처음에는 총 100 step (0.01s * 100 = 1s) 으로 설정했습니다. 하지만 이후 훈련 속도가 더디어지는 문제가 발생하여 30 step으로 줄여 다시 전처리를 진행했습니다. 그 결과로 14,000개의 train data, 1,000개의 validation data, 그리고 (416-30-1=385) 개의 test data가 생성되었습니다.

먼저 전체 데이터에 대한 인덱스를 랜덤하게 섞어 앞에서부터 14000, 1000, 385개씩 끊어가며 data set을 만들었습니다.

data set에는 지금을 포함한 30 step 동안의 실제 XY 위치 정보와 direction /linear_velocity/angle_velocity 정보, 다음 step의 XY 위치, 그리고 (모델의 target data가 될) 현재 step과 다음 step의 XY 위치의 변화량(dx, dy)이 포함되어 있습니다.

모델의 target을 실제 위치가 아닌, 위치의 "변화량"으로 설정한 이유는 다음과 같습니다. 이 내용은 **"memo.txt"**에도 적혀있습니다. 고등학교 때 배운 물리를 떠올렸습니다. 물체의 다음 위치를 계산하는 것에 필요한 것은 물체의 질량과 현재 속도, 그리고 힘입니다. 이 3가지 물리량을 정확히 안다면 물체가 어떻게 얼마나 이동할 것인지 계산할 수 있습니다. 질량의 경우 여기서는 생략할 수 있고 속도는 linear velocity 정보가, 힘은 linear velocity 정보에 direction 정보를 함께 이용하여 구할 수 있으리라 생각했습니다. 그리고 이런 값들을 이용해서 물리 공식으로 얻을 수 있는 것은 최종 위치가 아닌, 위치의 변화량입니다. 저는 모델이 이러한 일종의 물리 공식을 학습하길 원했습니다. 따라서 모델이 바로 다음 위치를 출력하는 것이 아니라 현재 물리량을 보고 위치의 변화량을 계산한 후, 이 모델의 출력값과 현재 위치 정보를 더하여 최종 predicted next position을 구하기로 결심했습니다.

```
# 4) 여기서는 model input으로 img 형식이 들어간다. img는 다음과 같이 만든다
#    - pos 데이터도 0~1 범위를 가지도록 normalize 한다. 단, 변화량 계산은 normalize 전에 한다 (너무 작지 않고 적당한 loss 값을 얻기 위해)
#      실제 pos로 복원하기 위해서 MinMaxScaler도 저장한다 => 필요 없을지도? (cur pos와 next pos를 normalize 하기 전에 기록할 수 있으므로)
#    - ViT에서 사용한 아이디어를 일부 채용한다
#    - [[posX, posY, dirX], [dirY, linX, linY], [argX, argY, argZ]] 값을 가지는 3X3 블록을 만든다
#    - 위의 블록을 회전하면서 총 4개의 블록을 이어붙여 6X6 블록을 만든다
#    - 다시 위의 블록을 좌우대칭/상하대칭으로 이어붙여 6X6X3 블록을 만든다 => 이것이 하나의 이미지가 되어 모델에 들어간다
```

preprocess_img.py는 데이터를 SimpleCNN 모델에 사용하기 위한 전처리 과정이 담겨 있습니다. 주어진 데이터를 일종의 이미지 데이터로 변환하기 위한 과정이 주석과 코드로 적혀 있습니다. 이미지 데이터에는 XY 위치, XY 방향, XY 선형 속도, 그리고 XYZ 각속도 데이터를 이용하였고 모두 0~1의 범위를 가지도록 normalize 했습니다.

위 값들을 붙이고 쌓으면서 블록을 만들었습니다. 이것은 ViT 논문에서 영감을 받은 아이디어를 채용한 것입니다. ViT에서는 input으로 들어온 이미지를 일정한 단위로 자르고 여러 방법으로 padding, sliding 합니다. 저는 위에서 만든 블록을 좌우대칭, 상하 대칭, 그리고 회전하면서 3X3X1 크기의 데이터를 6X6X3 으로 확장하였고, 이를 모델의 input으로 설정하였습니다.

실제로는 사용하지 않았지만, normalize 한 position 값을 복원시키기 위해 normalize에 사용했던 sklearn의 MinMaxScaler도 **“data/1224_img/scaler”** 폴더에 저장했습니다.

III. main/data_manager.py 에 대한 부록

이 파일에는 pytorch 의 DataSet 모듈과 DataLoader 모듈의 역할을 대신하는 함수들이 구현되어 있습니다. 각각의 **sample_train_data_*** 함수는 위의 전처리를 거친 데이터를 로드하여 목적에 맞게 적절하게 변형하여 반환합니다. 그러면 **train_data_loader_*** 함수는 sample_train_data 함수의 결과물을 받은 후, mini_batch_size 에 맞게 데이터를 잘라 yield 합니다. yield 된 결과물을 받아 모델의 input 과 target 으로 사용합니다.

이 구조를 사용한 이유는, 학부 연구생을 하면서 데이터 처리에 대한 자유도와 유연성에 있어 pytorch 에 내장된 DataSet, DataLoader 모듈을 그대로 이용하는 것보다 훨씬 높은 이점을 가지고 있다는 것을 알았기 때문입니다. 비록 이번 과제에서는 이 구조의 장점을 모두 이용했다고 할 수는 없었지만, 더 익숙해지고 자유롭게 사용할 수 있도록 연마하기 위해 이번 과제에서도 이 구조를 사용해보았습니다.

loader 가 데이터를 받은 후 shuffle 된 인덱스 리스트와 함께 데이터를 sampler 에게 전달하면,

```
def train_data_loader_TF(data, batch_size, device, use_angle_v=False, shuffle=False) :
    total_len = len(data)
    idxs = np.arange(total_len)
    if shuffle :
        np.random.shuffle(idxs)
    dir_X, dir_Y, lin_X, lin_Y, prev_dX, prev_dY, cur_pos, next_pos, targets = sample_train_data_TF(data, idxs, use_angle_v)
```

sampler 에서 데이터에 적절한 변형을 가한 후 다시 전체 데이터를 반환합니다.

```
return dir_x, dir_y, lin_x, lin_y, prev_dx, prev_dy, cur_pos, next_pos, targets
```

그러면 loader 는 위의 데이터를 적절하게 자르고 배치하여 하나씩 yield 하는 흐름을 가집니다.

```
yield prev_dx, prev_dy, dir_x, dir_y, lin_x, lin_y, c, n, t
```

2. 사용한 모델

I. LSTM Attention Model

[선택한 이유]

가장 먼저 시도한 모델입니다. 데이터를 시계열 형식으로 사용하기로 하였고, "시계열 하면 역시 LSTM 이 먼저다"라는 생각으로 가장 먼저 시도하게 되었습니다. 그리고 일종의 트렌드 라고 할 수 있는 attention 도 함께 사용하면 좋을 것 같아 LSTM 과 Attention 기법이 혼합된 LSTM Attention 모델을 구현하게 되었습니다.
사실 지인분들과 시작한 프로젝트에서 저는 이 모델을 구현하고 있었습니다. 목적은 주식의 regression 이었고, 이번 과제와 목적이 비슷하기에 가장 먼저 시도한 것에 무의식적인 이유가 있었던 것 같기도 합니다.

[구현 과정]

이 이미지를 기반으로 구현했습니다. 단, 이 구조는 기계 번역을 가정하여 짜여 있기에 약간의 변형을 가했습니다. decoder 에 해당하는 LSTM cell 의 sequence length 가 1 이며 최종 output 역시 단어 사전의 index 가 아닌, 최종 regression 값으로 설정했습니다.

[모델 결과 및 요약]

결과적으로 이 모델은 실패했습니다. 어떤 input 이 들어와도 항상 0 이라는 결과를 냈기 때문입니다. 이 실패로 인해 다음 모델을 탐색하는 계기가 되었습니다.
다른 모델들을 구현하면서 그리고 보고서를 작성하면서 실패 원인을 분석해보니, classification 을 위한 모델을 regression 작업을 할 수 있도록 적절하게 변형시키지 못했던 것 같습니다. softmax 함수를 relu 로 변경하지 않은 것, 그리고 마지막 output layer 에 fc layer 를 추가하지 않은 것 등을 실패의 원인으로 분석했습니다..

II. Dual Stage Attention Model (DSA)

[선택한 이유]

attention 과 LSTM 을 그대로 사용하면서 regression 작업에 이용할 수 있는 방법이 무엇일까 계속 탐색했습니다. 그 결과 한 논문과 그에 대한 리뷰를 찾을 수 있었고, 그것이 Dual Stage Attention Based RNN 이었습니다.

[구현 과정]

https://arxiv.org/abs/1704.02971 이 논문과 https://simpling.tistory.com/12 이 포스팅을 참고하여 구현했습니다.

[모델 결과 및 요약]

생각보다 더 좋은 성능을 보여 놀랐습니다.
처음에는 100 step 의 time step 을 사용하였고, 각각 X 와 Y 위치의 변화량을 예측하는 모델 2 개를 사용했지만, 훈련 시간이 매우 오래 걸리는 것을 보고 훈련과정을 더 간단하게 바꾸었습니다. 데이터의 time step 은 30 으로, 그리고 X 와 Y 에 대한 각각의 모델을 따로 구현하는 것이 아니라 하나의 모델만을 훈련하되, input 에 차이점을 두어 모델이 X 와 Y 데이터 각각에 대한 학습 경로를 탐색하도록 유도했습니다.
100 step 의 데이터와 두 개의 모델을 사용했을 때보다 더 빠르게 좋은 성능을 보였지만, 여전히 빠른 속도라고는 할 수 없었기에 loss 가 적절한 값으로 수렴하기 전까지 남는 시간 동안 몇 가지 모델을 더 실험해보기로 했습니다.

III. Simple CNN Model

[선택한 이유]

인공지능 연구실에서 음성 관련 인공지능을 실험하면서 한 가지 배웠던 것은, 음성과 같이 이미지가 아닌 데이터도 얼마든지 이미지 데이터로 변환하여 사용할 수 있다는 것이었습니다. 대표적인 것이 MFCC Spectrogram 이었습니다. 그 이후부터 저는 인공지능 모델과 데이터를 다룰 때 항상 데이터를 이미지로 바꿔보는 것을 시도했습니다. 이번 과제에서도 주어진 데이터를 최대한 이미지답게 바꿔보았습니다.

[구현 과정]

먼저 custom convolution layer 를 만들었습니다. Conv2d_GLU 모듈은 pytorch 의 convolution layer 와 그것의 output 에 비선형성을 주는 모듈이라고 생각하시면 될 것 같습니다. Simple CNN 모델은 이 Conv2d_GLU 와 nn.Conv2d, 그리고 fully connected layer 로 이루어진 매우 간단한 CNN 모델입니다.

[모델 결과 및 요약]

test 결과 몇몇 case 에 대해서는 괜찮은 예상을 했지만, 대부분은 완전히 틀린 위치를 출력했습니다. 이에 대한 원인은 크게 2 가지가 있었던 것 같습니다.

먼저 데이터의 이미지화가 적절하지 못했다는 것입니다. 처음에 위치를 0~255 로 normalize 한 뒤에 지난 경로를 픽셀 좌표로, 그리고 그때의 방향과 속도 정보를 채널로 주려고 하였지만 한 가지 간과한 것이 있었습니다. 무언가의 '위치'가 데이터로 주어질 때는 대부분 float type 이지만, 이미지의 픽셀 좌표는 정수 단위로 이루어져 있다는 것입니다. 저는 모든 위치를 0~255 의 정수로 normalization 해보려고 했지만, 이 경우 너무 많은 정보가 손실된다고 생각했습니다. 대안으로 위의 **preprocess_img.py** 의 방법을 이용했지만, 역시 좋은 변환은 아니었던 것 같습니다.

두 번째는 DSA 와 달리, X 와 Y 의 위치 변화량을 예측하는 데에 완전히 동일한 input 을 사용했다는 것입니다. 결국 모델은 X 와 Y 데이터 사이의 차별성을 학습할 수 없었을 것이며 어쩌면 이 모델의 성능이 낮은 것은 피할 수 없었던 문제였던 것 같습니다.

첫 번째 문제점은 게임 플레이 AI 를 강화학습으로 만들기 위해 게임 유닛의 위치 데이터를 이미지화할 때도 맞닥뜨린 문제였습니다. 결국 이번에도 이 문제를 완벽히 해결하지 못한 것이 아쉽지만, Transformer 모델을 다루면서 한 가지 아이디어가 떠올랐습니다. 바로 vae 의 encoder 처럼, learnable custom Embedding Layer 를 구현하여 float type position 데이터를 long type pixel location 으로 바꾸어 latent vectorization 하는 것입니다. 이 아이디어를 게임 플레이 AI 제작에 이용해볼 예정입니다.

IV. Transformer Encoder Base Model

[선택한 이유]

저는 attention 기법을 좋아합니다. 그동안 black box 라고만 인식되었던 신경망의 학습 경로와 결과에 대한 근거를 파악할 수 있다고 생각하기 때문입니다. 실제로 제가 참고했던 포스팅 작성자 중 하나는, 기계 번역 Transformer 를 주식 regression 용도로 바꾸는 작업을 통해 deep learning 의 black box 문제를 해결할 수 있었다고 합니다 (https://www.linkedin.com/pulse/how-i-turned-nlp-transformer-time-series-predictor-zimbres-phd).
이런 제가 "Attention is all you need"를 표방한 Transformer 를 선택하는 것은 자연스러운 것일지도 모릅니다.

[구현 과정]

https://doheon.github.io/%EC%BD%94%EB%93%9C%EA%B5%AC%ED%98%84/time-series/ci-4.transformer-post/#conclusion 를 참고했습니다.
본래는 원본 Transformer 를 최대한 그대로 이용해보는 것을 시도했었지만, 실패로 돌아가고 위의 포스팅에서 영감을 받아 pytorch 에서 제공하는 Transformer 의 Encoder Layer 만 이용하기로 했습니다. Transformer 의 Encoder 가 들어온 데이터를 적절히 encoding 하면, 그것을 간단한 fully connected layer 에 통과시켜 위치 변화량을 예측하도록 했습니다. 단, Transformer 는 기본적으로 1 차원 데이터만을 이용하기에 총 5 가지 feature(지난 30 step 의 X 또는 Y 위치 변화량, X 방향, Y 방향, X 선형 속도, Y 선형 속도)가 있는 데이터를 그대로 이용할 수는 없었습니다. 그래서 총 5 개의 Transformer Layer 를 가지는 통합 모델을 만들고, layer 마다 하나의 feature 를 담당하여 encoding 한 후 각각의 결과를 concatenation 하여 fully connected layer 에 통과시키는 방법을 이용했습니다.

[모델 결과 및 요약]

(처음에 실패한 코드는 **"model_deprecated.py"**에 옮겼습니다)
train set 에서 괜찮은 loss 값을 보여주었지만, DSA 와 비교하면 부족한 부분이 드러났습니다. DSA 의 경우 validation set 에 대한 total_loss 가 28~27 쯤부터 수렴하기 시작했다면 Transformer 는 48~47 에서 수렴하는 것으로 보입니다. 그리고 계속 훈련이 진행되면서 train set 에 대한 loss 는 조금씩 줄어들고 있지만, validation set 에 대한 loss 는 크게 줄어들지 않습니다.
모델에 추가적인 개선이 필요했습니다.

V. Transformer Encoder with CNN Model

[선택한 이유]

위의 Transformer Encoder Base 모델을 개선한 모델입니다. 저는 5 개의 feature 에 대해 각각 다른 Transformer Encoder Layer 를 사용한 것이 문제라고 생각했습니다. 이 구조로 인해 모델이 충분하게 데이터의 모든 특성을 고루 살피지 못하게 된 것이라는 가정을 했습니다.

[구현 과정]

이에 대한 개선 방향은 다음과 같습니다.
먼저 주어진 5 개의 feature 를 하나로 모아 (batch_size, seq_length, 5)의 shape 을 가지는 데이터로 만든 후, CNN 을 통해 (batch_size, seq_length, 1)의 shape 를 가지는 latent vector 를 생성합니다. 그리고 이 vector 를 하나의 Transformer Encoder Layer 에 통과시키기로 했습니다. 마지막으로 Transformer Encoder Layer 의 출력을 다시 fully connected layer 에 통과시켜 최종 regression 예측값을 출력하도록 했습니다.
이는 모델이 데이터의 특성을 제대로 파악할 수 있도록 데이터를 CNN 과 Transformer Layer, 두 번의 encoding 과정을 거친 후, fully connected layer 에 데이터의 특성을 모두 포함한 좋은 latent vector 가 전달될 수 있도록 학습 경로를 유도함으로써 위의 Transformer Encoder Base 모델의 문제점을 해결하고자 한 것입니다

[모델 결과 및 요약]

가장 마지막으로 시도한 모델이 가장 빠른 훈련 속도와 가장 좋은 성능을 보였습니다. 마치 복선 가득한 소설을 읽는 것 같습니다.
모든 모델 중, train set 에 대한 loss 값은 1~0 에 가까울 정도로 매우 작았으며 validation set 에 대한 loss 역시 27~26 으로 가장 작았습니다.
하지만 모델에 대한 평가는 test set 으로 해야 정확할 것이며, 위의 모델들을 포함한 모델들에 대한 정확한 비교 분석은 이후 4 번 챕터에서 진행하겠습니다.

3. 훈련 결과

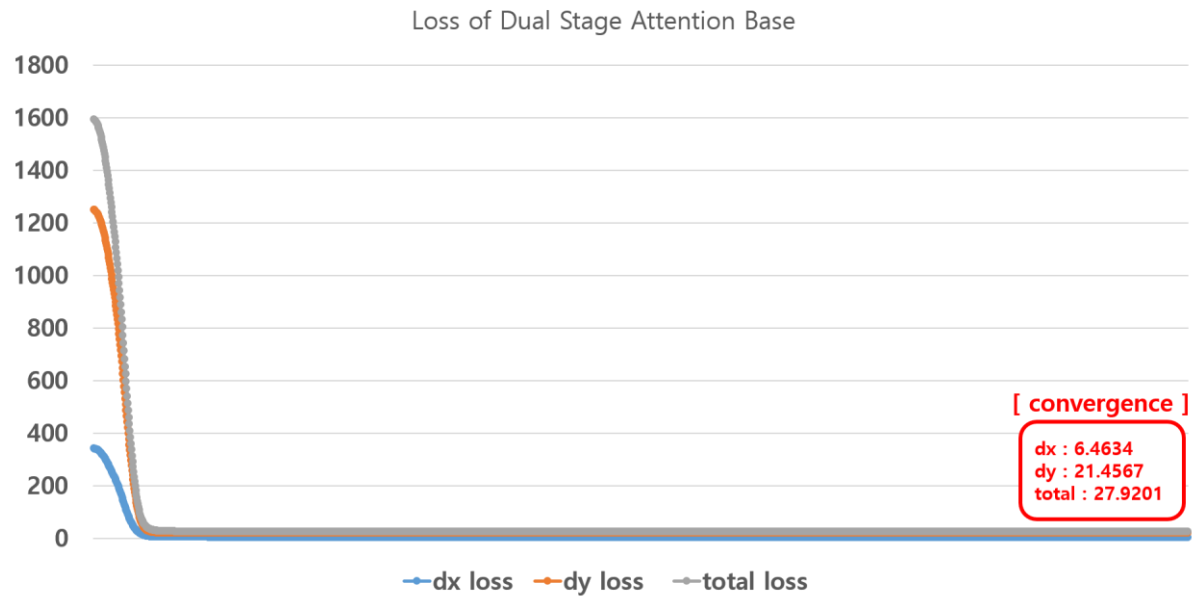
1) 이 챕터의 결과는 각 모델이 2000 epoch 까지 훈련됐을 때를 기준으로 했습니다.

2) 소수로 적힌 값들은 모두 소수점 네 번째 자리에서 반올림한 값입니다.

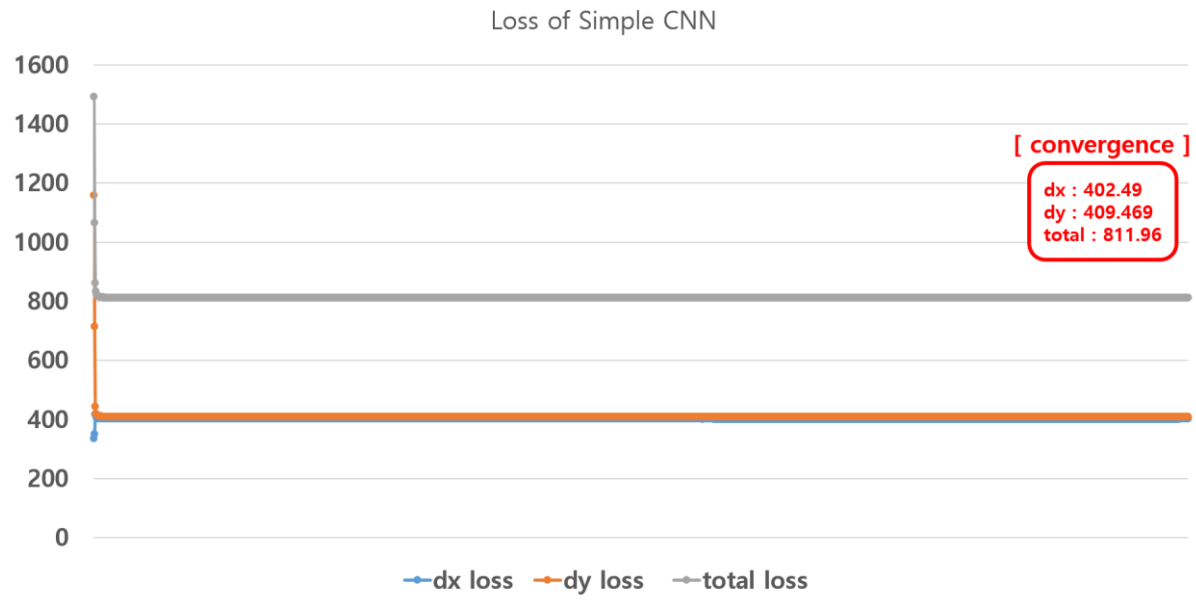
3) loss 는 Mean Squared Error 를 사용했습니다.

I. 각 모델의 train set 에 대한 loss graph

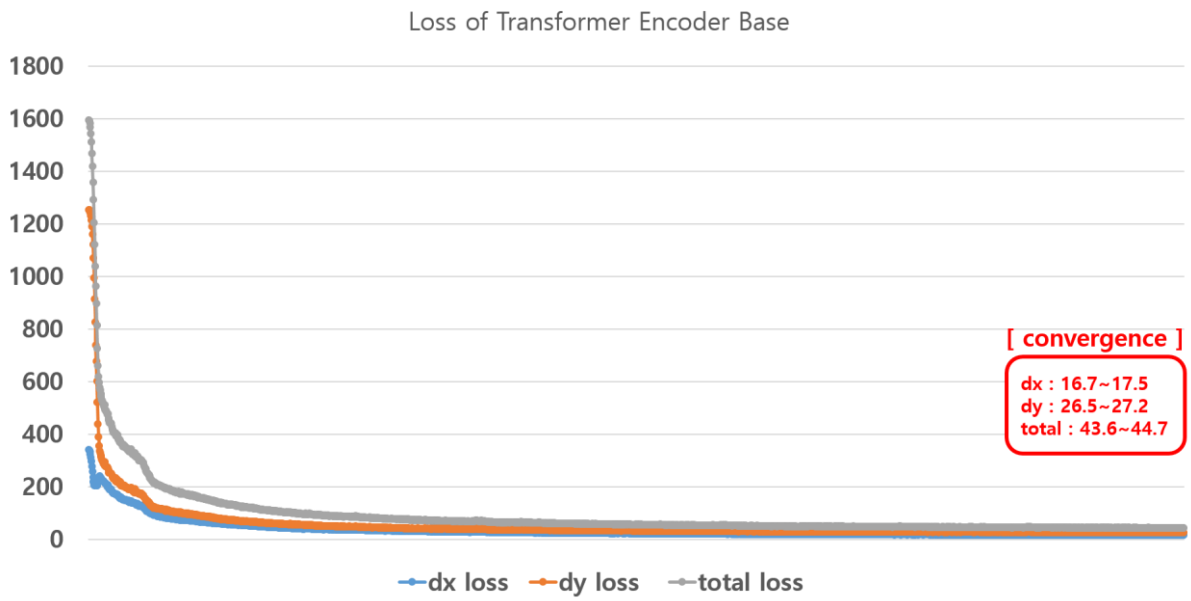
① Dual Stage Attention Base



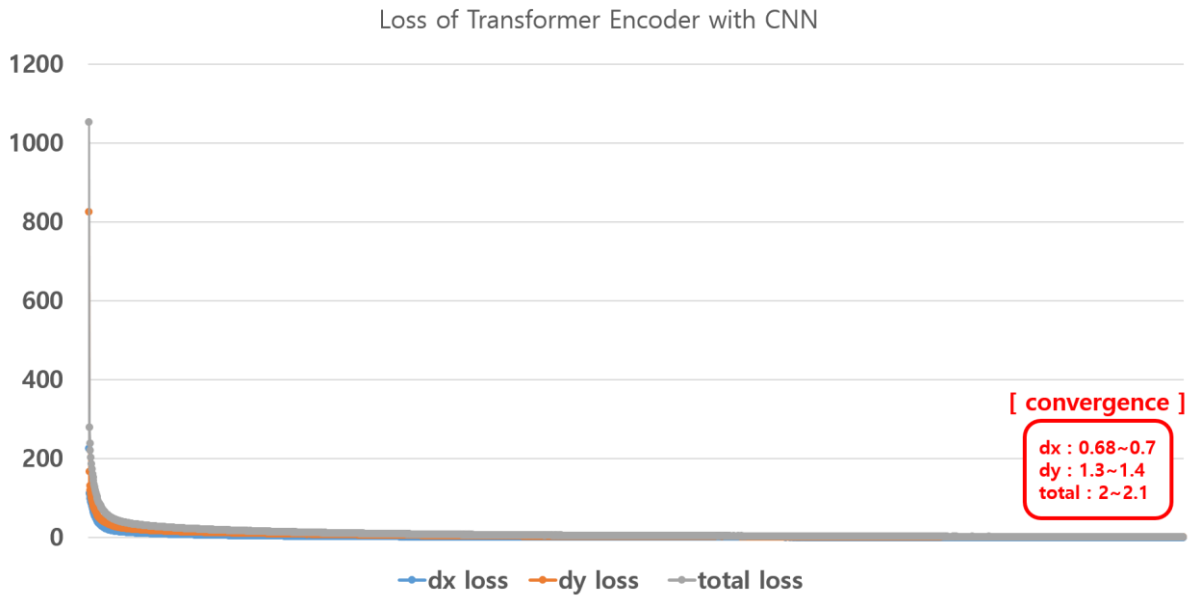
② Simple CNN



③ Transformer Encoder Base

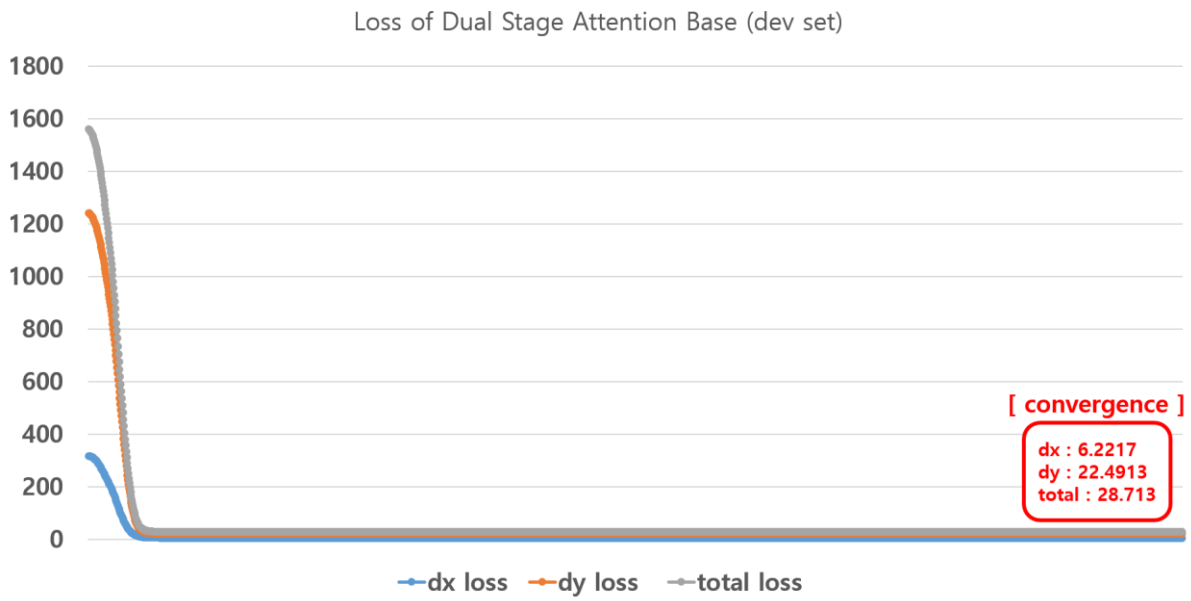


④ Transformer Encoder with CNN

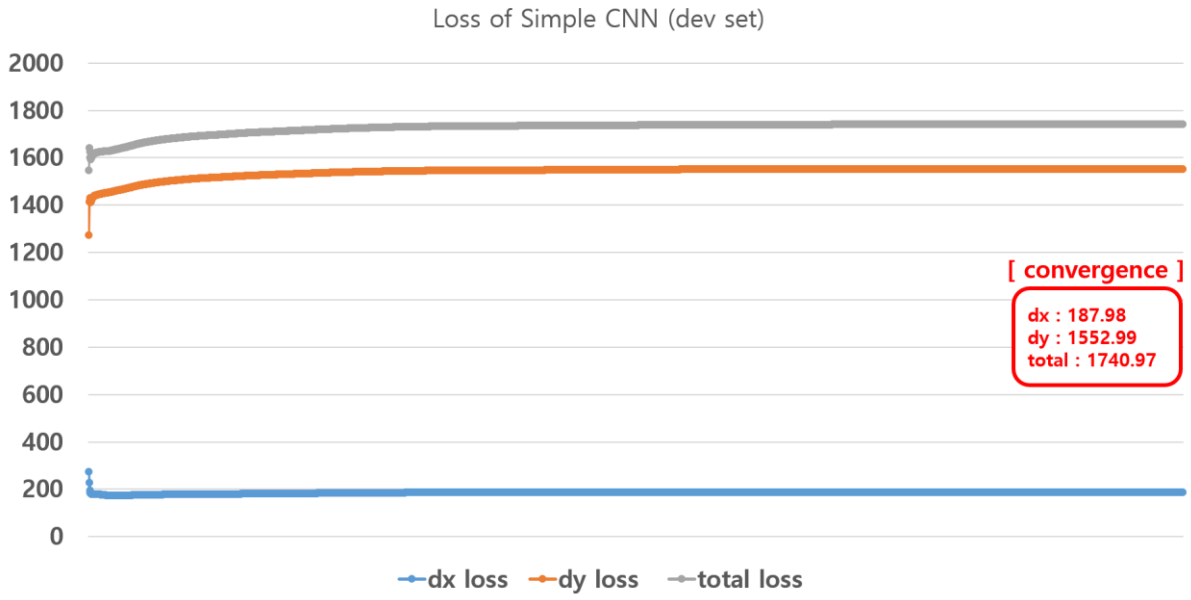


II. 각 모델의 validation set에 대한 loss graph

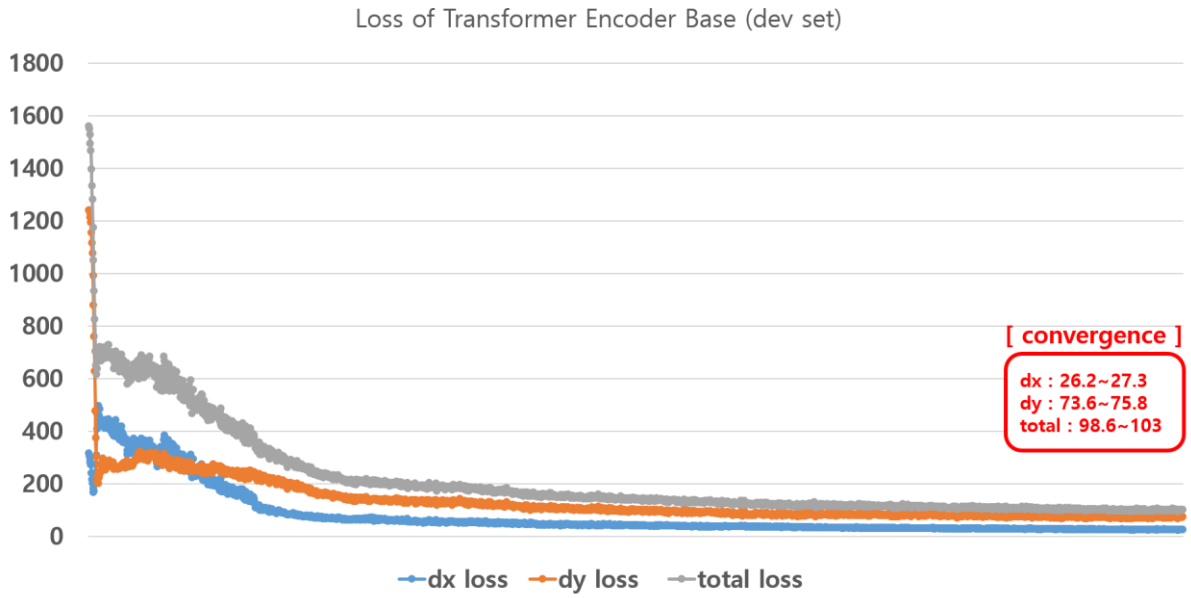
① Dual Stage Attention Base



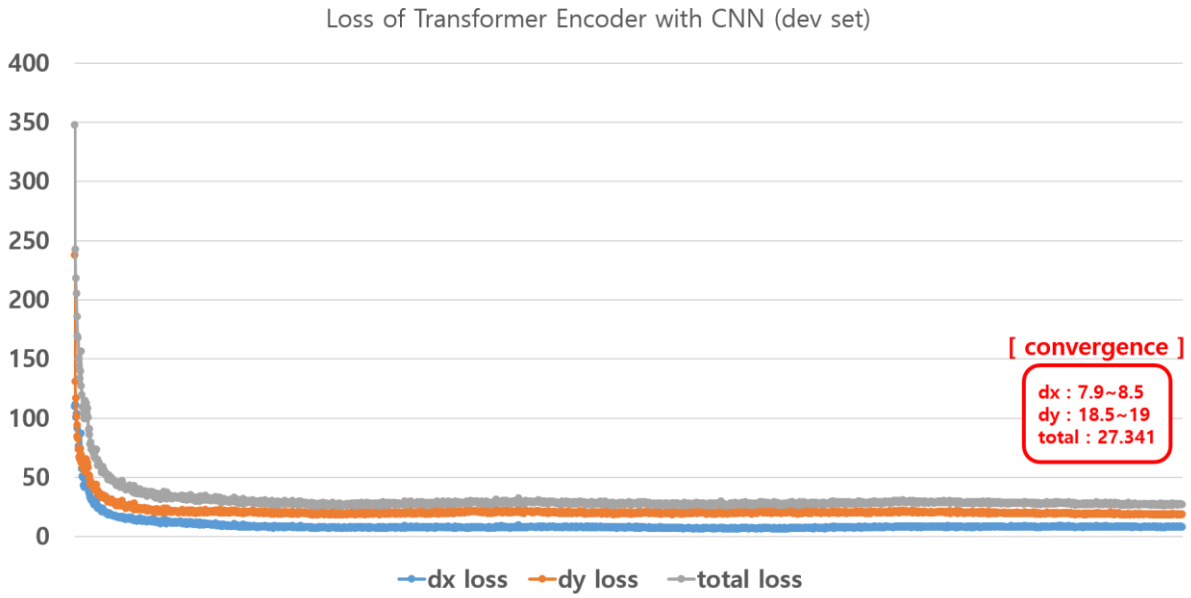
② Simple CNN



③ Transformer Encoder Base



④ Transformer Encoder with CNN



III. 최종 정리

	1 epoch당 훈련 시간	(train set) 최소 dx loss	(train set) 최소 dy loss	(train set) 최소 total loss	(validation set) 최소 dx loss	(validation set) 최소 dy loss	(validation set) 최소 total loss
LSTM Attention	X	X	X	X	X	X	X
DSA	152.727s	6.4603	21.4572	27.9206	6.2168	22.4913	28.7131
Simple CNN	4.785s	333.5617	409.4694	811.9623	174.4216	1272.747	1547.2884
TF Encoder Base	174.941s	16.5443	26.5495	43.5502	26.0907	68.1087	94.8863
TF Encoder with CNN	121.778s	0.6746	1.3036	2.008	6.384	18.3183	25.6338

위 loss graph 의 결과를 정리한 표입니다.

SimpleCNN 의 경우 훈련 속도를 제외하면 훈련 중 성능이 가장 떨어졌습니다.
Transformer Encoder Base 모델은 train set 에 대해서는 나쁘지 않은 결과를 보여주었지만, validation set 에 대해서는 안 좋은 성능을 보여줬습니다. 이와 비슷하게, Transformer Encoder with CNN 모델 또한 train set 에 비해 validation set 에 대한 성능이 안 좋게 나온 것을 확인할 수 있었습니다. 어쩌면 Transformer 를 baseline 으로 잡은 모델로 regression 을 하면 train set 에 약간의 overfitting 이 발생하는 것일지도 모릅니다.
반면 Dual Stage Attention Base 모델은 train set 에서나 validation set 에서나 비슷한 성능을 안정적으로 보여주고 있습니다. 모델의 안정성과 일관성만을 두고 본다면, Dual Stage Attention Base 모델이 overfitting 이나 underfitting 같은 문제에서 자유롭다고 할 수 있겠습니다.

그래서 위와 같은 결과가 나오고, 저는 어쩌면 test set 에 대해서는 Dual Stage Attention Base 모델의 성능이 가장 뛰어날 지도 모른다고 생각했습니다. 이 가정에 관한 결과를, 다음 4 번 챕터에서 확인하실 수 있습니다.

4. 최종 테스트 결과

1) 성능 지표(distance)는 "실제 다음 위치"와 "예측한 다음 위치"의 euclidean distance 입니다.
2) 이 챕터의 결과는 각 모델의 test set 에 대한 성능 측정에서 가장 좋은 성능을 보인 epoch 의 모델 parameter 를 load 하여 진행한 결과입니다.

I. 각 모델의 test set 에 대한 최종 성능

	mean distance	minimum distance	minimum epoch
LSTM Attention	X	X	X
DSA	4.1714937845636815	4.170640016412418	2000
Simple CNN	25.870343554636904	25.857960224542285	1996
TF Encoder Base	9.526382366922984	8.652956545972204	1822
TF Encoder with CNN	4.035073444774697	3.7355521398318277	1942

모든 test set 의 데이터에 대한 결과를 정리한 표입니다. column 마다 모델별로 test set 에 관한 euclidean distance 의 평균값, 최솟값, 그리고 최솟값일 때의 epoch 가 적혀 있습니다.

위 표를 보면, Simple CNN 모델의 성능이 가장 안 좋으며, Transformer Encoder with CNN 모델의 성능이 가장 좋은 것을 확인할 수 있습니다.

3 번 챕터의 표에서나, 이 표에서나 Dual Stage Attention Base 는 안정적인 성능을 가지고 있음을 알 수 있습니다. 평균 distance 나 최소 distance 의 차이가 거의 없었습니다. 또한 다른 모델들은 최종 epoch 인 2000 이 아닌 이전 epoch 에서 최적의 성능을 가지고 있는 것에 비해, Dual Stage Attention Base 는 가장 마지막 epoch 에서 최적의 성능을 가지고 있습니다. 실제로 **"train_logs/1224_trainLog_DSA_30step.txt"**를 보시면 Dual Stage Attention 의 loss 가 다시 상승하는 구간 없이, 느리지만 시간이 지날수록 계속 나아지고 있음을 알 수 있습니다.

다만, 저의 예상과는 달리 test set 에 대해서도 Transformer Encoder with CNN 모델의 성능이 가장 좋게 나왔습니다. 특히 최소 distance 에서 Dual Stage Attention Base 모델보다 더욱더 좋게 나온 것을 확인했습니다.

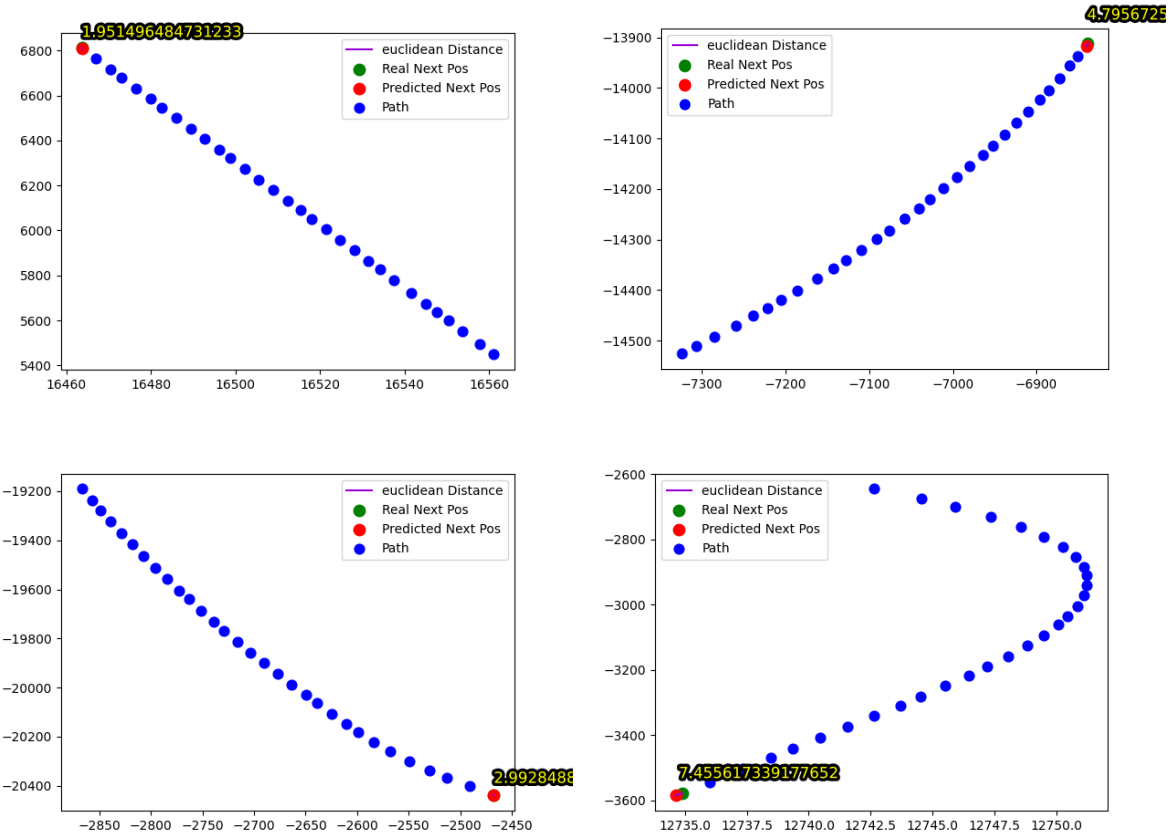
II. 각 모델의 test set 에 대한 real next pos, pred next pos 의 비교

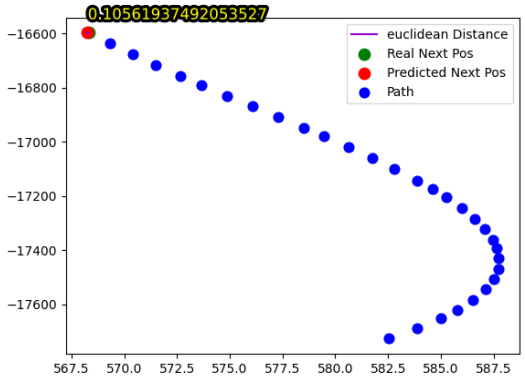
1) test set 에서 20(16)개의 case 를 샘플링하여 그린 그래프입니다. 푸른색 점은 현재 위치 또는 현재까지의 경로, 초록색 점은 0.01 초 후의 실제 위치, 붉은색 점은 0.01 초 후의 모델이 예상한 위치입니다.

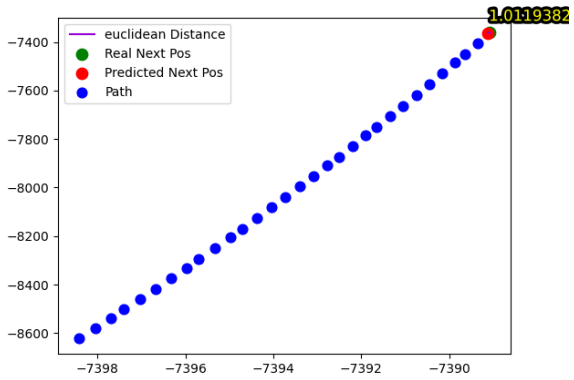
2) 그래프에 써진 숫자는 실제 위치와 모델의 예상 위치 사이의 euclidean distance 를 의미합니다.

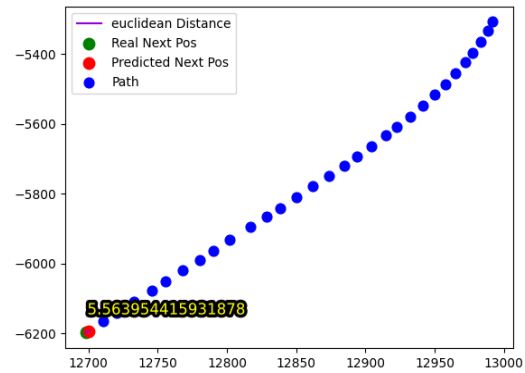
3) 이 graph 들은 모두 test_graph 폴더에 저장되어 있습니다.

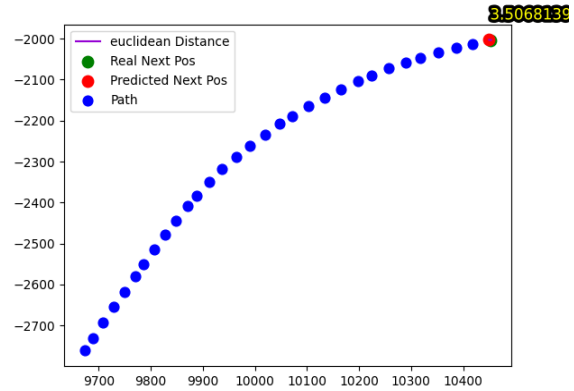
① Dual Stage Attention Base

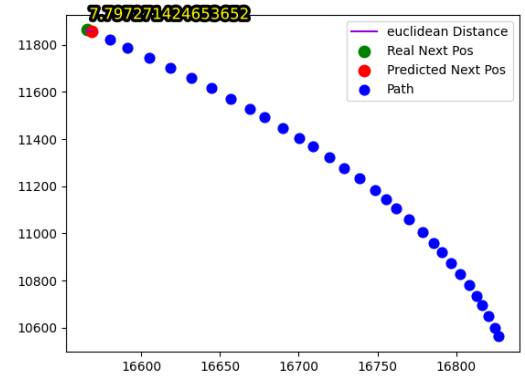


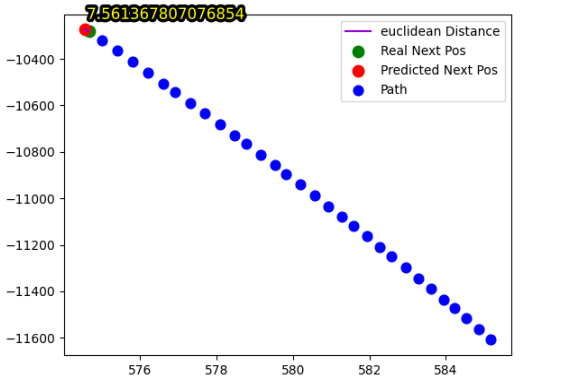

0.10561937492053527
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

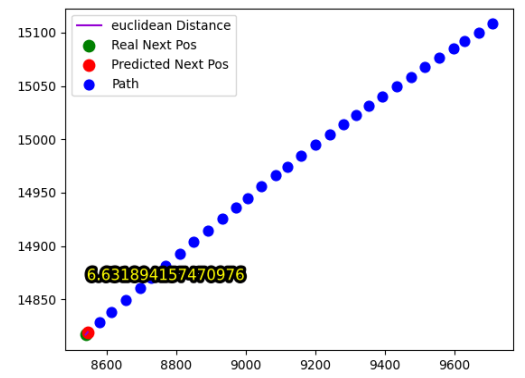

1.0119382
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

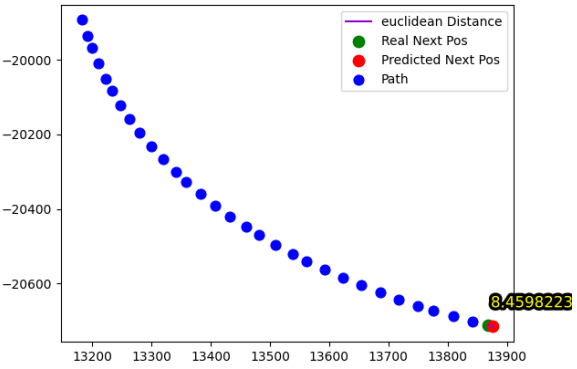

5.563954415931878
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

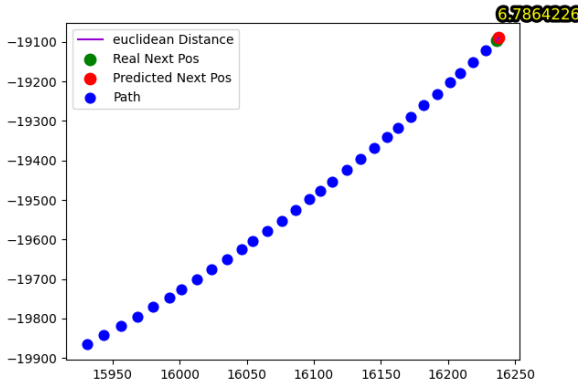

3.5068139
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

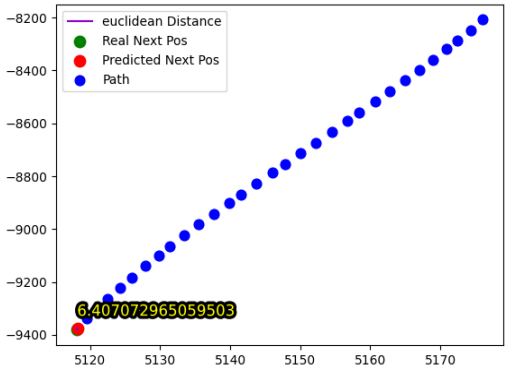

7.797271424653652
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

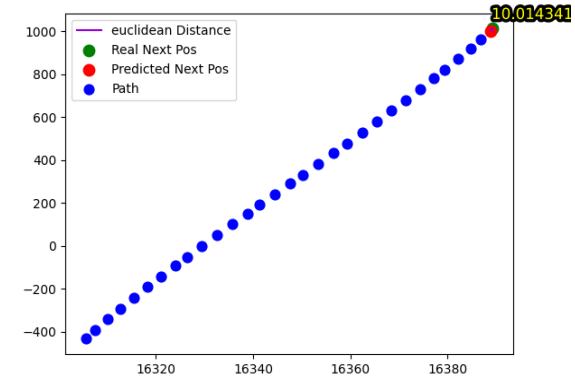

7.561367807076854
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

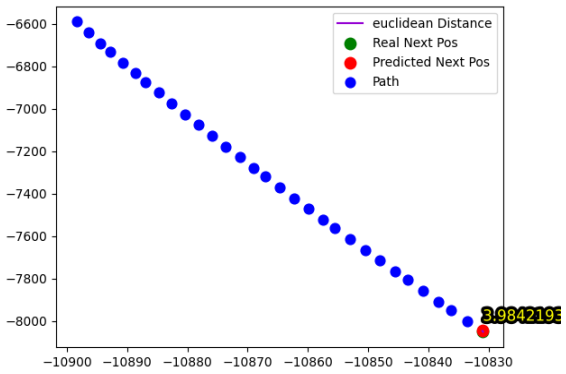

6.631894157470976
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

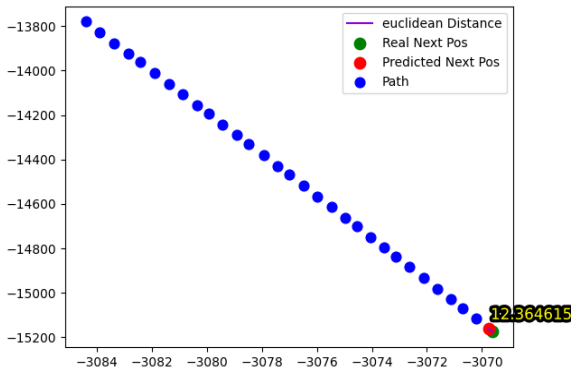

8.4598223
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

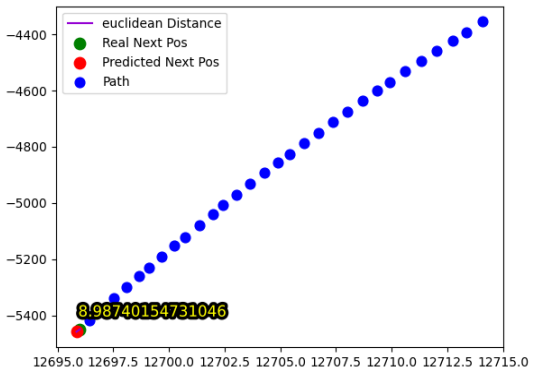

6.7864226
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

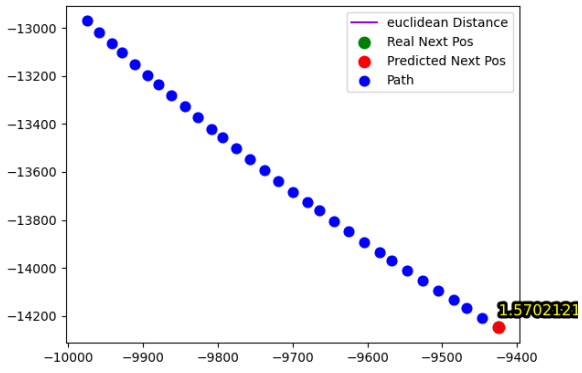

euclidean Distance
Real Next Pos
Predicted Next Pos
Path
6.407072965059503

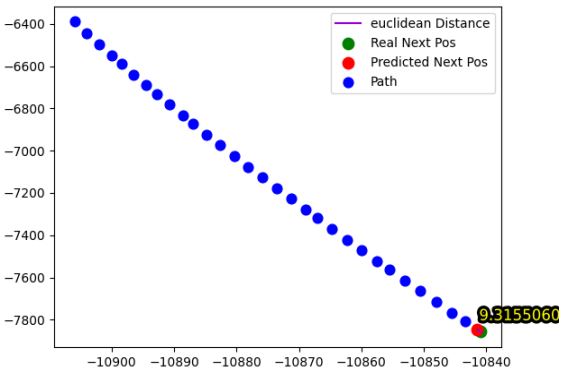

10.014341
euclidean Distance
Real Next Pos
Predicted Next Pos
Path


euclidean Distance
Real Next Pos
Predicted Next Pos
Path

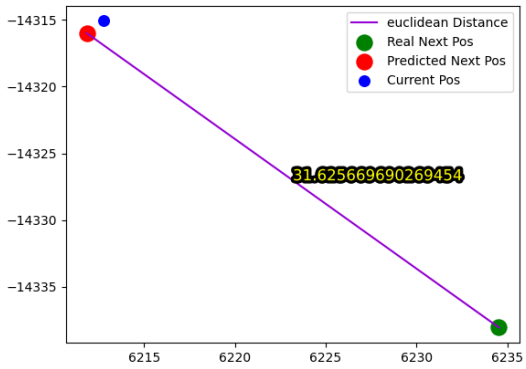

euclidean Distance
Real Next Pos
Predicted Next Pos
Path
12.364615

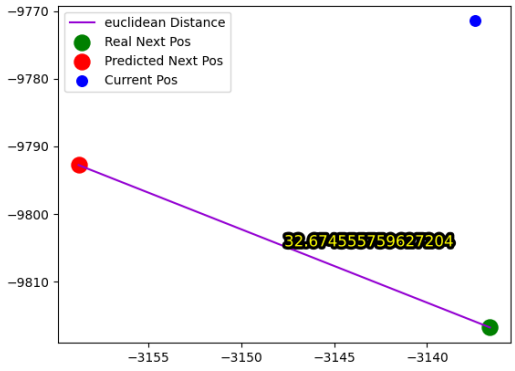

euclidean Distance
Real Next Pos
Predicted Next Pos
Path
8.987401547310 46

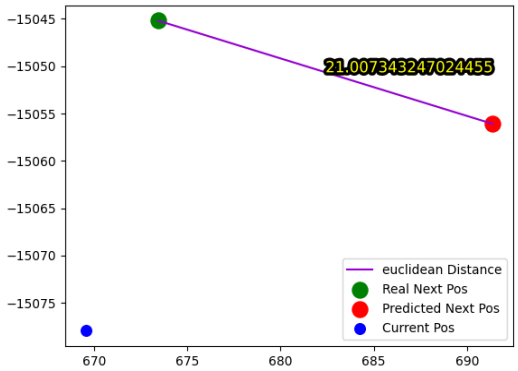

euclidean Distance
Real Next Pos
Predicted Next Pos
Path
1.5702121

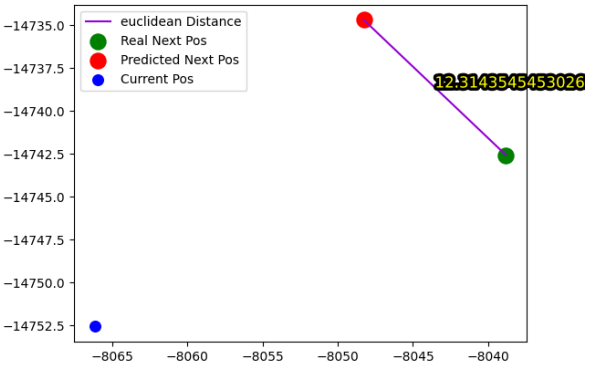

euclidean Distance
Real Next Pos
Predicted Next Pos
Path

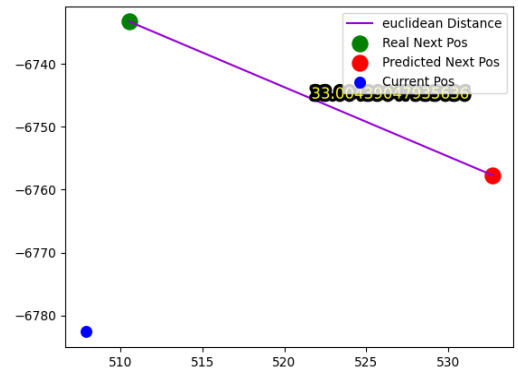
② Simple CNN

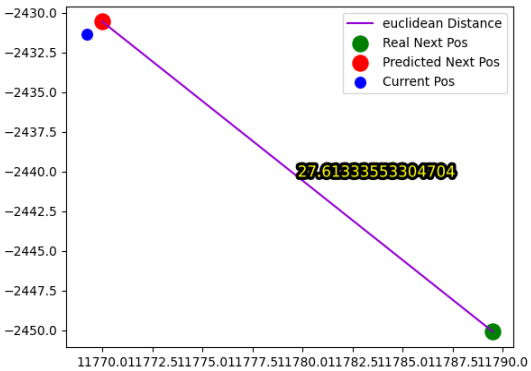

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
31.625669690269454

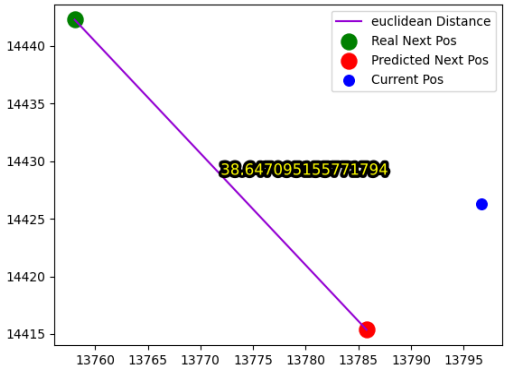

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
32.67455759627204

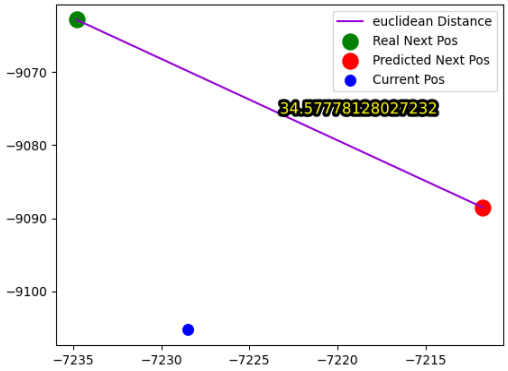

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
21.007343247024455

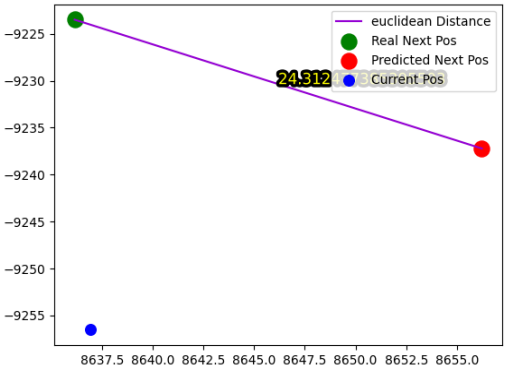

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
12.31435454530

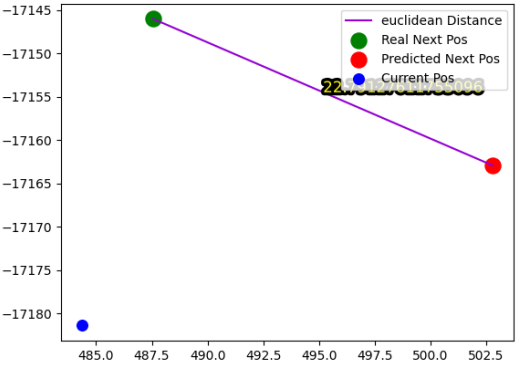

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos

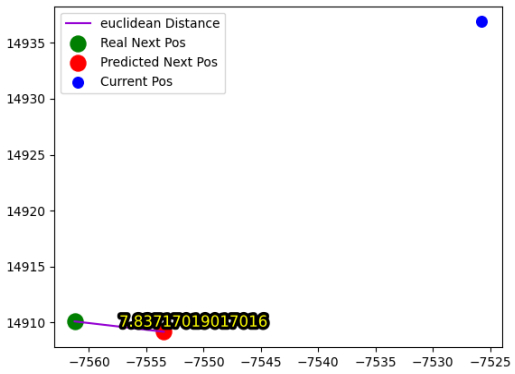

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
27.61333553304704

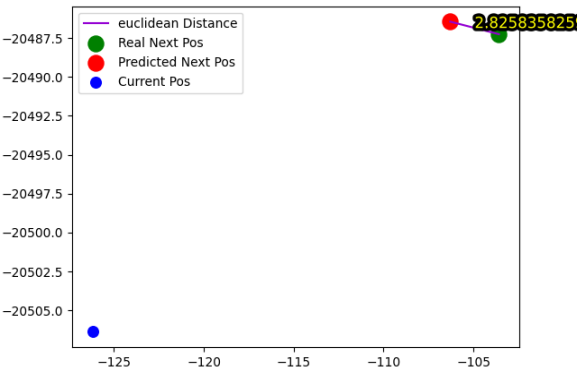

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
38.647095155771794

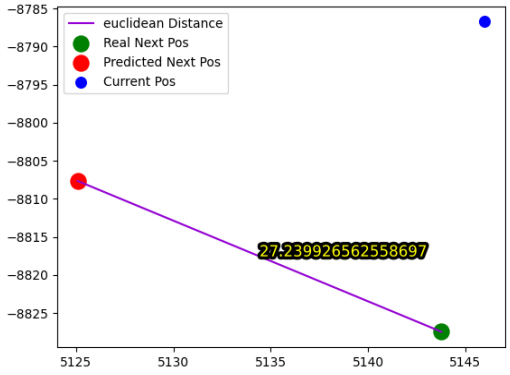

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
34.577781280272 32

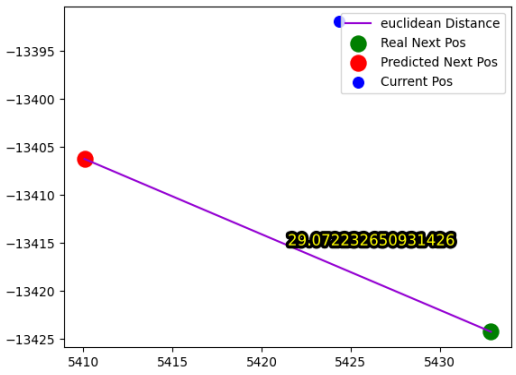

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
24.312

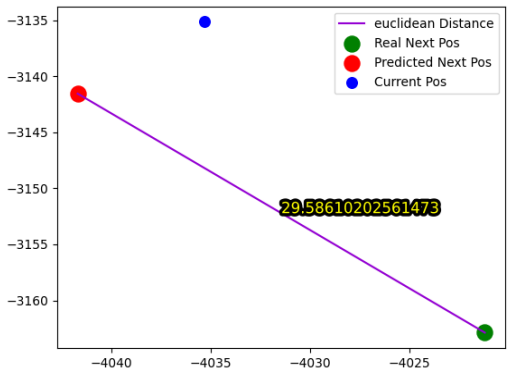

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos

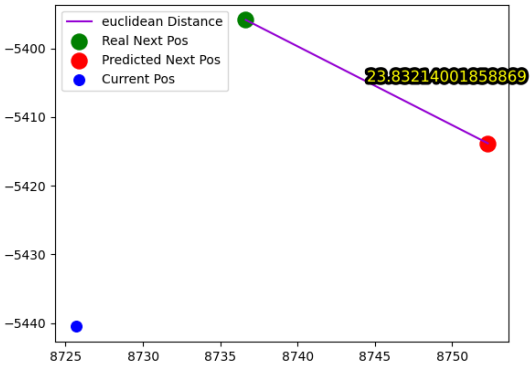

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
7.837170190170 16


euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
2.825835825

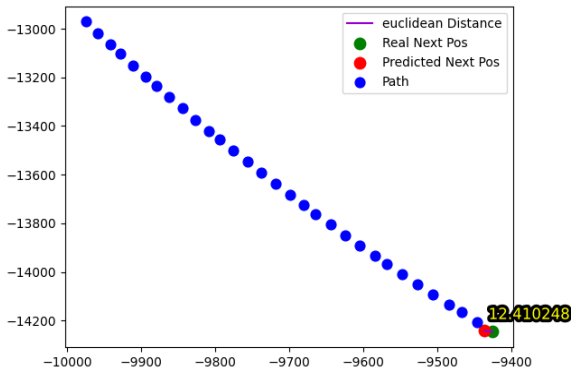

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
27.239926562558697

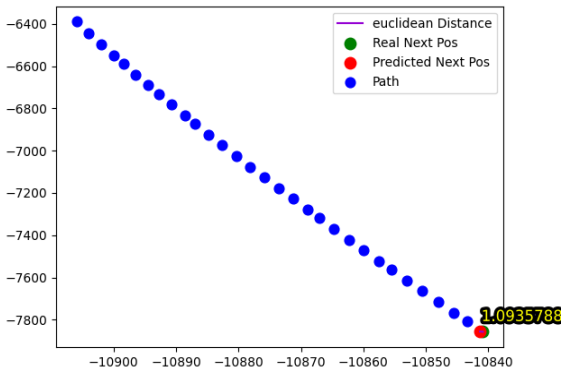

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
29.072232650931426

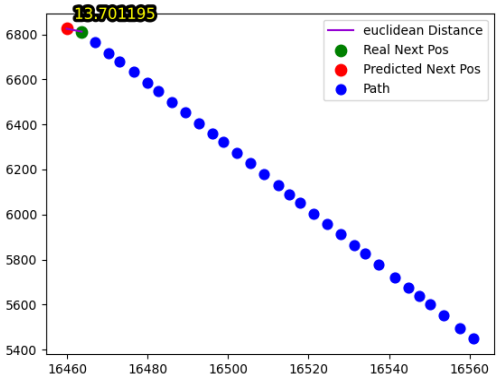

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
29.58610202561473

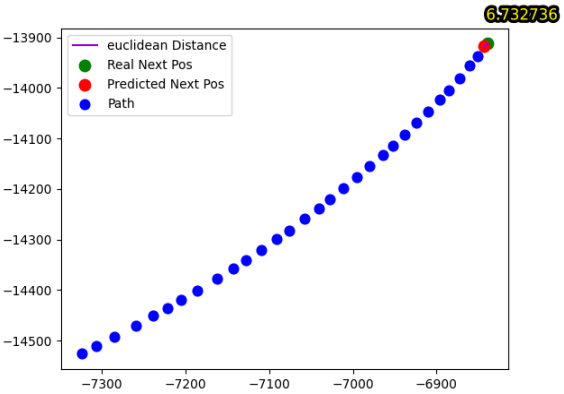

euclidean Distance
Real Next Pos
Predicted Next Pos
Current Pos
23.83214001858869

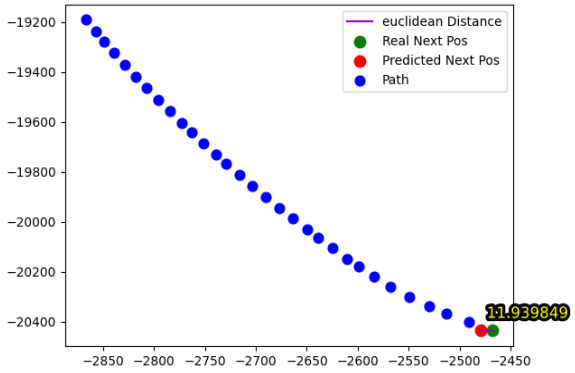
③ Transformer Encoder Base

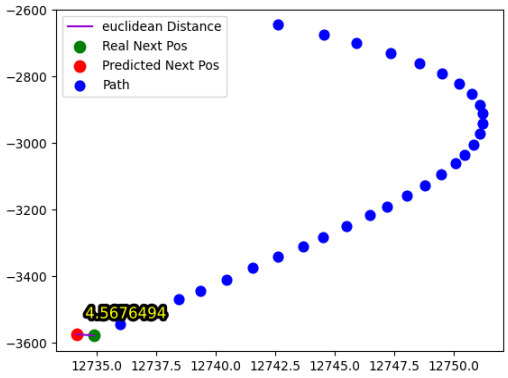


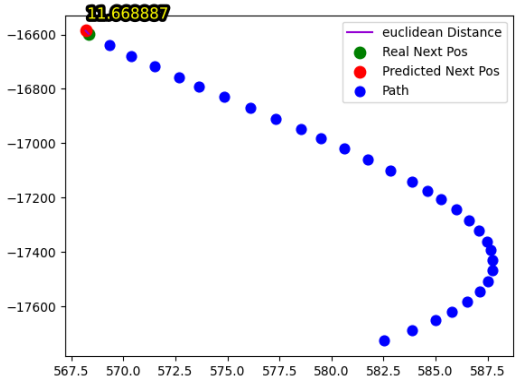


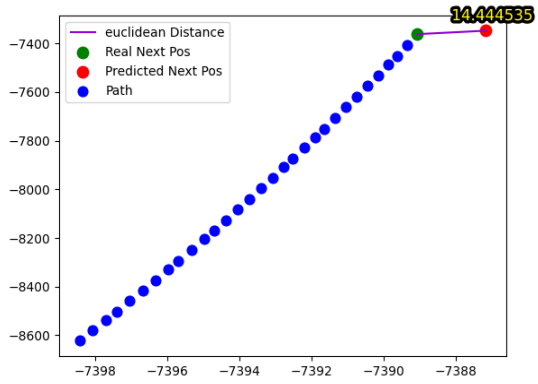


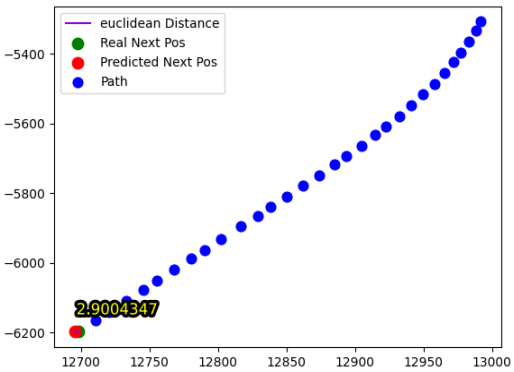


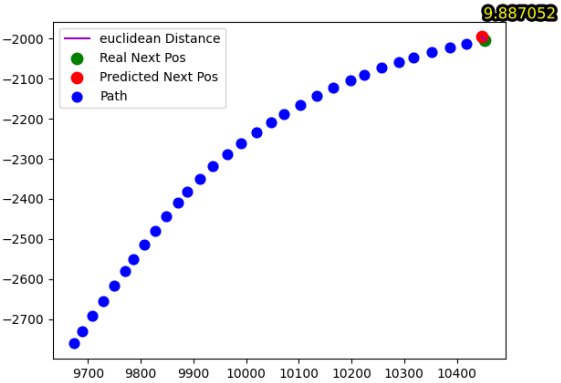


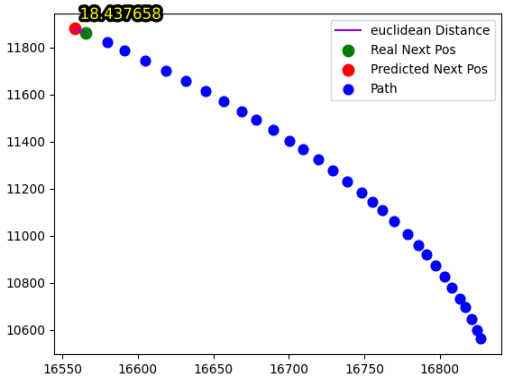


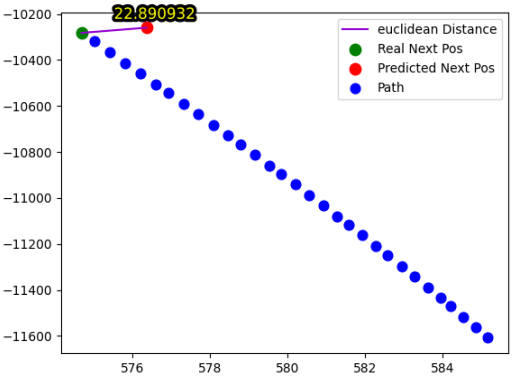

11.668887
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

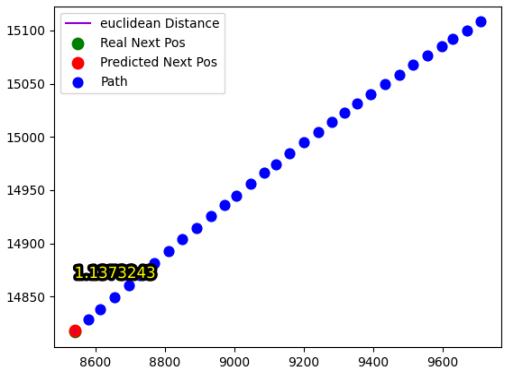

14.444535
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

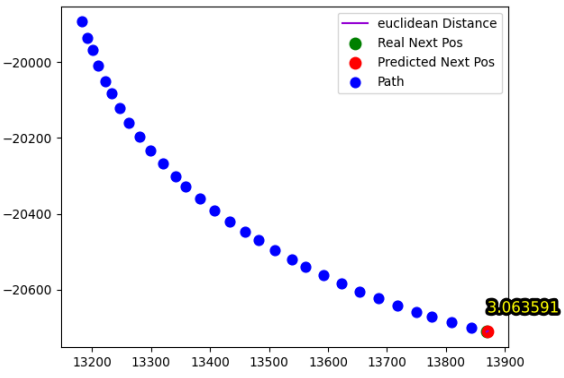

2.9004347
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

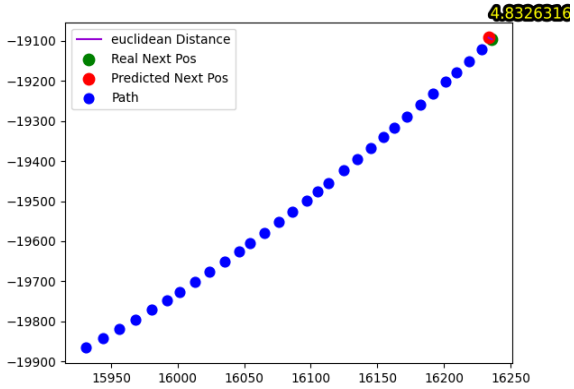

9.887052
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

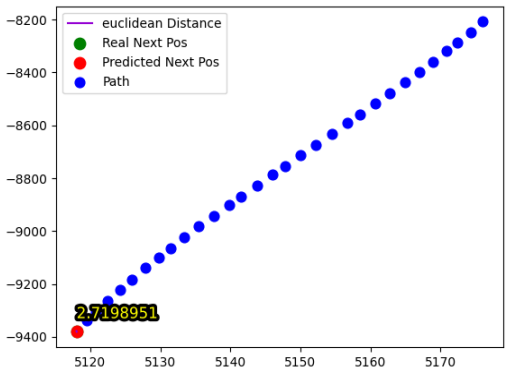

18.437658
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

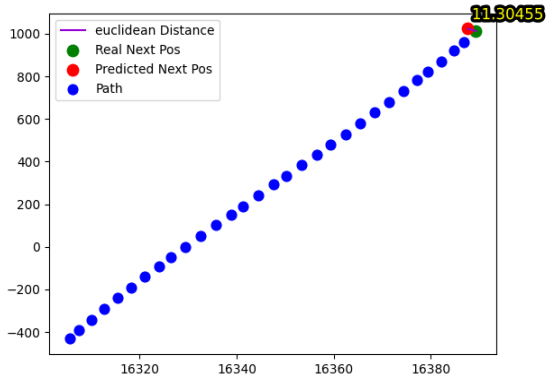

22.890932
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

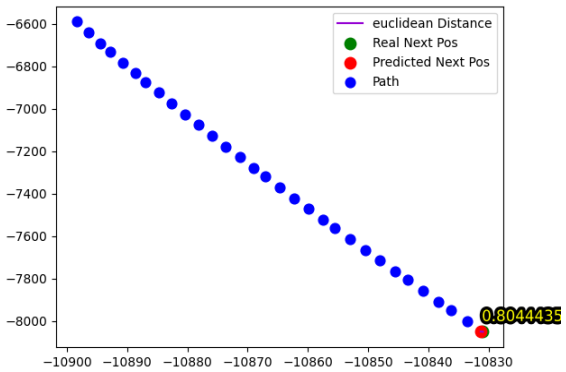

1.1373243
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

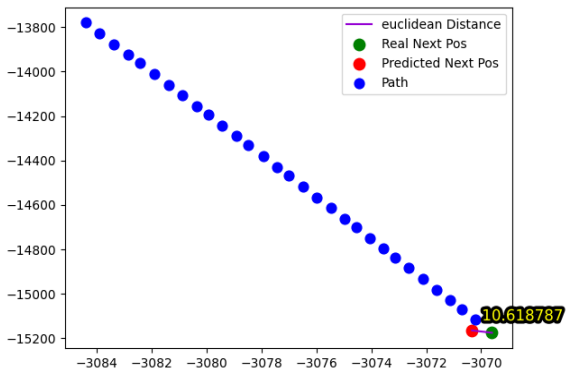

3.063591
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

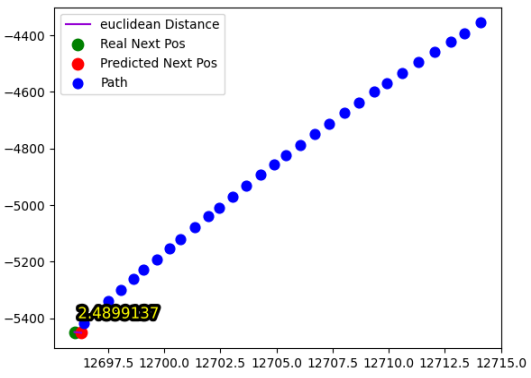

4.8326316
euclidean Distance
Real Next Pos
Predicted Next Pos
Path


2.7198951
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

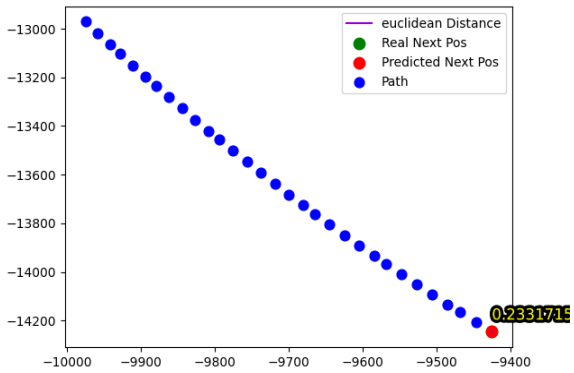

11.30455
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

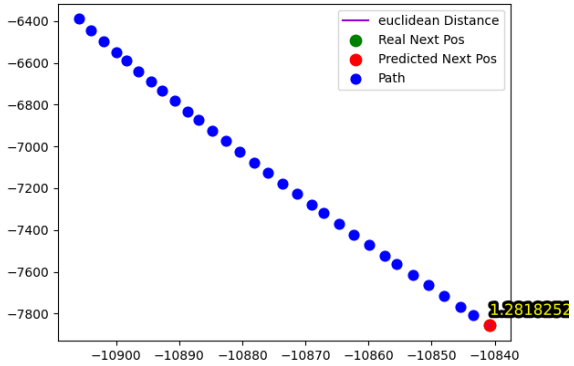

0.8044435
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

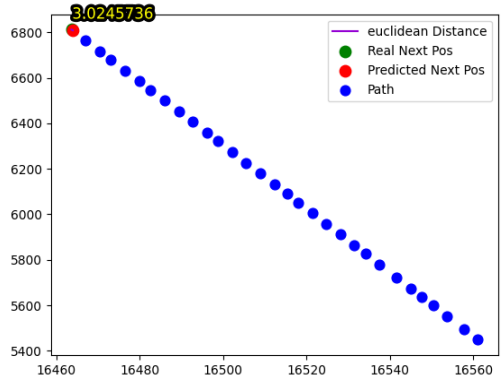

10.618787
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

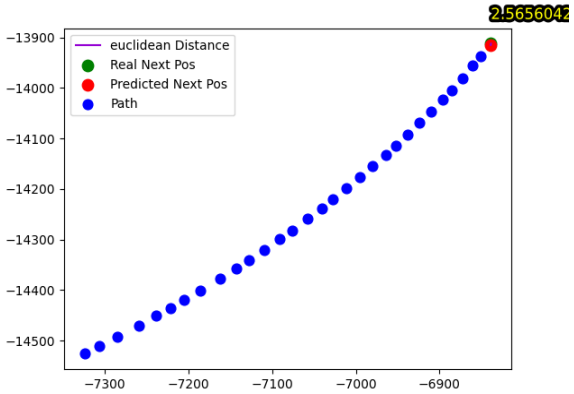

2.4899137
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

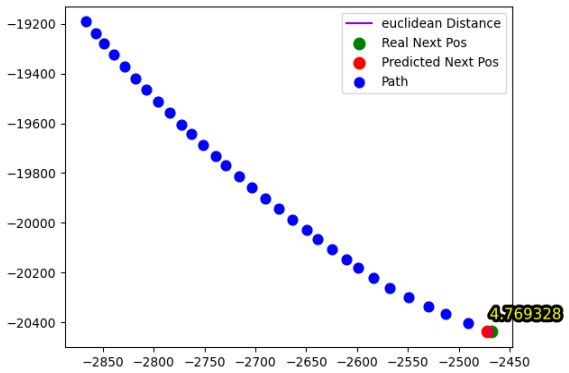
④ Transformer Encoder with CNN

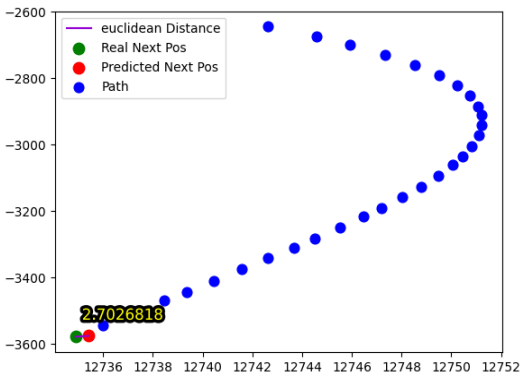


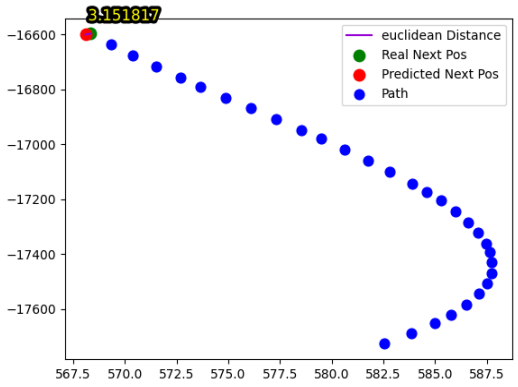


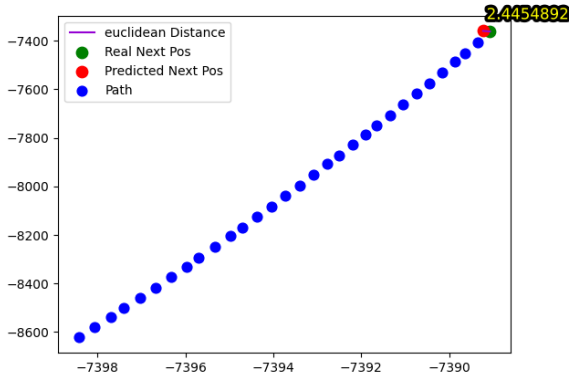


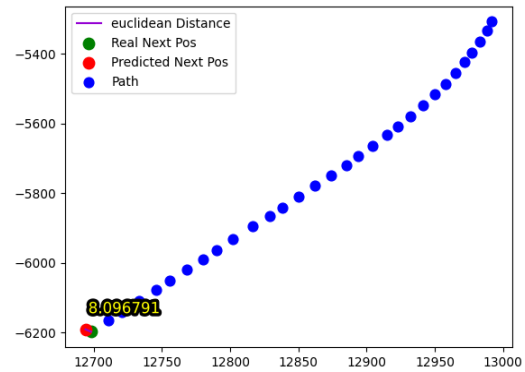


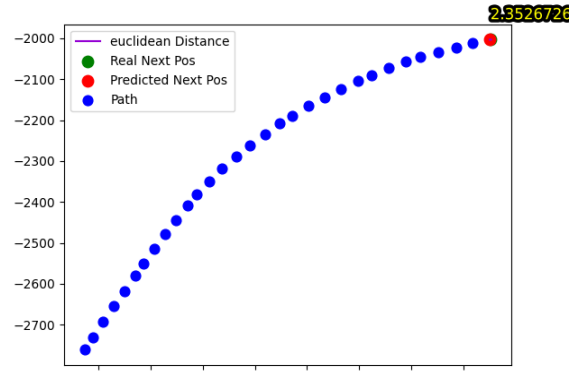


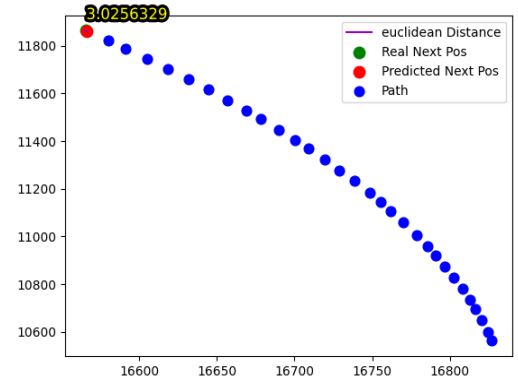


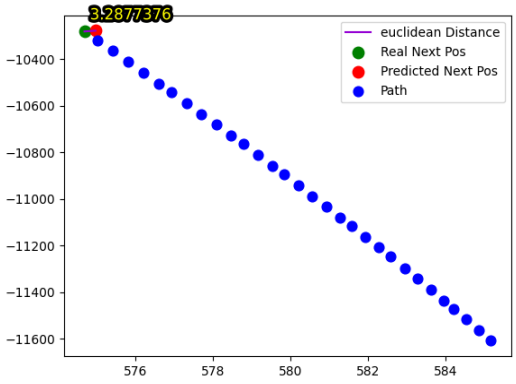

3.151817
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

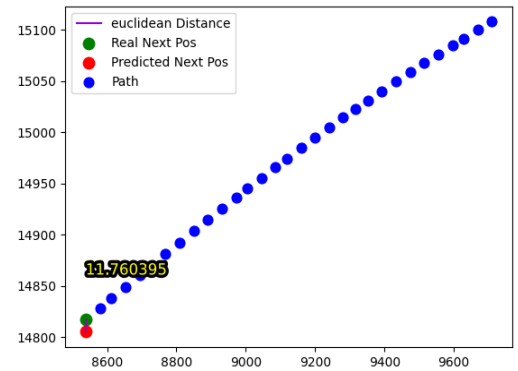

2.4454892
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

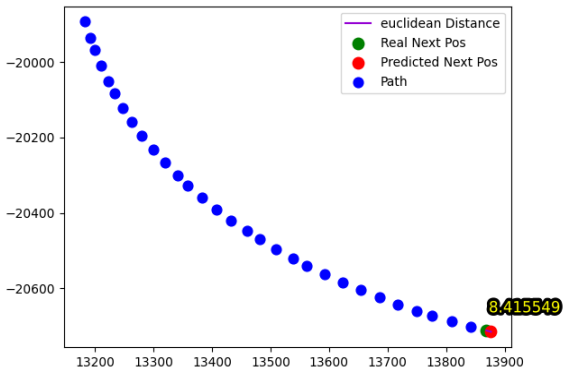

8.096791
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

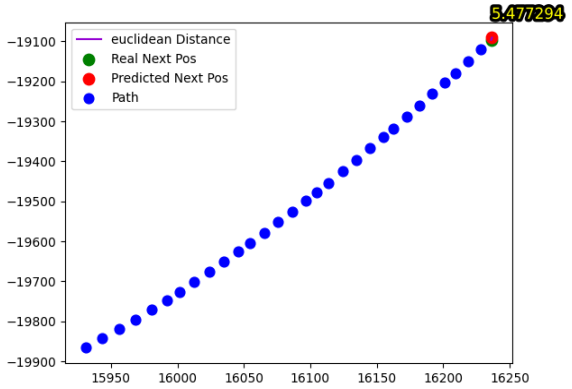

2.3526726
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

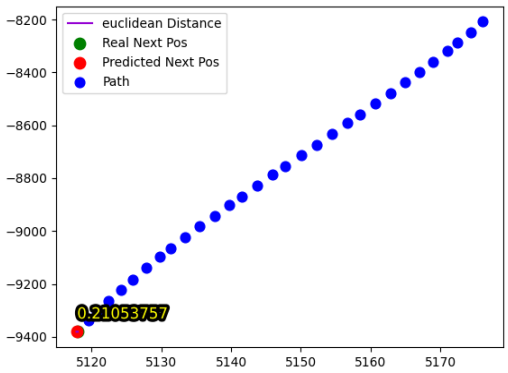

3.0256329
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

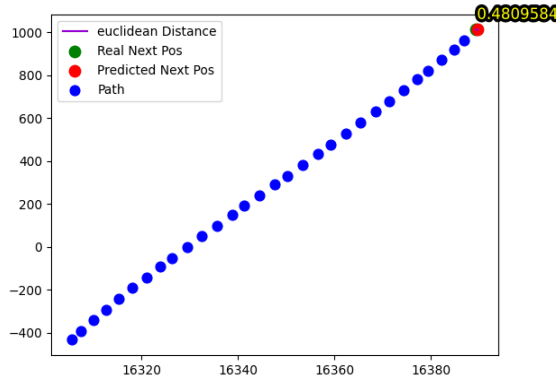

3.2877376
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

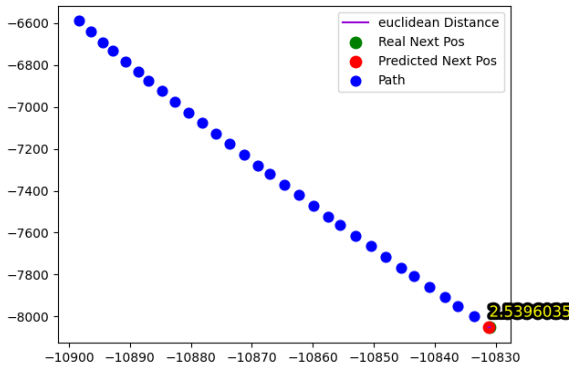

11.760395
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

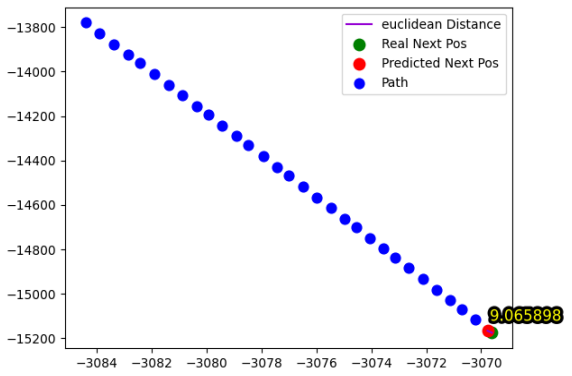

8.415549
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

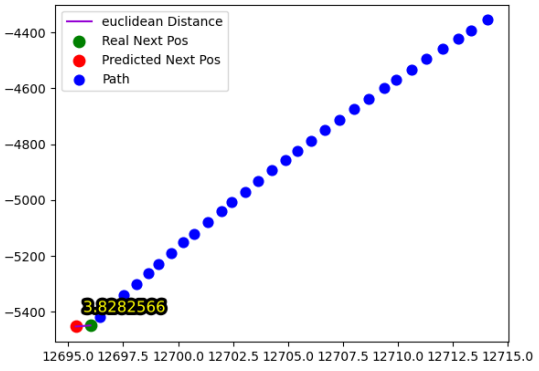

5.477294
euclidean Distance
Real Next Pos
Predicted Next Pos
Path


0.21053757
euclidean Distance
Real Next Pos
Predicted Next Pos
Path


0.4809584
euclidean Distance
Real Next Pos
Predicted Next Pos
Path


2.5396035
euclidean Distance
Real Next Pos
Predicted Next Pos
Path


9.065898
euclidean Distance
Real Next Pos
Predicted Next Pos
Path


3.8282566
euclidean Distance
Real Next Pos
Predicted Next Pos
Path

5. 결 론

I. 가장 좋은 성능의 모델은?

4 번 챕터의 테스트 그래프를 표로 정리하면 아래와 같습니다.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
LSTM Attention	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
DSA	1.570	9.316	1.951	4.796	2.993	7.456	0.106	1.012	5.564	3.507	7.797	7.561	6.632	8.460	6.786	6.407	10.014	3.984	12.365	8.987
Simple CNN	31.626	32.675	21.007	12.314	33.004	27.613	38.647	34.578	24.312	22.791	7.837	2.826	27.240	29.072	29.586	23.832	-	-	-	-
TF Encoder Base	12.410	1.094	13.701	6.733	11.940	4.568	11.669	14.445	2.900	9.887	18.438	22.890	1.137	3.063	4.833	2.720	11.305	0.804	10.619	2.490
TF Encoder with CNN	0.233	1.282	3.025	2.566	4.769	2.703	3.151	2.445	8.097	2.353	3.026	3.288	11.760	8.416	5.477	0.210	0.481	2.540	9.066	3.828

위 그래프와 표는 모든 test set 에 대해 기록한 것이 아닌, 20 개를 sampling 해서 만든 것이기에 4 번 챕터의 최종 성능 평가와 조금 다른 결과가 나올 수 있습니다.

표를 보시면 DSA 가 이긴 횟수는 4 번, Transformer Encoder Base 는 7 번, 그리고 Transformer Encoder with CNN 은 9 번임을 알 수 있습니다.

위 표에서 CNN 과 비교하지 않은 것은, CNN 의 경우 test set 이 다르기에 참고용으로만 적고 다른 모델들과 비교하지 않았습니다. 반면 다른 세 개의 모델의 경우 모두 동일한 test set 으로 평가를 진행했기에 정확한 성능 비교가 가능합니다. 따라서 이 챕터에서 선정하는 최고의 모델도 DSA, Transformer Encoder Base, Transformer Encoder with CNN, 이 세 가지 중 하나에서 정하겠습니다.

만약 제가 한 연구실의 연구원이고 논문을 작성하는 학생이라면, 최고의 모델로 Transformer Encoder with CNN 모델을 뽑을 것입니다. 가장 좋은 성능의 결과를 보였기 때문입니다.

하지만, 만약 이 실험이 사업적 또는 실용적 목적을 위한 것이라면 저는 Dual Stage Attention Base 모델을 최고의 모델로 뽑고 싶습니다. 그 이유로는 크게 두 가지가 있습니다.
먼저 안정성입니다. 저는 소프트웨어가 고객들에게 제공된다면, 가장 먼저 생각해야 할 것은 소프트웨어의 안정성이라고 생각합니다. 성능이 비슷하다면, 성능이 조금 안 좋더라도 안정적인 결과를 내는 소프트웨어를 선택해야 고객을 유지할 수 있을 것이기 때문입니다. Dual Stage Attention Base 모델은 train set 에서나, validation set 에서나, 그리고 test set 에서나 비슷한 성능과 결과를 보였습니다. 저는 이 사실이 꽤 인상적으로 다가왔습니다. 모델이 주어진 데이터가 아닌 real world 에 대해서도 좋은 성능을 보일 수 있다는 근거가 되기 때문입니다.
두 번째는 발전 가능성입니다. 4 번 챕터에서 한 번 말씀드렸듯이, Dual Stage Attention Base 모델은 중간에 loss 가 상승하는 구간 없이 지속해서 감소하기만 했습니다. 2000 epoch 를

넘어서도 train loss 나 validation loss 가 상승하지 않고 0.0001 씩이라도 내려간 모델은 위의 4 가지 모델 중 Dual Stage Attention Base 가 유일했습니다. 지금은 제한된 시간 때문에 훈련을 중단하였으나, 만약 훈련을 계속했다면 Transformer Encoder with CNN 모델의 성능을 따라잡을 수 있었을지도 모릅니다.

결론을 요약하겠습니다.
제가 연구원의 입장이라면 최고는 Transformer Encoder with CNN 모델로 뽑을 것입니다.
제가 사업가라는 입장이라면 최종 모델은 Dual Stage Attention 모델로 고를 것입니다.

II. 배울 수 있었던 점

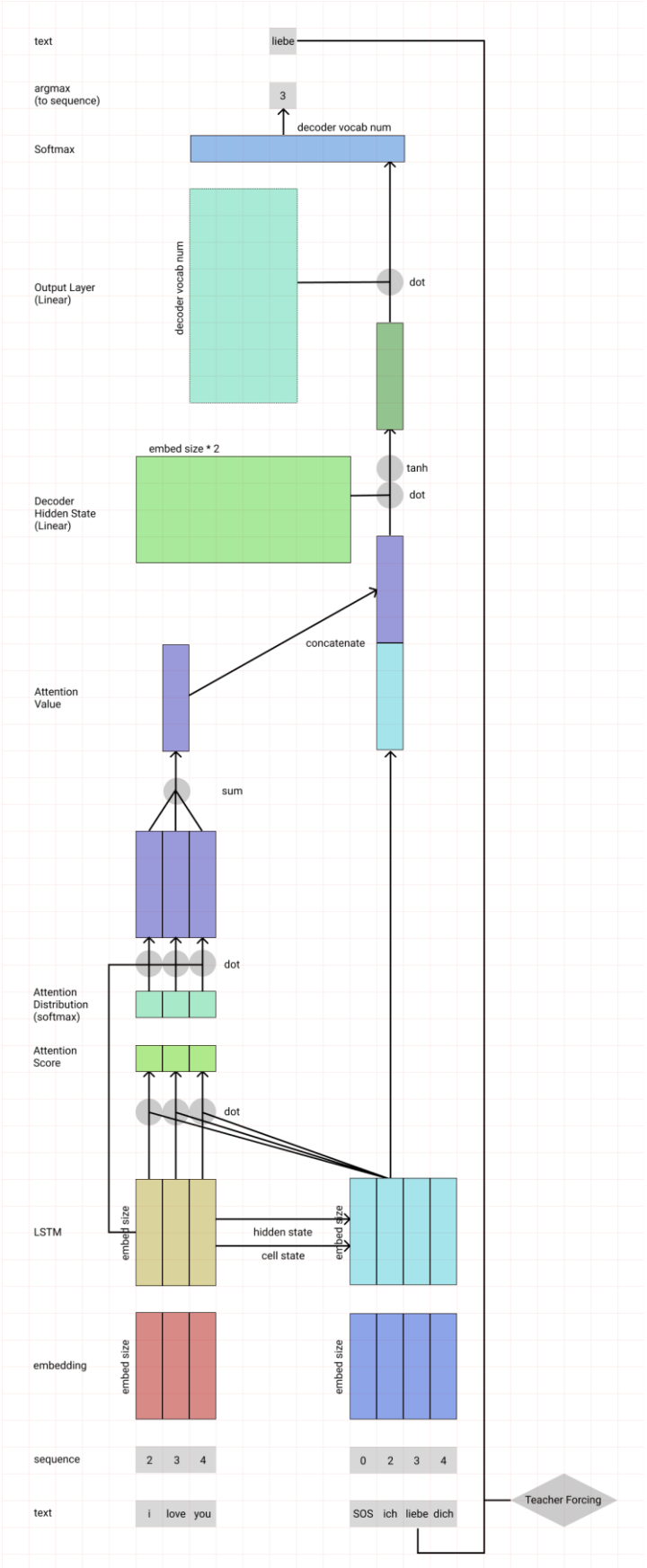
22 일부터 시작하여 현재 보고서를 완성한 28 일까지, 총 6 일이라는 시간 동안 많은 공부를 하였고 좋은 영감을 얻을 수 있었습니다. 제가 만들었던 모델의 문제점을 파악하고 새로운 모델을 탐색하는 시간이 있었고, 지난 시간 동안 풀지 못했던 문제에 대한 해결책의 실마리를 찾을 수도 있었습니다. 이 과제가 계기가 되어 그동안 핑계를 대며 멈춰 놨던 공부와 탐색의 시간이 흐르게 되었고 그것은 물 흐르듯이 많은 깨달음과 영감을 가져다주었습니다.

실험뿐만 아니라 이 보고서를 작성하는 것에도 많은 보람을 느낄 수 있었습니다. 제 실험을 다시 한번 돌아보면서 지나쳤던 문제점들을 다시 살펴볼 수 있었고 인공지능을 공부하기 시작한 2020 년부터 지금까지 거쳐온 시간을 느낄 수 있었습니다.

이번 과제를 수행할 기회를 주셔서 감사의 인사를 드리며, 보고서를 마칩니다.

- 서형진 올림

(첨부 사진)


text
liebe
argmax
(to sequence)
3
decoder vocab num
Softmax
decoder vocab num
Output Layer
(Linear)
dot
embed size * 2
tanh
dot
Decoder
Hidden State
(Linear)
concatenate
Attention
Value
sum
dot
Attention
Distribution
(softmax)
Attention
Score
dot
LSTM
embed size
hidden state
cell state
embed size
embedding
embed size
embed size
sequence
2 3 4
0 2 3 4
text
i love you
SOS ich liebe dich
Teacher Forcing