태양광 발전량 데이터 분석과 TFT 모델 활용 예측모델 개발

대전대신고등학교

20628 장규중

ABSTRAC: 정확한 pv 시스템 발전량 예측을 위해 데이터 분석, TFT 이론 설명 및 예측 모델 구현을 한다. 발전량에 큰 영향을 미치는 풍속, 방사능 수치, 햇빛, 기온, 상대 공기 습도, 기압들의 상관계수 분석을 한다. 25 도 이상의 값이 적은 데이터를 통해 학습하여 생기는 일반화 오류문제 해결법에 대해 고안한다. 데이터와 시간과의 관계, 시간에 대해 다른 특징을 가지는 그룹을 데이터를 분석을 통해 찾고 시간에 대해 변하지 않는 공변량을 설정하고 구하여 이러한 정보들이 적용하여 48 시간의 데이터로 24 시간 동안의 발전량을 예측하는 모델을 구현한다. 소수점 단위의 오차 값이 나왔지만 표준화되지 않은 상태에서는 만족할만한 값은 아니다. 모델 개선을 위해 결과값이 음수가 될 수 없다는 조건을 효과적으로 적용한다면 더 낮은 오차 값을 가질 수 있을 것으로 예상한다.

서론 (Introduction)

전기에너지는 현재 생활에 필수적인 요소로 자리잡았다. 전기에너지에 대한 수요는 급증하지만 기존 연료의 고갈과 사용에 의한 환경오염은 재생 에너지의 필요성을 증가시킨다. PV 태양 전지는 상호 연결되어 PV 모듈을 형성하여 태양 광선을 포착하고 태양열을 전기로 변환한다. 전력 소비 균형을 적절하게 유지하고 여분의 에너지 비축, 시스템 문제 확인을 위해서는 정확한 예측이 필요하지만 이는 기상 환경의 영향을 많이 받는 발전량은 예측하기 어렵다. 정확한 pv 발전량 예측을 위해 데이터 분석과 TFT 모델 이론 설명 및 구현, 추가적인 모델 개선 방법을 제공하고자 한다.

선형에 대한 설명

$$f(x) = 2x \qquad f(x) = 2x + 1$$

$$f(a+b) = f(a) + f(b) \qquad f(a+b) \neq f(a) + f(b)$$

$$f(ka) = kf(a) \qquad f(ka) \neq kf(a)$$

a 와 b 를 더하고 함수를 적용한 것과 a 와 b 에 함수를 적용하고 더한 것이 같은 중첩성과 상수 k 를 곱하고 적용한 것과 적용한 후 곱한 값이 같은 동질성이 성립하면 선형성을 가진다고 한다. f(x)= 2x+1 이라는 함수는 중첩성, 동질성이 없기 때문에 선형성을 띠지 않아 선형변환을 나타내는 행렬 간의 내적이

$$f(x) = 3x^2 + 2x + 3$$

$$g(x) = K(3x^2 + 2x + 3)$$
$$g'(x) = kf'(x)$$

불가능하다.

$$g(x) = 3x^5 + 2x^4 + 5x^3$$
$$k(x) = 2x + 4$$
$$f(x) = g(x) + k(x)$$
$$f'(x) = g'(x) + k'(x)$$

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{bmatrix} \begin{Bmatrix} 4 \\ 2 \\ 3 \\ 5 \\ 2 \\ 3 \end{Bmatrix} \Longrightarrow \begin{Bmatrix} 2 \\ 6 \\ 15 \\ 8 \\ 15 \end{Bmatrix}$$

$$f(x) = 3x^5 + 2x^4 + 5x^3 + 3x^2 + 2x + 4 \Longrightarrow f'(x) = 15x^4 + 8x^3 + 15x^2 + 6x + 2$$

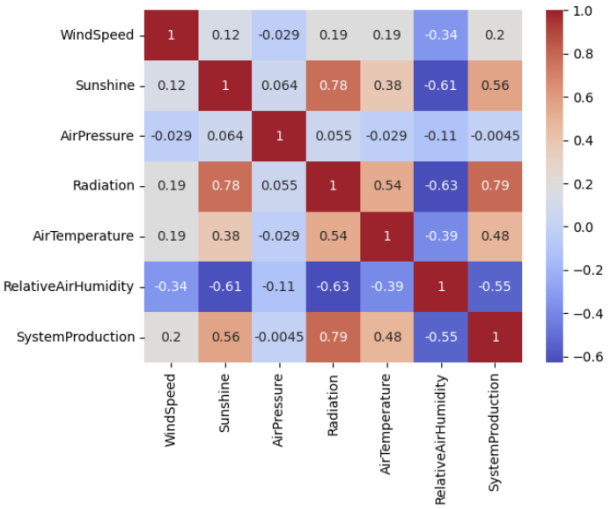
<Fig 1. 행렬 연산으로의 미분 표현 >

<Fig1>은 미분연산이 중첩성과 동질성을 가져 f(x)와 f′(x)의 관계가 선형성을 가지는 것을 보여준다. <Fig4>와 같이 입력과 출력의 관계가 선형성을 가지는 미분연산은 행렬의 내적으로 표현이 가능하다. 이때 f(x)=2x+1 을 행렬의 내적에 사용할 수 없는 것이 아니다

<Fig 2. 선형변환에서의 역할 표현>

<fig2>을 보면, 도구 재료 결과'로 표현했을 때 도구로 사용하지 못할 뿐 '재료, 결과'는 가능하다. 즉 선형이라는 것은 입력과 출력의 관계를 설명하는 것이고 결과가 2 차함수 이상의 곡선이더라도 문제를 푼 방식 즉 입력과 결과의 관계가 선형인 선형변환만이 이루어졌다면 선형회귀이다.

EDA(Exploratory Data Analysis)

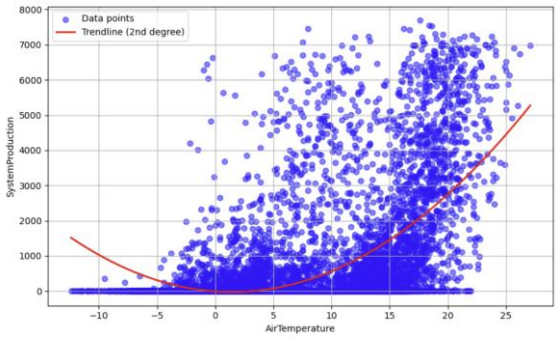


<Fig 3. 상관관계 히트 맵>

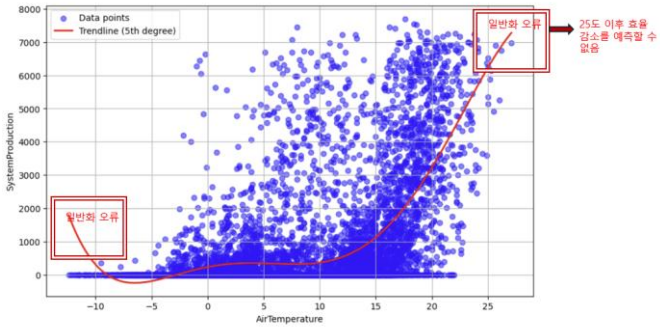
1) 방사능: 방사능 값이 높을수록 패널에 도달하는 에너지가 많아져 발전량도 증가한다.
2) 햇빛: 태양광 패널은 태양의 빛을 흡수하여 에너지를 생산한다.

3) 상대 습도: 높은 습도는 태양광의 도달하는 양을 줄이고, 패널에 영향을 주어 효율을 감소시킬 수 있어 음의 상관관계를 보이는 것을 확인할 수 있다.

4)기온: 기온이 오를수록 생산량이 증가하지만 25 도 이상으로 너무 높으면 오히려 효율이 감소할 수 있다. (아쉽게도 사용한 데이터는 기온 값이 25 도를 많이 벗어나지 않는다.)



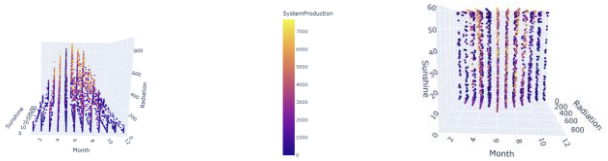
<Fig 4. 기온-생산량 2 차함수 선형회귀 추세선>



<Fig 5. 기온-생산량 5 차함수 선형회귀 추세선 >

<fig 4,5>을 통해 25 도 이상의 데이터가 현저히 적은 상황에서는 25 도 이후 추세가 감소할 것이라는 사실을 알 수 없고 이는 일반화 오류가 발생하여 25 도 이상의 상황에서 잘못된 추론을 하게 된다.

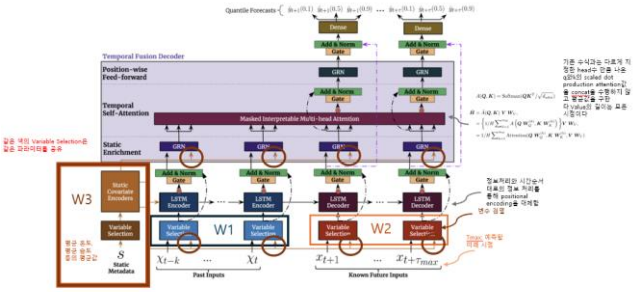
5)바람속도: 적절한 바람은 패널의 온도를 낮추어 효율을 높일 수 있다.



<Fig 8. 달, 햇빛, 방사능 수치 분포 시각화>

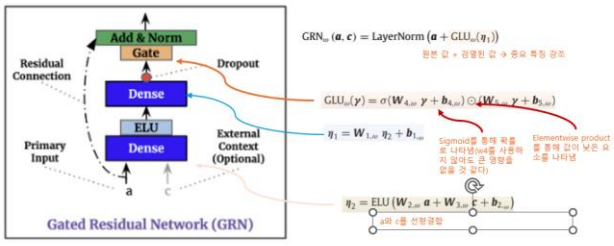
<fig 6>을 통해 여름에는 생산량이 높고 넓은 범위를 가지고, 햇빛은 다양한 값을 가지고, 겨울에는 낮고 좁은 범위의 값을 가지고, 햇빛은 부분적인 값을 가지는 것을 확인할 수 있다. 이는 계절에 따라 특성이 구분됨을 확인할 수 있고, group id 를 지정함으로써 계절에 따른 데이터 특성을 학습할 수 있다.

TFT(Temporal Fusion Transformer)



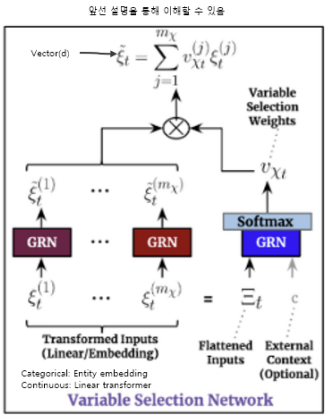
<Fig 8. TFT 아키텍처>

같은 색깔의 변수 선택 부분은 같은 파라미터를 공유한다고 가정한다. 인코딩한 공변량 정보는 매 GRN 구조, LSTM 구조, VS 구조에 사용된다.



<Fig 9. GRN 아키텍처>

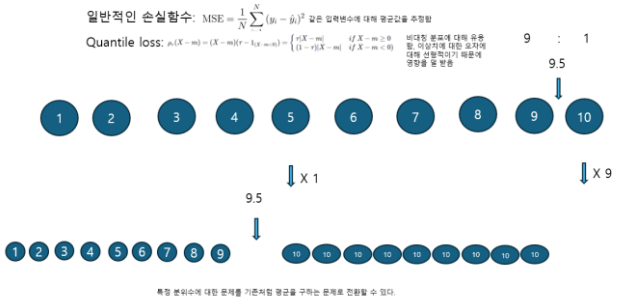
<fig 8>에서 ELU 를 활용하는 부분에선 입력 변수와 공변량의 선형변환의 합으로 표현되고, ELU 를 적용한 출력 값을 다시 선형변환한 n1 을 확률에 따라 덜 중요한 값을 약화시키는 GLU 의 입력 변수로 사용된다. n1 의 선형변환을 확률 값으로 변환하여 또 다른 n1 의 선형 변환된 값과 요소별 곱을 수행하여 중요도에 따라 값을 조정하고 이를 기존 확률변수와 더하여 중요 요소 값을 강조한다.



<Fig 10: VSN 아키텍처>

<fig 9>카테고리 값은 Entity embedding 을 연속적인 값은 Linear transformer 을 거친 각각의 시점의 입력 값은 GRN 을 거친다. 입력 값을 flatting 한 값과 공변량을 GRN 을 거치고 SoftMax 를 통해 각각 확률로 나타내게 된다. 이를 Vxt 의 각각의 시점과 시그마를 통해 수행하여 각 시점의 변수를 선택하게 된다.

Quantile loss



일반적인 손실함수는 같은 확률변수에 대해서 평균값을 구하는 문제를 풀게 된다. 하지만 데이터의 분포가 비대칭이라면 학습 데이터를 떠나 일반화를 생각할 때 밀집된 부분의 값을 구하는 방식이 더 좋은 성능을 가지게 된다. MSE 는 비대칭을 고려하지 않고 평균만 구하지만 특정 분위수에 집중한다면 더욱 좋은 일반화 성능을 가지는 모델을 만들 수 있다. 만약 1~10 까지의 정수 값이 존재하고 9.5 지점(9:1)의 값을 구하는 문제를 푼다면, 왼쪽에는 9 를 오른쪽에는 1 을 곱하여 9.5 지점을 2 분위수 위치로 옮겨 기존의 문제 변환하게 된다. 또한 기존처럼 오차에 대해 제곱을 하지 않고 선형적인 특성으로 인해 이상치에 대한 영향을 덜 받게 된다.

구현코드

미래의 24 시간 동안의 햇빛, 풍속, 방사능 수치, 기온, 상대 공기 습도, 기압 값을 예측하여 이를 생산량 예측 모델에 사용한다.

```
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(df[features])
# feature별로 최대,소 계산 및 스케일러 적용


class Dataset(Dataset):
    def __init__(self, data, seq_length, pred_length):
        self.data = torch.FloatTensor(data)
        self.seq_length = seq_length
        self.pred_length = pred_length

    def __len__(self):
        return len(self.data) - self.seq_length - self.pred_length + 1

    def __getitem__(self, idx):
        x = self.data[idx:idx+self.seq_length]
        #입력 데이터(seq_length)
        y = self.data[idx+self.seq_length:idx+self.seq_length+self.pred_length]
        #정답 값(pred_length)
        return x, y
```

데이터에 최솟값을 빼고 최댓값을 최솟값으로 뺀 값을 나누어 스케일링 한다. 데이터의 타입을 float tensor 로 바꾼 후 입력 길이와 추론 길이를 고려하여 입력 값과 정답 값을 정의하는 클래스를 만든다.

```
class LSTMForecaster(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(LSTMForecaster, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)

        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])

        return out
```

LSTM 모델의 hidden, cell 과 학습을 정의한다. 이 모두를 device 로 보낸다.

```
seq_length = 24*9
pred_length = 24
batch_size = 32
dataset = Dataset(scaled_data, seq_length, pred_length)
train_size = int(0.8 * len(dataset))
train_dataset, val_dataset = torch.utils.data.random_split(dataset, [train_size, len(dataset) - train_size])
# 8:2로 쪼개기

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)

model = LSTMForecaster(input_size=len(features), hidden_size=64, num_layers=2, output_size=len(features)*pred_length)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters())


num_epochs = 300
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

하이퍼 파라미터 정의 및 train(shuffle=True), val 데이터 분할. 모델, 손실함수, 최적화 정의, device 를 정의한다.

```
num_epochs = 300
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

for epoch in range(num_epochs):
    model.train()
    train_loss = 0
    for batch_x, batch_y in train_loader:
        batch_x, batch_y = batch_x.to(device), batch_y.to(device)
        optimizer.zero_grad()
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y.view(batch_y.size(0), -1))
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    model.eval()
    val_loss = 0
    with torch.no_grad():
        for batch_x, batch_y in val_loader:
            batch_x, batch_y = batch_x.to(device), batch_y.to(device)
            outputs = model(batch_x)
            loss = criterion(outputs, batch_y.view(batch_y.size(0), -1))
            val_loss += loss.item()

    print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss/len(train_loader):.4f}, Val Loss: {val_loss/len(val_loader):.4f}')
```

학습을 수행하고, 매 epoch 마다 train_loss 와 val_loss 를 정의한다.

```
model.eval()
predictions = []

# 데이터셋을 예측값으로 바꾸기
with torch.no_grad():
    for i in range(0, len(scaled_data) - seq_length, pred_length):
        input_seq = torch.FloatTensor(scaled_data[i:i+seq_length]).unsqueeze(0).to(device)
        pred = model(input_seq)
        pred = pred.cpu().numpy().reshape(-1, len(features))
        predictions.append(pred)

all_predictions = np.vstack(predictions)

all_predictions_original = scaler.inverse_transform(all_predictions)

start_index = seq_length
end_index = start_index + len(all_predictions_original)
df.iloc[start_index:end_index, df.columns.get_indexer(features)] = all_predictions_original

print("Updated DataFrame:")
print(df)
```

데이터셋을 예측 값으로 변경다.

```
df1[["WindSpeed", "Sunshine", "AirPressure", "Radiation", "AirTemperature", "RelativeAirHumidity"]]
=df[["WindSpeed", "Sunshine", "AirPressure", "Radiation", "AirTemperature", "RelativeAirHumidity"]].values
```

새로운 데이터를 불러온 후 각 특성 열을 예측 값으로 변환한다.

```
class CustomTemporalFusionTransformer(TemporalFusionTransformer):
    def forward(self, x):
        x = {k: v.to(self.device) if isinstance(v, torch.Tensor) else v for k, v in x.items()}
        #딕셔너리 형태의 x를 key,value로 분리한 후 torch의 tensor객체라면 device로 보낸다. cpu or gpu(안됨..)
        return super().forward(x)
        #TemporalFusionTransformer로직에 추가 작업
```

딕셔너리 형태의 x 를 key, value 로 분리한 후 torch 객체라면 device 로보낸다. temporal trasformer 로직에 추가 작업을 적용한다.

```
def training_step(self, batch):
        x, y = self._process_batch(batch)
        outputs = self(x)
        #   예측을 수행
        predictions = self._get_predictions(outputs)
        #   torch.tensor형태의 output 추출
        loss = self.loss_func(predictions, y)
        #   quantileloss적용 로스값 구함
        self.log("train_loss", loss, on_step=True, on_epoch=True)
        #   각 step,epoch마다 loss를 기록함
        return loss

    def validation_step(self, batch, batch_idx):
        x, y = self._process_batch(batch)
        outputs = self(x)
        predictions = self._get_predictions(outputs)
        loss = self.loss_func(predictions, y)
        self.log("val_loss", loss, on_step=False, on_epoch=True)
        return loss



    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=self.model.hparams.learning_rate)
        # 옵티마이저 정의
    def forward(self, x):
        return self.model(x)#출력 생성
```

예측을 수행하여 output 을 추출한 후 loss 를 정의한다. 매 epoch 마다 loss 를 기록한다. Train, val step 을 정의한다. optimizer 을 정의한다.

```
max_encoder_length = 24 * 9 #입력 길이(부족할 경우 대부분 0으로 채움)
max_prediction_length = 24 #출력 길이

training_cutoff = df["time_idx"].max() - max_prediction_length
#   가장 최근 시점에서 예측하려는 길이를 뺌

time_varying_features = ["WindSpeed", "Sunshine", "AirPressure", "Radiation",
                         "AirTemperature", "RelativeAirHumidity", "SystemProduction"]

for feature in time_varying_features:
    mean_feature = df[feature].mean()
    df[f'mean_{feature}'] = mean_feature


training = TimeSeriesDataSet(
    df[lambda x: x.time_idx <= training_cutoff],#예측할 범위는 제외
    allow_missing_timesteps=True,
    time_idx="time_idx",
    target="SystemProduction",
    group_ids=["group_id"],
    static_categoricals=["group_id",'season'],# 고정 카테고리
    static_reals=["mean_WindSpeed", "mean_Sunshine", "mean_AirPressure", "mean_Radiation",
                  "mean_AirTemperature", "mean_RelativeAirHumidity", "mean_SystemProduction"],# 고정 실수값
    time_varying_known_reals=[ "WindSpeed","AirPressure", "Radiation",
                               "AirTemperature", "RelativeAirHumidity"], # 알수있는 실수값
    time_varying_unknown_reals=["SystemProduction"],# 알 수 없는 실수값
    max_encoder_length=max_encoder_length,
    max_prediction_length=max_prediction_length,
)
```

Encoder 길이와 예측 길이를 정의하여 과거의 9 일의 데이터로 미래의 하루를 예측하는 문제로 정의한다. Time idx, target, 고정 카테고리, 고정 실수값, 매 시간 알 수 있는 정보, 알 수 없는 정보를 결정하여 training 을 정의한다.

```
checkpoint_path = "/kaggle/working/tft_checkpoint.ckpt"
if os.path.exists(checkpoint_path):
    os.remove(checkpoint_path)
checkpoint_callback = ModelCheckpoint(
    monitor="val_loss",
    dirpath="/kaggle/working/",
    filename="tft-{epoch:02d}-{val_loss:.2f}",
    save_top_k=1,
    mode="min",
    save_last=True)
trainer = pl.Trainer(
    max_epochs=2,
    accelerator="cpu",
    devices=1,
    precision=32,
    gradient_clip_val=0.1,
    callbacks=[early_stop_callback, lr_logger, progress_bar, checkpoint_callback],
    logger=logger)
trainer.fit(
    tft,
    train_dataloaders=train_dataloader,
    val_dataloaders=val_dataloader,)
trainer.save_checkpoint(checkpoint_path)
best_model_path = trainer.checkpoint_callback.best_model_path
best_tft = LightningTemporalFusionTransformer.load_from_checkpoint(best_model_path)
predictions = best_tft.model.predict(val_dataloader)
actuals = torch.cat([y[0] for x, y in iter(val_dataloader)])
predictions = predictions.detach().cpu().numpy()
actuals = actuals.detach().cpu().numpy()
```

이미 check point 경로가 존재한다면 삭제한다. 저장 경로를 설정하여 checkpoint 를 정의한다. Epoch 수, 가속화 기기, 정밀도, callback 를 결정하여 trainer 를 정의한다.

```
validation = TimeSeriesDataSet.from_dataset(training, df, min_prediction_idx=training_cutoff + 1)
#  training의 형태를 기반, training의 마지막 시점 이후부터
batch_size = 256
train_dataloader = training.to_dataloader(train=True, batch_size=batch_size, num_workers=0)#pin_memory=True)
#       학습 용도(shuffle등이 적용), batchsize로 묶음, 여인 프로세스 사용
val_dataloader = validation.to_dataloader(train=False, batch_size=batch_size, num_workers=0) #pin_memory=True)

# 모델 학습
pl.seed_everything(42)
# 랜덤 시드 고정
tft = LightningTemporalFusionTransformer()#TFT모델 선언
if os.path.exists('/kaggle/working/tft-epoch=01-val_loss=9.10.ckpt'):
    best_tft = LightningTemporalFusionTransformer.load_from_checkpoint(best_model_path)
early_stop_callback = EarlyStopping(monitor="val_loss", min_delta=1e-4, patience=10, verbose=True, mode="min")
#  val_loss가 1e-4보다 10epoch동안 작으면 손실을 최소화하는 방향으로 모니터링한다.(상세정보 출력)
lr_logger = LearningRateMonitor()# 학습률 모니터링
logger = TensorBoardLogger("lightning_logs")# 로그를 'lightning_logs'에 기록
progress_bar = TQDMProgressBar(refresh_rate=1)# 1step마다 업데이트

checkpoint_path = "/kaggle/working/tft_checkpoint.ckpt"
```

결과 (Results)

Static reals = none, epoch=50, patience=10, lr=1e-3

```
MSE: 32.2155
```

Static reals != none, epoch=50, patience=10, lr=1e-3

```
MAE: 3.2223
MSE: 12.6167
```

Static reals != none, epoch=50, patience=15, lr=1e-2

```
MAE: 2.9587
MSE: 9.1420
```

Groud_ids= Season, Static reals != none, epoch=50, patience=15, lr=1e-3

```
MAE: 0.6058
MSE: 0.4698
```

결론 및 논의

태양광 발전량에 영향을 미치는 여러 요소들을 분석한 결과, TFT(Temporal Fusion Transformer) 모델을 사용하여 예측 성능을 개선할 수 있음을 확인하였다. 불규칙한 데이터에 대해서도 오차를 소수점 이하로 낮추는 성과를 거두었으나, 데이터 표준화를 적용하지 않은 실질적인 값에서는 만족스럽지 못한 오차를 보였다. 특히, 생산량이 음수로 예측되는 상황은 현실적이지 않으며, 과거 데이터를 반영하는 기간이 너무 짧은 점이 이러한 오차의 주요 원인으로 파악되었다. 따라서 모델의 신뢰성을 높이기 위해서는 음수 값이 불가능 하다는 정보와 더불어 과거 데이터를 충분히 반영할 수 있는 기간 설정이 필요하다.

추가탐구

태양광 발전량은 음수가 될 수 없다는 사실을 모델에 반영하고, 과거 시점의 데이터 범위를 조절함으로써 더 효율적인 예측 모델을 구축할 수 있을 것으로 판단된다. 또한, 25℃ 이상의 데이터가 부족한 상황에서 발생할 수 있는 일반화 오류 문제 또한 해결해야 한다.

REFERENCES

Zotero 또는 기타 인용 소프트웨어 사용을 적극 권장합니다. 저작물을 인용할 때는 직접 인용과 간접 인용을 구분해야 합니다. ** 본인에게서 비롯되지 않은 모든 생각이나 아이디어는 반드시 인용해야 합니다.

Afroz: (solar powe generation data)

https://www.kaggle.com/datasets/pythonafroz/solar-powe-generation-data

Jan Beitner(demand forecasting with the Temporal Fusion Transformer)

Demand forecasting with the Temporal Fusion Transformer - pytorch-forecasting documentation