Time Series Forecasting

홍성준

Contents

- Study 1: **ARIMA**
- Experiment 1: **changing input datasets**
- Study 2 : **iTransformer**
- Experiment 2: **Testing with MLP**
- Next Goal

Assignment

- 8시간 뒤의 **object temperature**를 예측하는 assignment

- Dataset은 다음과 같은 구성으로 이루어져 있다.

serial	측정 위치의 Serial number ex)'20220428_000_41P0S1R1AX_1'
Temp	온도
Object_temp	측정하고자 하는 물체의 온도
Humi	습도
Pressure	압력
Date_time	측정 시간 year/month/date/hour/minute/second 정보

Problem recognition

- 기존의 forecasting에는 **lightgbm** 모델을 사용했다.

- **1시간 후**의 object_temp에 대한 온도는 **잘 예측**이 되었지만 **8시간 후**의 object_temp에 대한 온도 예측은 RMSE와 영하 온도에 대한 **예측이 잘 이루어지지 못했다**.

- 이에 대해 **다양한 방법론적 접근**을 통해 개선을 시도해본다.

Study 1 : ARIMA

- ARIMA = AR + I + MA

- AR(Autoregressive) model : 과거의 여러 시점들의 **variable**을 이용해서 t 시점의 **variable**을 예측
  - $y_t = \emptyset_0 + \emptyset_1 y_{t-1} + \emptyset_2 y_{t-2} + \cdots + \emptyset_p y_{t-p} + \varepsilon_t$

- MA(Moving Average) model: 과거의 여러 **error**들을 통해서 t 시점의 variable를 예측
  - $y_t = \theta_0 + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \cdots + \theta_q \varepsilon_{t-q} + \varepsilon_t$

ARIMA

- ARMA(Autoregressive Moving Average) : AR과 MA를 합쳐서 예측
  - $y_t = \emptyset_0 + \varepsilon_t + \emptyset_1 y_{t-1} + \cdots + \emptyset_p y_{t-p} + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q}$

- ARIMA를 적용하기 위해서는 data가 **stationary** 해야 한다.
  - **Differencing** 기법을 이용해서 data를 stationary 하게 적용

- 1st differencing : $Y_t = X_t - X_{t-1} = \nabla X_t$
- n^{st} differencing : $Y_t^{(n)} = Y_t^{(n-1)} - Y_{t-1}^{(n-1)} = \nabla^n X_t$

ARIMA

- **Variable of ARIMA** : (p,d,q)
  - p : AR model의 independent variable의 개수
  - d : differencing의 차수
  - q : MA model에서 사용할 오차의 개수

- $y_t = \emptyset_0 + \varepsilon_t + \emptyset_1 y_{t-1} + \cdots + \emptyset_p y_{t-p} + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q}$

ARIMA model processing

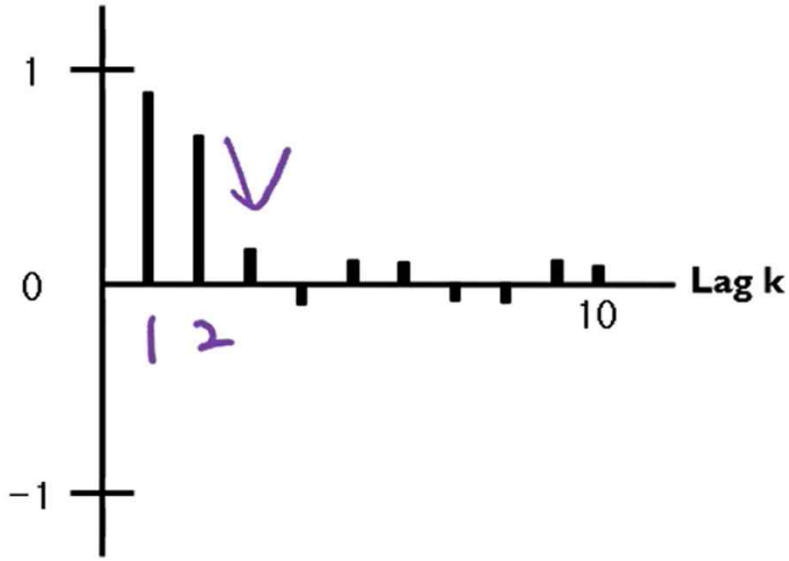
1. **Data preprocessing :** To make data **stationary**
2. **Identify Model**: **Test** demonstrative model
3. **Estimate Parameters**
4. **Diagnosis check**: IF not suitable, back to 2
5. **Use model to forecast**

Identification of model

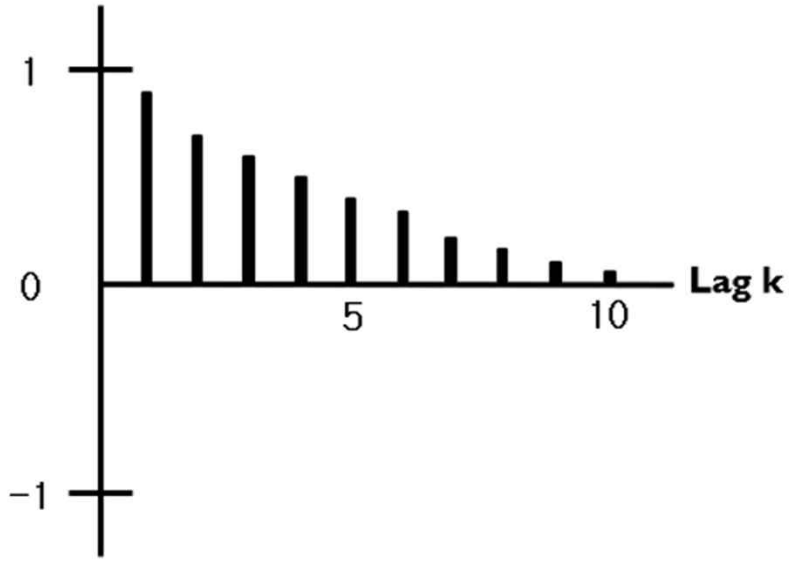
- Graphical method(**ACF, Partial ACF**)를 사용해서 MA(q), AR(p), ARMA(p,q) 중 어떤 모델을 사용할 지 판단한다.

Model	ACF	Partial ACF
MA(q)	Cut off after lag q	Die out
AR(p)	Die out	Cut off after lag p
ARMA(p,q)	Die out	Die out

- 위 표의 기준을 따라서 대략적으로 결정한 후 AIC, prediction error 등을 고려해서 더 좋은 모델을 예측한다.



(a) Cuts off after lag 2



(b) Damped exponential dying down

Experiment 1: changing the dataset

- 기존에는 5개의 이전 state를 가지고 이후를 예측했다.

- 8시간 뒤의 온도를 예측하기 위해서 **24시간의 time stamp**의 temp_obj를 사용해서 예측
- Temp_obj 이외에는 **8시간 전의 습도와 날짜정보**만 사용함

- **Lightgbm**을 통해서 예측

Testing with common sense

- Case 1: prediction with **past 1-day**
- Case 2: prediction with **difference of past 2 days**

Case	RMSE	MAE	STD	precision	recall	학습 시간
Case 1	4.06	2.88	4.06	0.71	0.70	0
Case 2	6.26	4.45	6.26	0.72	0.53	0

```python
def preprocessing2(df1): # df에는 preprocessing된 parquet이 들어올 것이다.
    df = df1.copy()

    num_lags = 24

    for i in range(8, num_lags+8):
        df[f'obj_before({i})'] = df['object_temp2'].shift(i)

    df['humi_before_32'] = df['humi'].shift(32)
    df['humi_before_20'] = df['humi'].shift(20)
    df['humi_before_8'] = df['humi'].shift(8)


    #object_temp = 1000인거 모두 drop
    df.dropna(inplace=True)
    #filtered_df = df[df['object_temp'] != 1000]

    print(len(df))
    #print(df.columns)
    new_df = df[['object_temp','year','month','date','hour','humi_before_8','humi_before_20','humi_before_32','obj_

    split_index = int(len(new_df)*0.7)

    df_train = new_df.iloc[:split_index]
    df_test = new_df.iloc[split_index:]

    X_train = df_train[['year','month','date','hour','humi_before_8','humi_before_20','humi_before_32','obj_before(
    y_train = df_train[['object_temp']]

    X_test = df_test[['year','month','date','hour','humi_before_8','humi_before_20','humi_before_32','obj_before(8)
    y_test = df_test[['object_temp']]

    return X_train, y_train, X_test, y_test
```

```python
def processing_in_time(df1):
    df = df1.copy()

    df = df[df['serial']=="20220428_000_41P0S1R1AX_1"]

    df['date_time'] = pd.to_datetime(df['date_time'])

    df.loc[:,'year'] = df['date_time'].dt.year
    df.loc[:,'month'] = df['date_time'].dt.month
    df.loc[:,'date'] = df['date_time'].dt.day
    df.loc[:,'hour'] = df['date_time'].dt.hour

    result_df = df.groupby(['year', 'month', 'date', 'hour']).first().reset_index()

    full_range = pd.date_range(start='2022-05-02', end='2024-06-24', freq='h')

    full_range_df = pd.DataFrame({
        'year' : full_range.year,
        'month': full_range.month,
        'date' : full_range.day,
        'hour': full_range.hour
    }).drop_duplicates()

    merged_df = pd.merge(full_range_df, result_df, on=['year', 'month', 'date', 'hour'])

    merged_df['humi'].fillna(method= 'ffill', inplace=True)
    merged_df['object_temp2'] = merged_df['object_temp']
    merged_df['object_temp2'].fillna(method='ffill', inplace=True)
    #merged_df['object_temp2'].fillna(1000)

    print(len(merged_df))
    return merged_df
```

```python
if __name__ == '__main__':
    df = pd.read_parquet('data/parquets/bme_20240625_preprocess.parquet')

    distinct_serial = df['serial'].unique()
    #print(distinct_serial)
    #args = _args()
    #args.train_months = 22
    #args.eval_months = 4

    #opt = load_config(args.config)
    print(df)
    df = processing_in_time(df)

    #X, y, X_val, y_val = preprocessing2(df)
    X, y, X_val, y_val = preprocessing_total(df)

    # Convert into list of (X, y) tuple pairs
    #valid_list = [(np.expand_dims(X_val[i], axis=0), np.expand_dims(y_val[i], axis=0))
    #              for i in range(len(y_val))]

    # callbacks = [lightgbm.early_stopping(stopping_rounds=100, verbose=True)]
    # params = {'n_estimators': 5000,
    #           'objective': 'regression',
    #           'metric': 'rmse',
    #           'learning_rate': 0.01,
    #           'subsample': 0.8,
    #           'colsample_bytree': 0.8,
    #           'max_depth': 7,
    #           'num_leaves': 31,
    #           'random_state': 42
    #           }

    start = time.time()
    booster = LGBMRegressor()
    booster.set_params(force_col_wise=True, num_leaves=31)
    booster.fit(X, y)  # , eval_set=valid_list, callbacks=callbacks
    train_time = time.time() - start

    start = time.time()
    pred = booster.predict(X_val)
    inf_time = time.time() - start
    y_val = y_val.values
    y_val = y_val.reshape(-1)
    print(metric_with_times(y_val, pred, train_time, inf_time))
```

Result

- 각각의 serial을 따로 학습시켜서 계산, 모든 serial을 합쳐서 계산해보았다.

1 = 20220428_000_41P0S1R1AX_1, 2 = 20221109_001_40P0S1R1A0_0, 3 = 20220826_000_40P0S0R0A0_0

Serial	RMSE	MAE	STD	Precision	recall	학습 시간
1	4.28	2.66	4.24	0.80	0.74	0.48
2	3.34	2.37	3.33	0.65	0.85	0.43
3	2.70	2.02	2.67	0.28	0.52	0.36
Total(8)	3.4	2.43	3.4	0.68	0.79	0.90

Conclusion

- **어떤 데이터셋을 학습시키는가**에 따라 차이가 많이 나는 것 같다.

- 모든 serial을 다 사용하여 training 할 때 대체로 좋은 결과값이 나오는 것 같다.

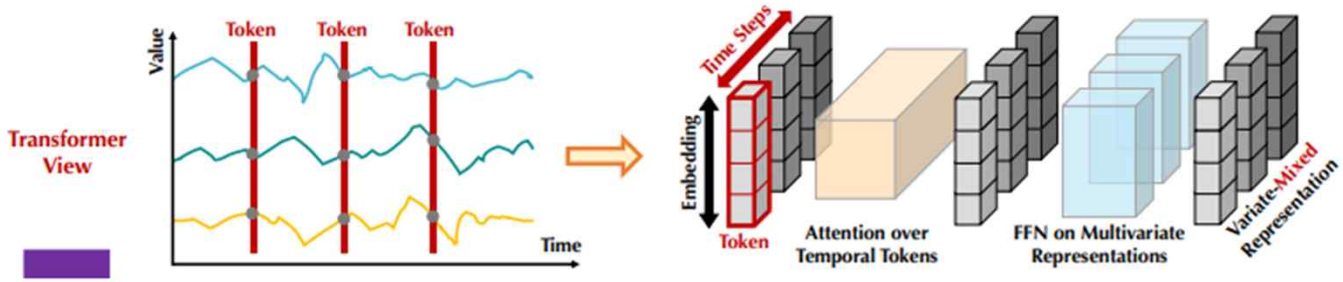
- RMSE와 예측 값의 **성능이 조금 아쉽다**.
  - 다른 모델이나 preprocessing 방법 등을 생각해서 성능을 향상시킨다.

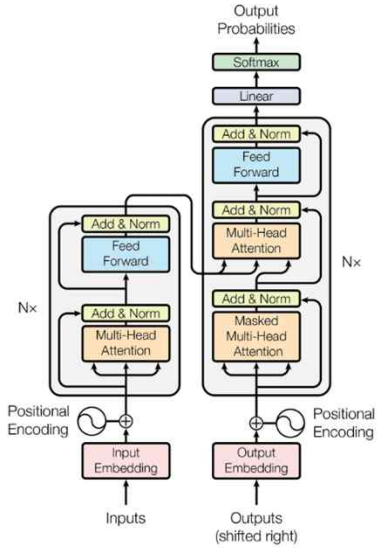
Study 2: iTransformer

- 기존의 transformer는 **Multi-level representation**과 **Pair-wise dependency**를 포착하는 능력으로 다양한 분야에서 사용되고 있다.

- 그 중 Time-series forecasting 에서는 **한 time-stamp**에 대해서 여러 변수들을 통합한 **embedding**을 만들고 각 temporal token에 대해서 **temporal dependency**를 포착하는 방법을 사용했다.

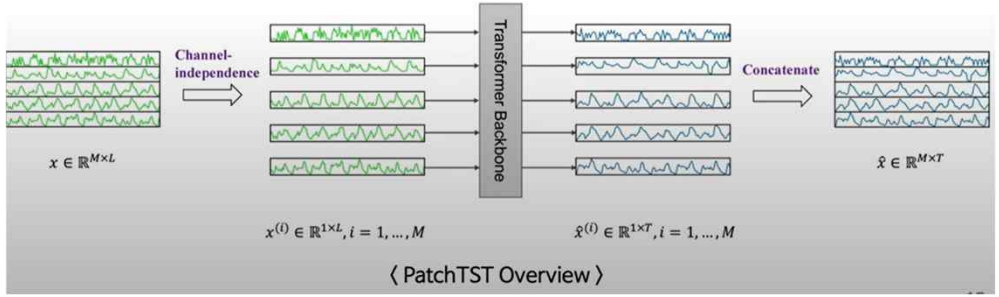
Classical Transformer

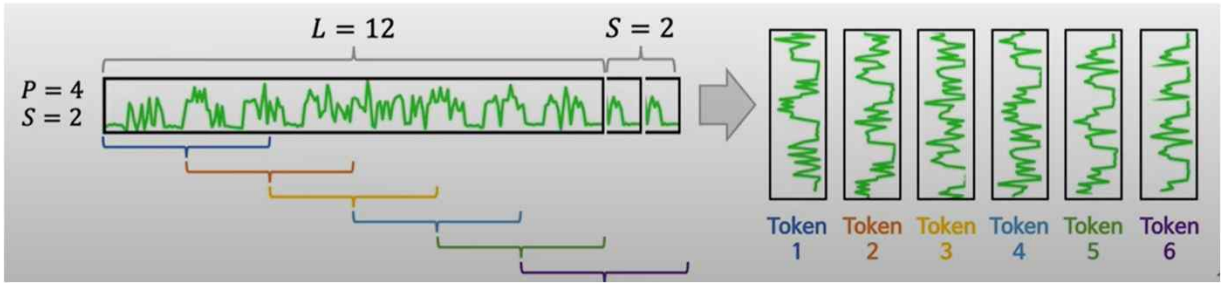




PatchTST

- Transformer에서 **성능과 효율성**에 대한 문제가 제기
- PatchTST 라는 논문에서 **channel independence**라는 method 가 제시되었다.





Problem

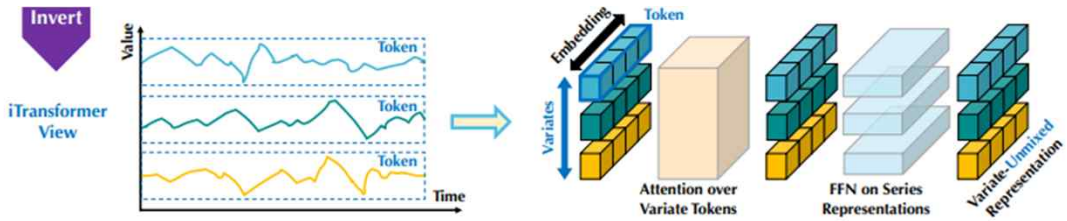
- 각 변수들은 다른 측정 방법을 사용한 변수들이기에 **embedding시 상관관계**가 고려되지 않을 수 있다.

- 각 token들은 **single time stamp**에 대한 정보만 포함하기 때문에 **local**하거나 **long-term**의 여러 **dependency**들을 학습하지 못할 수 있다.

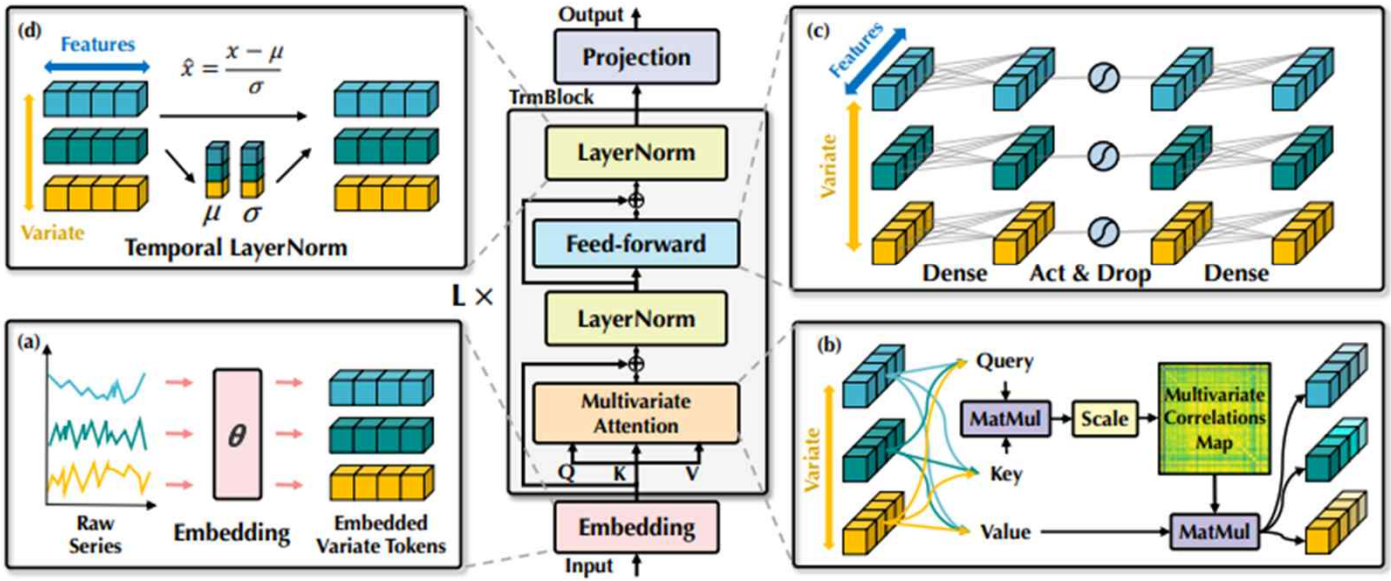
iTransformer의 개선 아이디어

- 각 time stamp 별로 embedding되던 기존의 방식에서 **Time series** 끼리 **embedding**을 구성하는 방식으로 변경

- 이에 맞춰서 model architecture도 변경



iTransformer 구조



iTransformer process

1. Embedding : 각각의 variate series들을 **MLP**를 이용해 각각의 **embedding** 제작

   -$h_n^0 = Em\,bedding \quad (X:n)$

2. Multivariate attention : 각각의 embedding을 W_Q, W_K, W_V 의 학습 가능한 matrix를 이용해 Q, K, V로 바꾼 이후 **attention**을 진행

   -기존 모델들과는 다른 방식, **변수 간의 관계**를 학습

iTransformer process

3. **LayerNorm** : **Convergence**와 **training stability**를 증가시키기 위해 제안

$$-LayerNorm \quad (H) = \{\frac{h_n - Mean(h_n)}{\sqrt{Var(h_n)}} | n = 1,2,3 \dots N\}$$

4. **Feed Forward Network**: 각 embedding 마다 순차적으로 적용, 여러 inverted block들이 쌓여서 미래에 관한 **representation**을 만들어 낸다.

Result

- 기존의 SOTA 모델들과 비교했을 때, 대부분의 데이터셋에서 더 **좋은 성능**을 보여줌
- 여러 Transformer 모델들에 대해서 기존의 구조 대신 **inverted 된 processing을** 적용 시 더 좋은 성능을 보여줌
- Channel independence와 비교해서 Inverted strategy가 성능과 효율성의 영역에서 좋은 결과값을 보여줌
- MLP를 각 series embedding에 적용하기 때문에 **look-back window**가 증가할 수록 **성능이 좋아지는 효과**가 생김

Experiment 2: Forecasting with MLP

- MLP를 이용해서 데이터셋을 학습시키고 예측
- 사용하는 data의 **series**, **look-back window**의 길이, epoch의 수 등에 따라서 학습과 예측을 시행해본다.
- 1~8 시간을 동시에 loss를 학습시킴

MLP model

```python
class MLP1days(nn.Module):
    def __init__(self, embeded_dim):
        super().__init__()
        hidden_layer1 = 20
        hidden_layer2 = 12
        output_layer = 8
        self.layers = nn.Sequential(
            nn.Linear(embeded_dim, hidden_layer1),
            nn.ReLU(),
            nn.Linear(hidden_layer1, hidden_layer2),
            nn.ReLU(),
            nn.Linear(hidden_layer2, output_layer)
        )

    def forward(self, x):
        return self.layers(x)
```

```python
class MLP2days(nn.Module):
    def __init__(self, embeded_dim):
        super().__init__()
        hidden_layer1 = 32
        hidden_layer2 = 16
        output_layer = 8
        self.layers = nn.Sequential(
            nn.Linear(embeded_dim, hidden_layer1),
            nn.ReLU(),
            nn.Linear(hidden_layer1, hidden_layer2),
            nn.ReLU(),
            nn.Linear(hidden_layer2, output_layer)
        )

    def forward(self, x):
        return self.layers(x)
```

MLP Process

```python
def preprocessing_mlp(df0, days, serial, serial_num):
    df = df0.copy()

    num_lags = 24*days

    df = df[df['serial']==serial]

    df['date_time'] = pd.to_datetime(df['date_time'])

    df.loc[:,'year'] = df['date_time'].dt.year
    df.loc[:,'month'] = df['date_time'].dt.month
    df.loc[:,'date'] = df['date_time'].dt.day
    df.loc[:,'hour'] = df['date_time'].dt.hour

    result_df = df.groupby(['year', 'month', 'date', 'hour']).first().reset_index()

    full_range = pd.date_range(start='2022-05-02', end='2024-06-24', freq='h')

    full_range_df = pd.DataFrame({
        'year' : full_range.year,
        'month': full_range.month,
        'date' : full_range.day,
        'hour': full_range.hour
    }).drop_duplicates()

    merged_df = pd.merge(full_range_df, result_df, on=['year', 'month', 'date', 'hour'])

    merged_df['humi'] = merged_df['humi'].ffill()

    merged_df['serialnum'] = serial_num
    #merged_df['humi'].fillna(method= 'ffill', inplace=True)

    df_X_cols = ['year']+['month']+['date']+['hour']+['serialnum']

    df_Y_cols = ['object_temp']

    for i in range(8, num_lags+8):
        merged_df[f'obj_before({i})'] = merged_df['object_temp'].shift(i)
        df_X_cols = df_X_cols + [f'obj_before({i})']

    merged_df['humi_before_8'] = merged_df['humi'].shift(8)
    df_X_cols = df_X_cols + ['humi_before_8']
```

```python
trainsets = TensorData(X, y)
trainloader = DataLoader(trainsets, batch_size=32, shuffle=False)

testsets = TensorData(X_val, y_val)
testloader = DataLoader(testsets, batch_size=32, shuffle=False)

loss_ = []

start = time.time()

for epoch in range(200):
    running_loss = 0.0

    for i, data in enumerate(trainloader, 0):
        inputs, values = data

        optimizer.zero_grad()

        outputs = model(inputs)
        #print("outputs : ", outputs)
        #print("values :", values)
        loss = criterion(outputs, values)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    loss_.append(running_loss/len(trainloader))

train_time = time.time()-start

predictions = torch.tensor([], dtype=torch.float)
actual = torch.tensor([], dtype=torch.float)

start = time.time()
with torch.no_grad():
    model.eval()

    for data in testloader:
        inputs, values = data
        outputs = model(inputs)

        predictions = torch.cat((predictions, outputs), 0)
        actual = torch.cat((actual, values), 0)
inf_time = time.time() - start
```

Result

- Using ‘20220428_000_41P0S1R1AX_1’ series

Condition	RMSE	MAE	STD	precision	recall	학습 시간
1day, 50epoch	4.08	3.07	3.79	0.53	0.89	27
1day, 100epoch	3.91	2.99	3.44	0.46	0.95	54
1day, 200epoch	4.09	3.18	3.44	0.56	0.91	110
2day, 50 epoch	4.74	3.48	4.42	0.48	0.93	27.5
2day, 100epoch	4.2	3.35	3.72	0.35	0.97	55

Result

- Using ‘20221109_001_40P0S1R1A0_0’ series

- 하지만 결과로부터 유의미한 특징을 찾을 수 없음
- 초기 weight들이 random으로 정해지는데 이에 따른 결과값 변동이 더 영향을 많이 미치는 것 같다

Condition	RMSE	MAE	STD	precision	recall	학습시간
1 day, 50 epoch	6.54	4.07	4.39	0.75	0.78	19.42
1 day, 100 epoch	4.68	3.36	3.81	0.78	0.91	38
1 day, 200 epoch	4.28	2.98	3.37	0.74	0.90	77.88
2 day, 100 epoch	4.68	3.31	3.89	0.85	0.81	40.9

Result

- Using 8 series

- **Increasing datasets** can improve the performance

Condition	RMSE	MAE	STD	precision	recall	학습 시간
1 day, 100 epoch	3.36	2.40	3.15	0.85	0.74	288
2 day, 100epoch	3.37	2.46	3.09	0.87	0.77	317
2 day, 200epoch	3.29	2.34	3.12	0.83	0.84	697

Result

- 8시간 뒤의 예측에 대한 결과값만 평가에 반영

- For ‘20220428_000_41P0S1R1AX_1’ series

Condition	RMSE	MAE	STD	precision	Recall	학습 시간
50 epoch, 2 days	6.50	5.35	6.35	0.30	0.89	26
200 epoch, 1days	4.8	3.65	4.32	0.57	0.87	123
200 epoch, 2 days	5.16	4.14	4.7	0.3	0.91	1.9

Result

- 여러 series를 합쳐서 결과 생성
- 세 번째 결과는 두 번째 결과에 각 **series number column** 추가
- 200epoch, 2 days의 결과를 사용했다

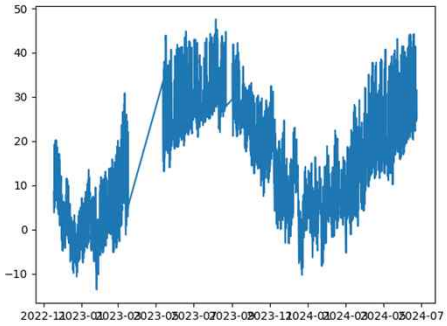
Condition	RMSE	MAE	STD	precision	recall	학습 시간
Using 8 series	4.06	3.03	3.8	0.77	0.75	657
Using 13 series	4.38	3.35	4.36	0.47	0.92	955
Using 13 series	3.92	2.94	3.86	0.64	0.84	1018

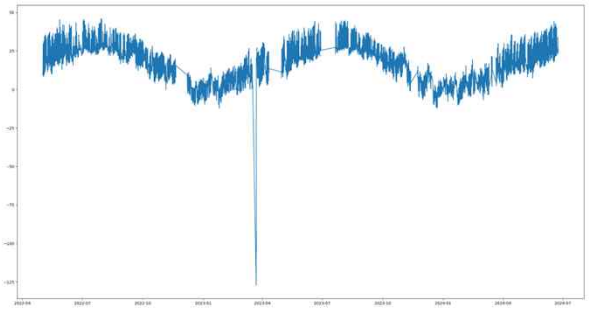
Conclusion

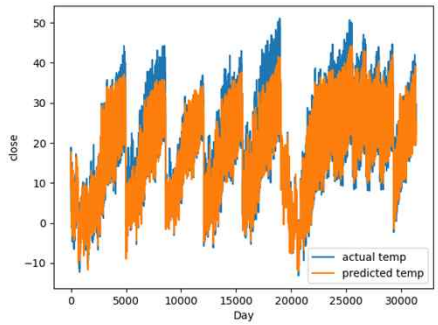
- MLP를 이용했을 때는 큰 성능의 향상을 얻지 못함
- 한 series만 사용했을 때보다 여러 series의 dataset들을 합쳐서 사용할 때 좋은 결과가 나옴
- 90% 이상의 영하 예측 성능을 위해서는 **다른 모델**과 **새로운 영하 예측에 대한 기준**을 정해야 함

Problem of dataset?

- 2년 정도의 기간
- 겨울 중에도 하루종일 영하인 경우는 별로 없고 영상과 영하를 반복한다.







Next Goal

- Using **K-NN** for classification
- Adjust **normalization method** to improve model
- **Redefine** precision and recall
- Find **new model** to improve prediction