

2021.09.17

# ／ TAnoGAN: Time Series Anomaly Detection with Generative Adversarial Networks

## TAnoGAN: Time Series Anomaly Detection with Generative Adversarial Networks

Md Abul Bashar

School of Computer Science

Centre for Data Science

Queensland University of Technology

Brisbane, Queensland 4000, Australia

Email: m1.bashar@qut.edu.au

Richi Nayak

School of Computer Science

Centre for Data Science

Queensland University of Technology

Brisbane, Queensland 4000, Australia

Email: r.nayak@qut.edu.au

**Abstract**—Anomaly detection in time series data is a significant problem faced in many application areas such as manufacturing, medical imaging and cyber-security. Recently, Generative Adversarial Networks (GAN) have gained attention for generation and anomaly detection in image domain. In this paper, we propose a novel GAN-based unsupervised method called TAnoGan for detecting anomalies in time series when a small number of data points are available. We evaluate TAnoGan with 46 real-world time series datasets that cover a variety of domains. Extensive experimental results show that TAnoGan performs better than traditional and neural network models.

time series data [10]. Recently a GAN framework coupled with the mapping of data to latent space has been explored for anomaly detection [3], [2]. While GAN has been extensively investigated in image domain for generation and anomaly detection, only a few works (e.g. [10], [2]) have explored the potential of GAN in time series domain.

In this paper, we propose a novel method, Time series Anomaly detection with GAN (TAnoGan)<sup>1</sup>, for unsupervised anomaly detection in time series data when a small number of data points are available. Detecting anomalies in time series

# 1 / 논문 리뷰

# Abstract

**Abstract**—Anomaly detection in time series data is a significant problem faced in many application areas such as manufacturing, medical imaging and cyber-security. Recently, Generative Adversarial Networks (GAN) have gained attention for generation and anomaly detection in image domain. In this paper, we propose a novel GAN-based unsupervised method called TAnoGan for detecting anomalies in time series when a small number of data points are available. We evaluate TAnoGan with 46 real-world time series datasets that cover a variety of domains. Extensive experimental results show that TAnoGan performs better than traditional and neural network models.

1. 시계열 데이터에서의 이상탐지는 제조, 의료영상, 사이버 보안 등 여러 분야에서 직면하는 문제이다.
2. 최근 GAN(Generative Adversarial Networks)은 이미지 분야에서 생성 및 이상탐지와 관련해 주목을 받고 있다.
3. 본 논문에서는 적은 수의 데이터를 가진 시계열에서의 이상탐지에 대해 TAnoGAN이라고 불리는 GAN기반의 비지도 학습 모델을 제시한다.
4. 46개의 다양한 도메인 데이터를 이용해 TAnoGAN을 평가하였으며, 여러 실험에서 TAnoGAN이 지곤의 신경망 모델들보다 성능이 우수하다는 것을 보여준다.

## 2 / 코드 리뷰, 모델구성

※ <https://github.com/mdabashar/TAnoGAN> source코드를 따라 단계별 Pytorch 코드 리뷰와 모델 구성을 파악하였습니다.

# 1. NabDataset

- 스트리밍 실시간 애플리케이션에서 이상 탐지를 위한 알고리즘을 평가하기 위한 새로운 벤치마크
- 50개 이상의 라벨이 붙은 실제 및 인공 시계열 데이터 파일과 실시간 애플리케이션을 위해 설계된 새로운 scoring mechanism으로 구성
- NabDataset 중 realKnownCause 분류에 속하는 ambient\_temperature\_system\_failure.csv를 사용
- realKnownCause는 이상 원인에 대해 알고 있는 데이터, no hand labeling
- ambient\_temperature\_system\_failure은 사무실 환경의 주변 온도를 나타냄
- 7207개의 행으로 구성
- Counter( {0.0: 6363, 1.0: 844})로 구성돼 844개의 이상치가 존재

<ambient\_temperature\_system\_failure>

timestamp	value
2013-07-04 0:00	69.88084
2013-07-04 1:00	71.22023
2013-07-04 2:00	70.8778
2013-07-04 3:00	68.9594
2013-07-04 4:00	69.28355
2013-07-04 5:00	70.06097
2013-07-04 6:00	69.27976
2013-07-04 7:00	69.36961
2013-07-04 8:00	69.16671
2013-07-04 9:00	68.98608
2013-07-04 10:00	69.96506
2013-07-04 11:00	70.55619
2013-07-04 12:00	70.30751
2013-07-04 13:00	70.24625
...	...
2014-05-28 15:00	72.58409

# 1. NabDataset

Real data	realAWSCloudwatch		AmazonCloudwatch 서비스에서 수집한 CPU사용 등 AWS 서버 측정
	realAdExchange		온라인 광고 클릭률: 클릭당 비용(CPC) 및 1,000개당 비용(CPM)
	realKnownCause	ambient_temperature_system_failure.csv	사무실 환경의 주변 온도
		cpu_utilization_asg_misconfiguration.csv	AWS(Amazon Web Services)에서 CPU 사용량 모니터링
		ec2_request_latency_system_failure.csv	Amazon의 East Coast 데이터 센터에 있는 서버의 CPU 사용량 데이터
		machine_temperature_system_failure.csv	대형 산업용 기계의 내부 구성 요소의 온도 센서 데이터
		nyc_taxi.csv	NYC 마라톤, 추수감사절, 크리스마스, 설날 및 눈폭풍 동안 5가지 이상 현상이 발생하는 NYC 택시 승객의 수
		rogue_agent_key_hold.csv	Timing the key holds for several users of a computer
		rogue_agent_key_updown.csv	Timing the key strokes for several users of a computer
	realTraffic		미네소타 트윈 시티 메트로 지역의 실시간 교통 데이터, 특정 센서의 승객, 속도 및 이동 시간이 포함
	realTweets		구글과 IBM과 같은 대형 상장 기업들의 트위터 언급 모음, 매 5분마다 지정된 언급 기호에 대한 언급 수
Artificial data	artificialNoAnomaly		이상 현상 없이 인위적으로 생성된 데이터
	artificialWithAnomaly		다양한 유형의 변칙으로 인위적으로 생성된 데이터

## 2. Make Dataset

pytorch의 Dataset을 상속받아 데이터를 구성

### Class NabDataset(Dataset):

```
def read_data(self, data_file=None, label_file=None, key=None, BASE=''):
    with open(BASE+label_file) as FI:
        j_label = json.load(FI)
        ano_spans = j_label[key]
    self.ano_span_count = len(ano_spans)
    df_x = pd.read_csv(BASE+data_file)
    df_x, df_y = self.assign_ano(ano_spans, df_x)

    return df_x, df_y
```

```
def assign_ano(self, ano_spans=None, df_x=None):
    df_x['timestamp'] = pd.to_datetime(df_x['timestamp'])
    y = np.zeros(len(df_x))
    for ano_span in ano_spans:
        ano_start = pd.to_datetime(ano_span[0])
        ano_end = pd.to_datetime(ano_span[1])
        for idx in df_x.index:
            if df_x.loc[idx, 'timestamp'] >= ano_start and df_x.loc[idx, 'timestamp'] <= ano_end:
                y[idx] = 1.0
    return df_x, pd.DataFrame(y)
```

- ① NabDataset의 50가지 데이터 Label 정보는 labels/ combined\_windows.json에 집합형태로 저장되어 있다. \*\*Appendix에 파일형태 첨부
- ② Ex)"realKnownCause/ambient\_temperature\_system\_failure.csv":  
[["2013-12-15 07:00:00.000000", "2013-12-30 09:00:00.000000"], ["2014-03-29 15:00:00.000000", "2014-04-20 22:00:00.000000"]]
- ③ Ano\_spans는 사용하는 데이터를 Key로 입력받아 그 데이터의 Label정보를 불러온 객체이다.

- ① 위에서 얻은 Label정보를 이용해 실제 데이터에 1을 부여하는 함수
- ② ["2013-12-15 07:00:00.000000", "2013-12-30 09:00:00.000000"] 사이에 해당하는 데이터에 모두 1을 부여하고 있다.

## 2. Make Dataset

pytorch의 Dataset을 상속받아 데이터를 구성

### Class NabDataset(Dataset):

```
# create sequences
def unroll(self, data, labels):
    un_data = []
    un_labels = []
    seq_len = int(self.window_length)
    stride = int(self.stride)

    idx = 0
    while(idx < len(data) - seq_len):
        un_data.append(data.iloc[idx:idx+seq_len].values)
        un_labels.append(labels.iloc[idx:idx+seq_len].values)
        idx += stride
    return np.array(un_data), np.array(un_labels)
```

```
## Dataset은 반드시 __len__ 함수를 만들어줘야함(데이터 길이)
## The __len__ function returns the number of samples in our dataset.
def __len__(self):
    return self.data_len
```

```
## torch 모듈은 __getitem__ 을 호출하여 학습할 데이터를 불러옴.
## The __getitem__ function loads and returns a sample from the dataset at the given index idx
def __getitem__(self, idx):
    return self.x[idx], self.y[idx]
```

- ① Unroll함수는 **sequence** 형태로 변환해주는 함수이다.
- ② 현재 window\_length는 60, stride는 1로 설정
- ③ data.iloc[idx:idx+seq\_len].values을 통해 seq\_len(60)만큼의 길이를 가지는 value값이 하나의 sequence가 된다.
- ④ Un\_label또한 60개의 값이 하나의 sequence에 대한 label로 담겨지는데 이는 나중에 전체 합이 0보다 크면 1의 label을 갖도록 변환된다.
- ⑤ self.y = torch.from\_numpy(np.array([1 if sum(y\_i) > 0 else 0 for y\_i in y])).float()

- ① Pytorch Dataset을 상속받은 부분, self.data\_len = x.shape[0] 다음을 통해 **데이터 길이를 반환**

- ② Item(index)에 해당하는 데이터(2번에서 만든 **input\_size의 tensor데이터**)를 실제로 반환



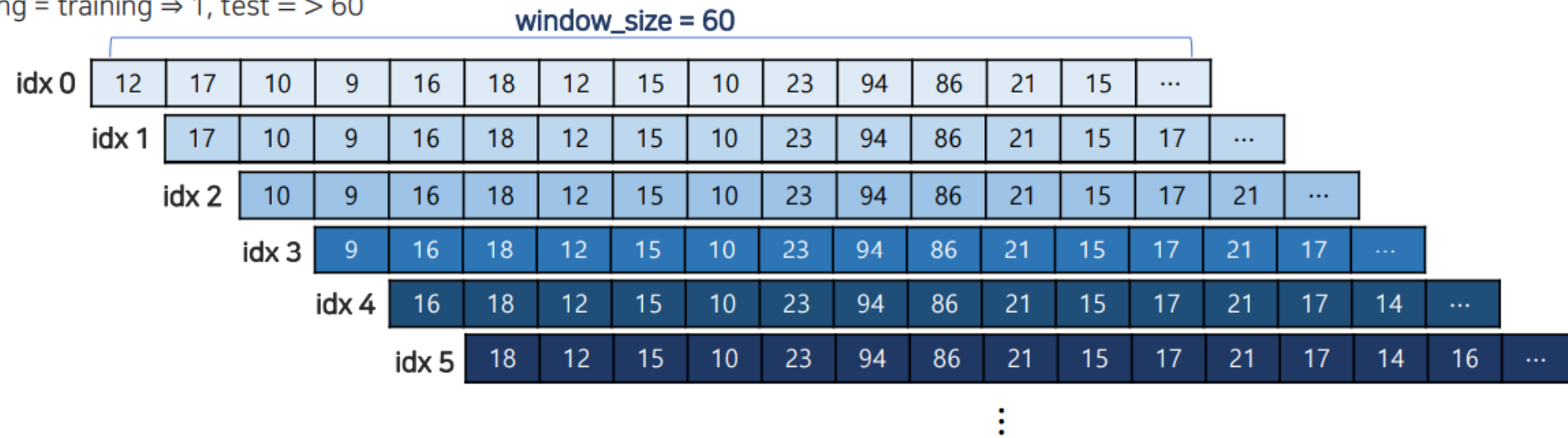
## 2. Make Dataset

### Dataset\_Example :

[Original Sequence]

Time stamp	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...
Value	12	17	10	9	16	18	12	15	10	23	94	86	21	15	17	21	17	14	16	12	...

- Window size = 60
- Sliding = training  $\Rightarrow$  1, test  $= > 60$



## 2. Make Dataset

### Dataset\_Example :

timestamp	value
2013-07-04 0:00	69.88084
2013-07-04 1:00	71.22023
2013-07-04 2:00	70.8778
2013-07-04 3:00	68.9594
2013-07-04 4:00	69.28355
2013-07-04 5:00	70.06097
2013-07-04 6:00	69.27976
2013-07-04 7:00	69.36961
2013-07-04 11:00	70.55619
...	...
2014-05-28 15:00	72.58409

- window\_length를 60으로 입력받아 60개의 row에 대해 정규화된 값들이 반환
- window의 개념으로 X[1]의 첫번째 값은 X[0]의 두번째가 됨을 볼 수 있다.

In [2]: dataset.x[0]

Out [2]: tensor([[ -0.3206],  
[ -0.0052],  
[ -0.0859],  
[ -0.5375],  
[ -0.4612],  
[ -0.2782],  
[ -0.4621],  
[ -0.4410],  
[ -0.4887],  
[ -0.5313],  
[ -0.3008],  
[ -0.1616],

[ -0.4607],  
[ -0.3570],  
[ -0.7187],  
[ -0.5476]])

In [2]: dataset.x[1]

Out [2]: tensor([[ -5.2283e-03],  
[ -8.5851e-02],  
[ -5.3754e-01],  
[ -4.6122e-01],  
[ -2.7817e-01],  
[ -4.6211e-01],  
[ -4.4095e-01],  
[ -4.8872e-01],  
[ -5.3125e-01],  
[ -3.0075e-01],  
[ -1.6157e-01],  
[ -2.2013e-01],

[ -3.5695e-01],  
[ -7.1867e-01],  
[ -5.4757e-01],  
[ -9.3573e-01]])

In [2]: dataset.x[2]

Out [2]: tensor([[ -8.5851e-02],  
[ -5.3754e-01],  
[ -4.6122e-01],  
[ -2.7817e-01],  
[ -4.6211e-01],  
[ -4.4095e-01],  
[ -4.8872e-01],  
[ -5.3125e-01],  
[ -3.0075e-01],  
[ -1.6157e-01],  
[ -2.2013e-01],  
[ -2.3455e-01],

[ -7.1867e-01],  
[ -5.4757e-01],  
[ -9.3573e-01],  
[ -1.0071e+00]])

## 2. Make Dataset

TORCH.UTILS.DATA

### Class DataLoader :

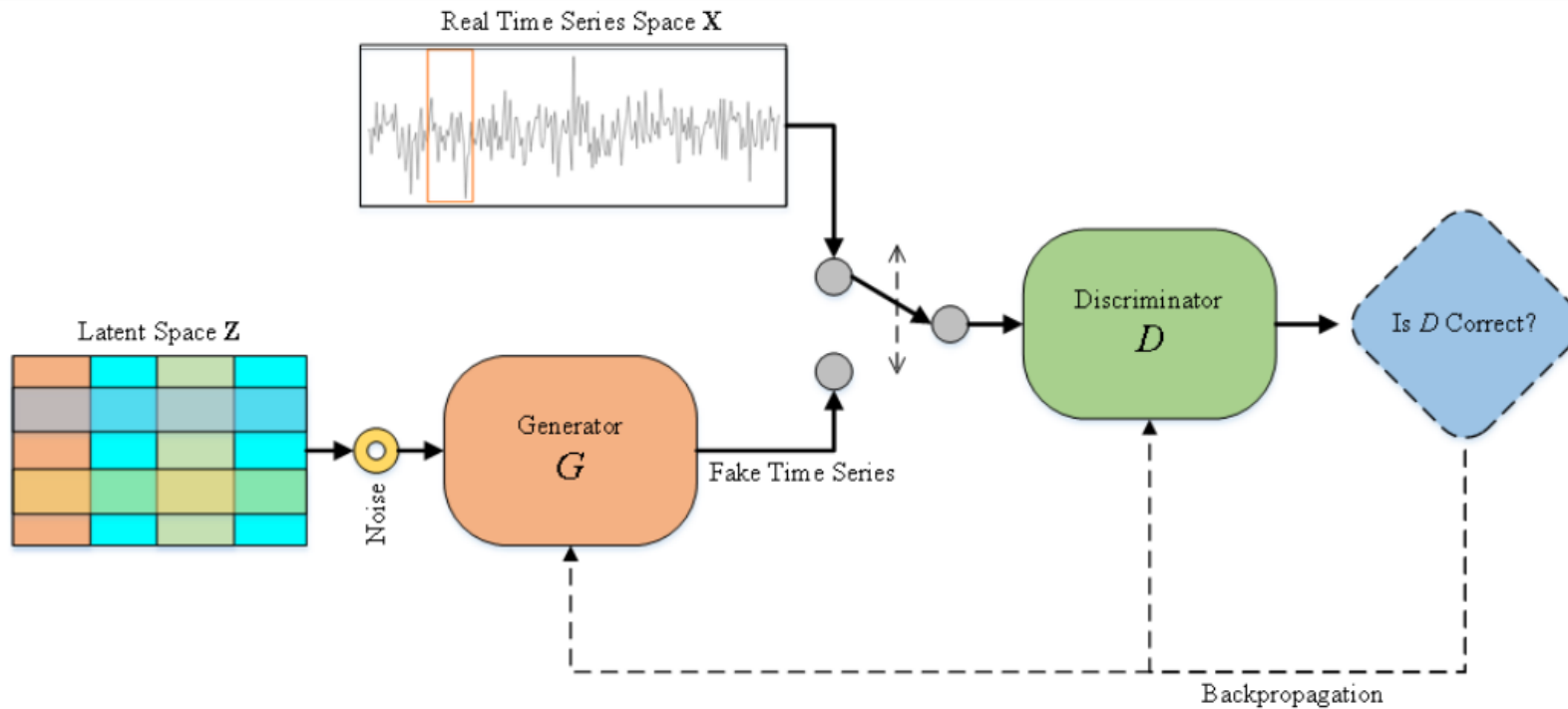
```
In [2]: DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
                  batch_sampler=None, num_workers=0, collate_fn=None,
                  pin_memory=False, drop_last=False, timeout=0,
                  worker_init_fn=None, *, prefetch_factor=2,
                  persistent_workers=False)
```

```
In [2]: class ArgsTrn:
          workers=4
          batch_size=32
          epochs=20
          lr=0.0002

          opt_trn=ArgsTrn()
```

- torch.utils.data를 이용해 Data Loader 형태로 변환
- **Dataloader class**는 **batch기반의 딥러닝모델 학습을 위해서 mini batch를 만들어주는 역할**
- 앞서 만들었던 dataset을 input으로 넣어주면 여러 옵션(데이터 묶기, 섞기, 알아서 병렬처리)을 통해 batch 생성
- batch\_size=opt\_trn.batch\_size
- "batch\_size " 를 32로 두어 앞서 보았던 3개의 window를 가지는 tensor 32개가 하나의 집단이 되어 학습이 이루어진다.

### 3. Model Architecture

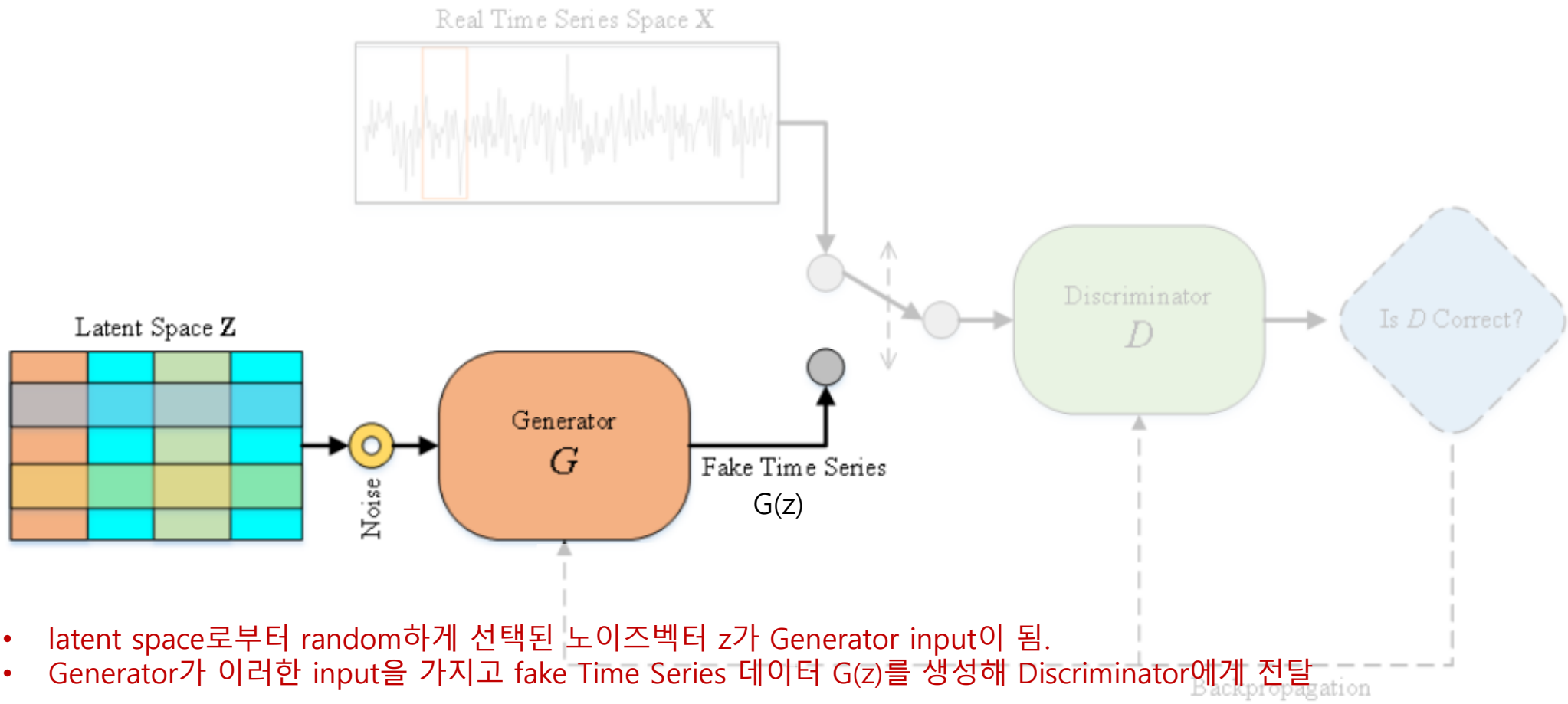


(a) Representing Normal Time Series with Generator  $G$

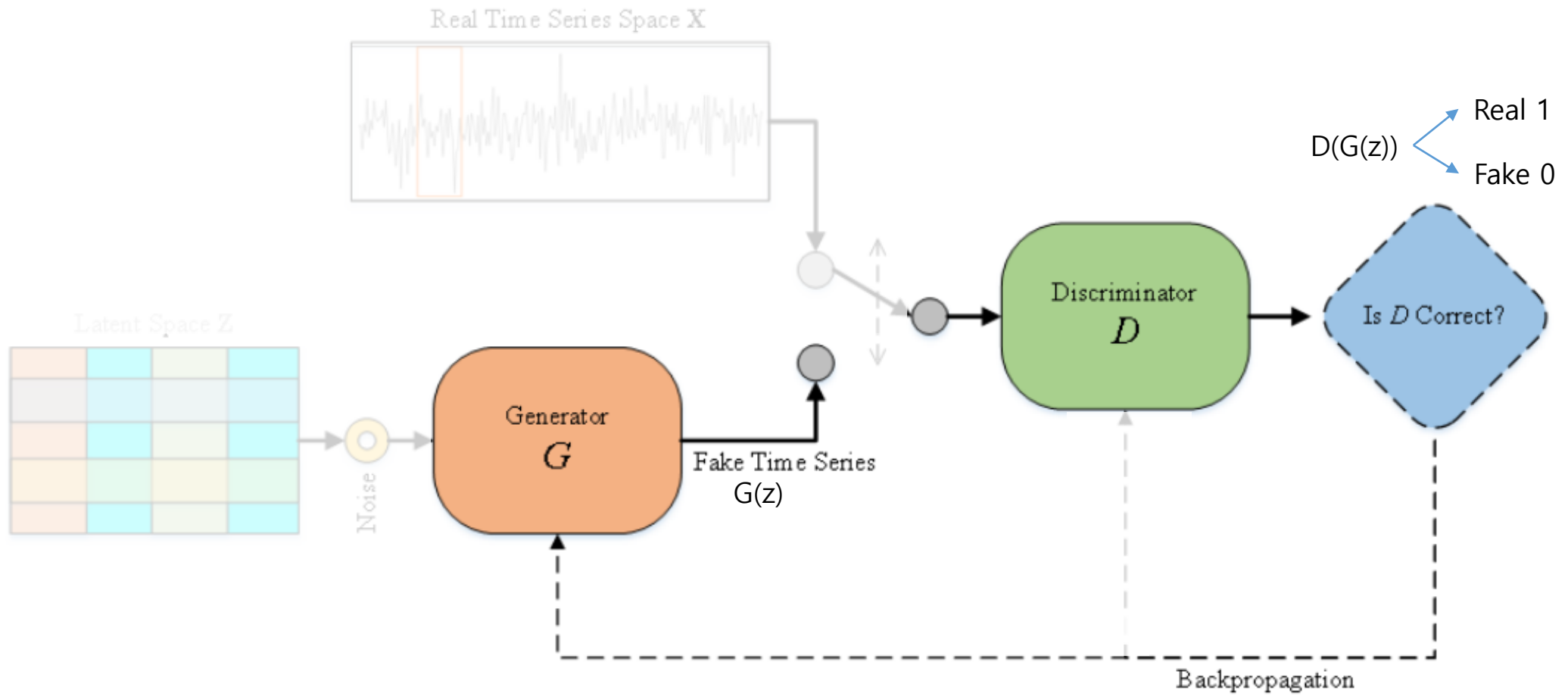
#### Training

- fake 데이터를 생성하는 generator  $G$ 와 생성된 fake 데이터를 real 데이터와 판별을 하는 Discriminator를 adversarial 하게 학습
- Generator와 Discriminator 모두 Input과 Output이 Time series의 Sequence 형태이므로 LSTM을 사용

### 3. Model Architecture

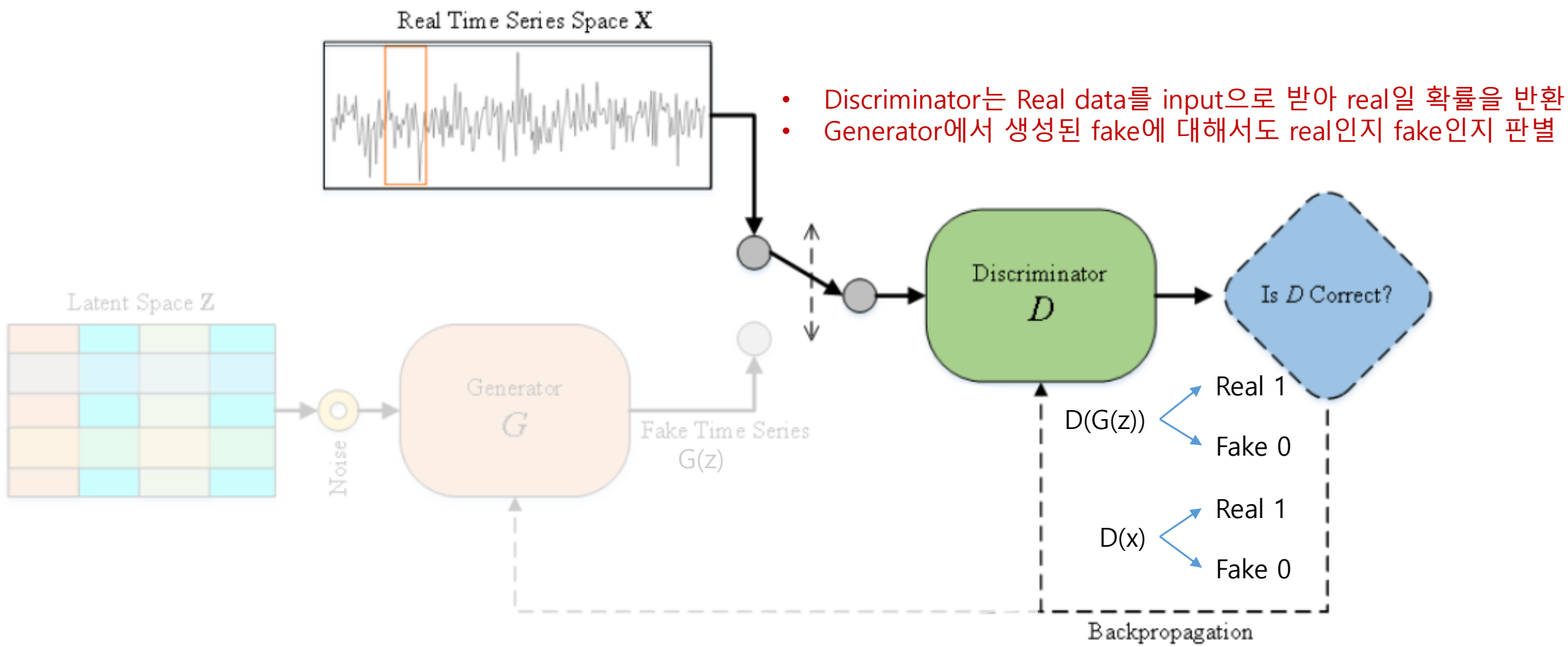


### 3. Model Architecture



- Generator는 Discriminator를 속일만큼 realistic data를 생성하는 것이 목표
- $G$ 가 생성한 데이터에 대해서는 Discriminator는 real 1을 반환해야 함
- 즉  $D(G(z))$ 가 1이냐 0이냐에 따라 이 차이를 역전파하여  $G$ 를 학습

### 3. Model Architecture



- Discriminator입장에서 Generator가 생성한 fake데이터를 fake 0으로 판별해야하므로  $D(G(z))$ 값과 정답 0의 차이에 의해 역전파가 발생
- Real 데이터에 대해서도 Discriminator는 real 1로 판단해야 하기 때문에  $D(x)$ 값과 정답 1의 차이에 의해 역전파가 발생

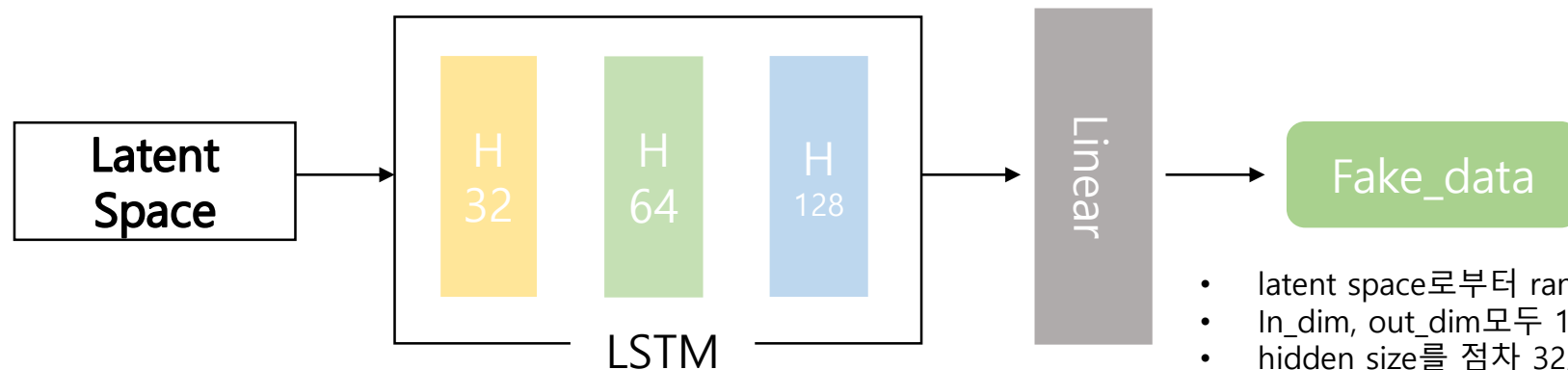
### 3. Model Architecture

**Class LSTMGenerator(nn.Module) :**

```
In [2]: def __init__(self, in_dim, out_dim, device=None):
        super().__init__()
        self.out_dim = out_dim
        self.device = device

        self.lstm0 = nn.LSTM(in_dim, hidden_size=32, num_layers=1, batch_first=True)
        self.lstm1 = nn.LSTM(input_size=32, hidden_size=64, num_layers=1, batch_first=True)
        self.lstm2 = nn.LSTM(input_size=64, hidden_size=128, num_layers=1, batch_first=True)

        self.linear = nn.Sequential(nn.Linear(in_features=128, out_features=out_dim), nn.Tanh())
```



- latent space로부터 random하게 선택된 노이즈벡터  $z$ 가 input으로 들어옴
- In\_dim, out\_dim 모두 1차원
- hidden size를 점차 32, 64, 128로 늘린 stacked LSTM을 사용

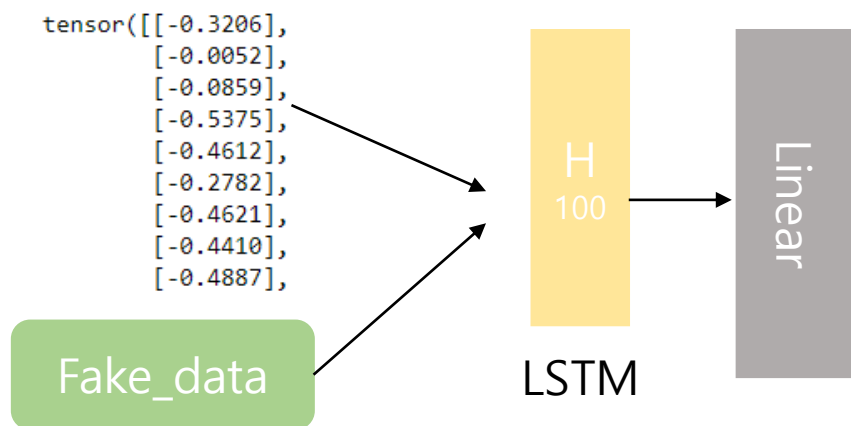


### 3. Model Architecture

**Class** LSTMDiscriminator(nn.Module) :

```
In [2]: def __init__(self, in_dim, device=None):
        super().__init__()
        self.device = device

        self.lstm = nn.LSTM(input_size=in_dim, hidden_size=100, num_layers=1, batch_first=True)
        self.linear = nn.Sequential(nn.Linear(100, 1), nn.Sigmoid())
```



- 깊은 discriminator를 사용하면 충분히 학습시킬만한 데이터가 존재하지 않기 때문에 과적합이 쉽게 발생
- 간단하고 얇은 구조로 싱글LSTM layer를 가지고 hidden unit은 100개인 discriminator를 설계, In\_dim 출력 모두 1차원

### 3. Model Architecture

#### Algorithm 1: Algorithm for TAnoGan

**Input:** A list of small sequences  $\mathbf{X}$ .

**Output:** A list of anomaly scores  $A$ .

```
1 Function adversarialTrain( $\mathbf{X}$ ):  
2   for number_of_epochs do  
3     Sample  $m$  noise vectors  $\{\mathbf{z}_1, \dots, \mathbf{z}_m\}$  from the noise prior  $p_g(\mathbf{z})$ .  
4     Generate  $m$  fake-data vectors  $\{G(\mathbf{z}_1), \dots, G(\mathbf{z}_m)\}$  from the  $m$   
       noise vectors.  
5     Sample  $m$  real-data vectors  $\{\mathbf{x}_1, \dots, \mathbf{x}_m\}$  from the data generating  
       distribution  $p_{data}(\mathbf{x})$ .  
6     Train  $D$  on the fake-data vectors and real-data vectors.  
7     Sample another  $m$  noise vectors  $\{\mathbf{z}_1, \dots, \mathbf{z}_m\}$  to from the noise prior  
        $p_g(\mathbf{z})$ .  
8     Train  $G$  on the second set of noise vectors.  
9   return  $G, D$ 
```

#### Summary

Epoch수만큼

- Noise vector  $\mathbf{z}$ 를  $m$ 개 sampling
- $M$ 개의 noise vector로부터 fake data인  $\{G(\mathbf{z}_1), \dots, G(\mathbf{z}_m)\}$ 을 생성
- $M$ 개의 real data를 sampling
- Fake data와 real data를 이용해 Discriminator 학습
- Latent space에서 또다른  $m$ 개의 noise vector sampling
- 두번째 noise vector를 이용해 Generator 학습

# 4. Anomaly Score

## Algorithm 1: Algorithm for TAnoGan

**Input:** A list of small sequences  $\mathbf{X}$ .

**Output:** A list of anomaly scores  $A$ .

```

1 Function adversarialTrain( $\mathbf{X}$ ):
2   for number_of_epochs do
3     Sample  $m$  noise vectors  $\{\mathbf{z}_1, \dots, \mathbf{z}_m\}$  from the noise prior  $p_g(\mathbf{z})$ .
4     Generate  $m$  fake-data vectors  $\{G(\mathbf{z}_1), \dots, G(\mathbf{z}_m)\}$  from the  $m$ 
       noise vectors.
5     Sample  $m$  real-data vectors  $\{\mathbf{x}_1, \dots, \mathbf{x}_m\}$  from the data generating
       distribution  $p_{data}(\mathbf{x})$ .
6     Train  $D$  on the fake-data vectors and real-data vectors.
7     Sample another  $m$  noise vectors  $\{\mathbf{z}_1, \dots, \mathbf{z}_m\}$  to from the noise prior
        $p_g(\mathbf{z})$ .
8     Train  $G$  on the second set of noise vectors.
9   return  $G, D$ 

```

```

10 Function anomalyScore( $\mathbf{X}, G, D$ ):
11   for  $i$  in 1 to  $m$  do
12     Sample a noise vector  $\mathbf{z}^i$  from the noise prior  $p_g(\mathbf{z}^i)$ .
13     for  $\lambda$  in 1 to  $\Lambda$  do
14       Generate a fake-data vector  $G(\mathbf{z}^i)$  from the noise vector  $\mathbf{z}^i$ .
15       Calculate  $\mathcal{L}(G(\mathbf{z}^i))$  for  $\mathbf{x}^i$  utilising  $G$  and  $D$ , and update  $\mathbf{z}^i$ 
16         using gradient descent.
17      $A(\mathbf{x}^i) = \mathcal{L}(G(\mathbf{z}^i))$ 
18   return  $A$ 

```

```

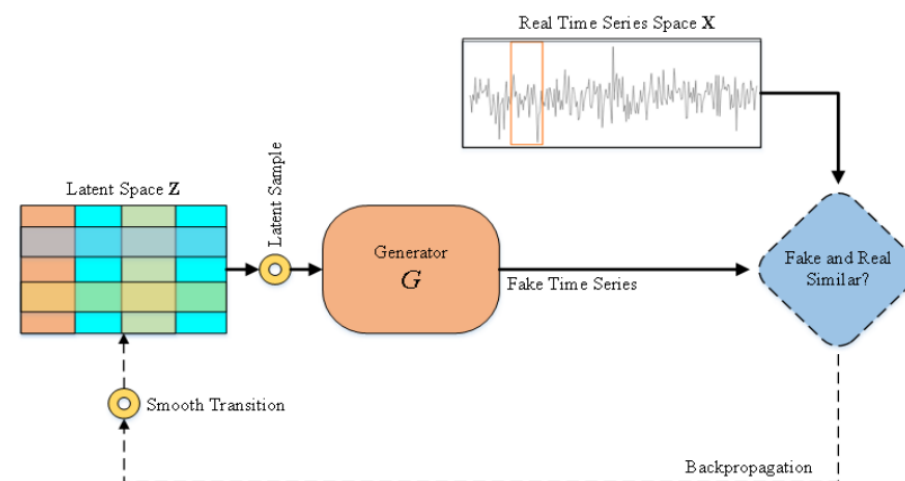
19 Function Main( $\mathbf{X}$ ):
20    $G, D = \text{adversarialTrain}(\mathbf{X})$ 
21    $A = \text{anomalyScore}(\mathbf{X}, G, D)$ 

```

## Summary

Discriminator와 Generator의 학습이 끝나면 새로운 데이터에 대해 anomaly score를 계산

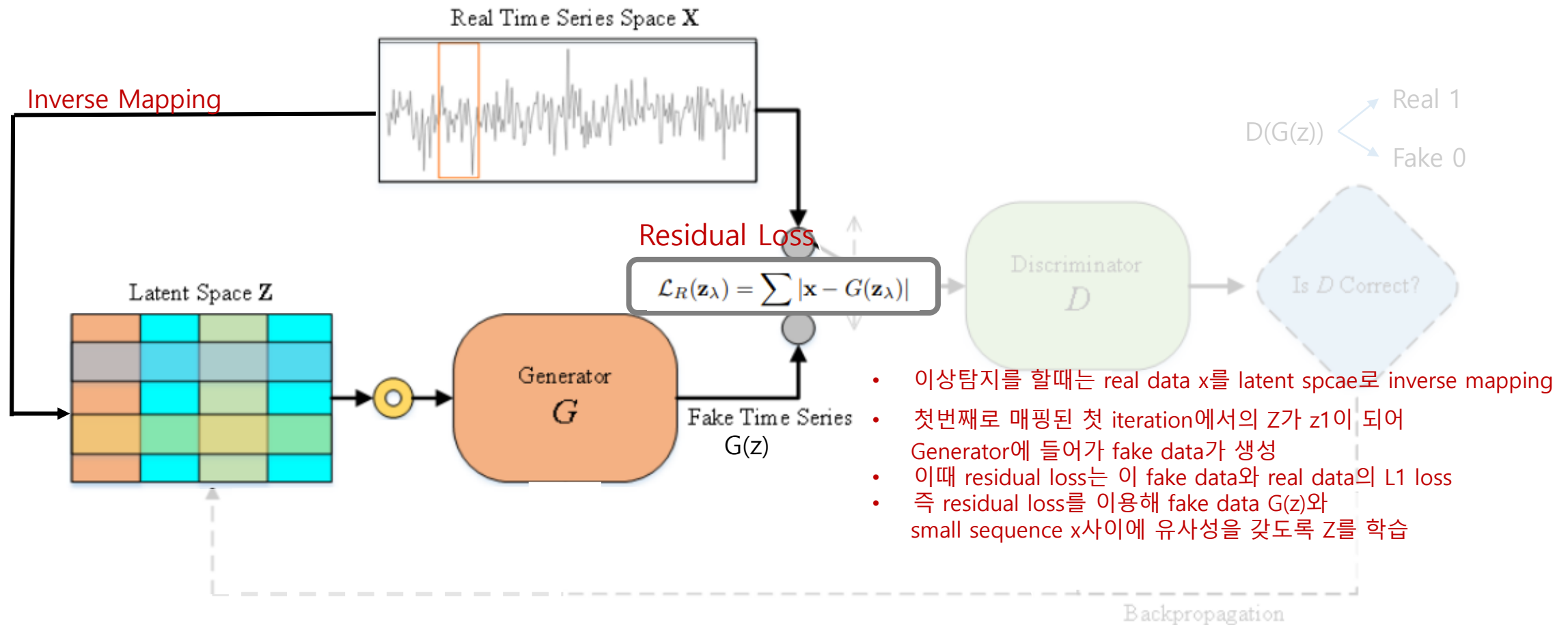
- Latent space로부터 noise vector를 sampling
- 이 Noise vector로부터 Generator는 fake data를 생성
- G와 D를 이용해 실제 데이터  $x$ 에 대한  $\text{Loss}(G(z))$ 를 계산
- 즉 fake data로 부터 얻은 loss를 경사하강법을 이용해  $z$ 를 업데이트



(b) Mapping Real-Data to the Latent Space

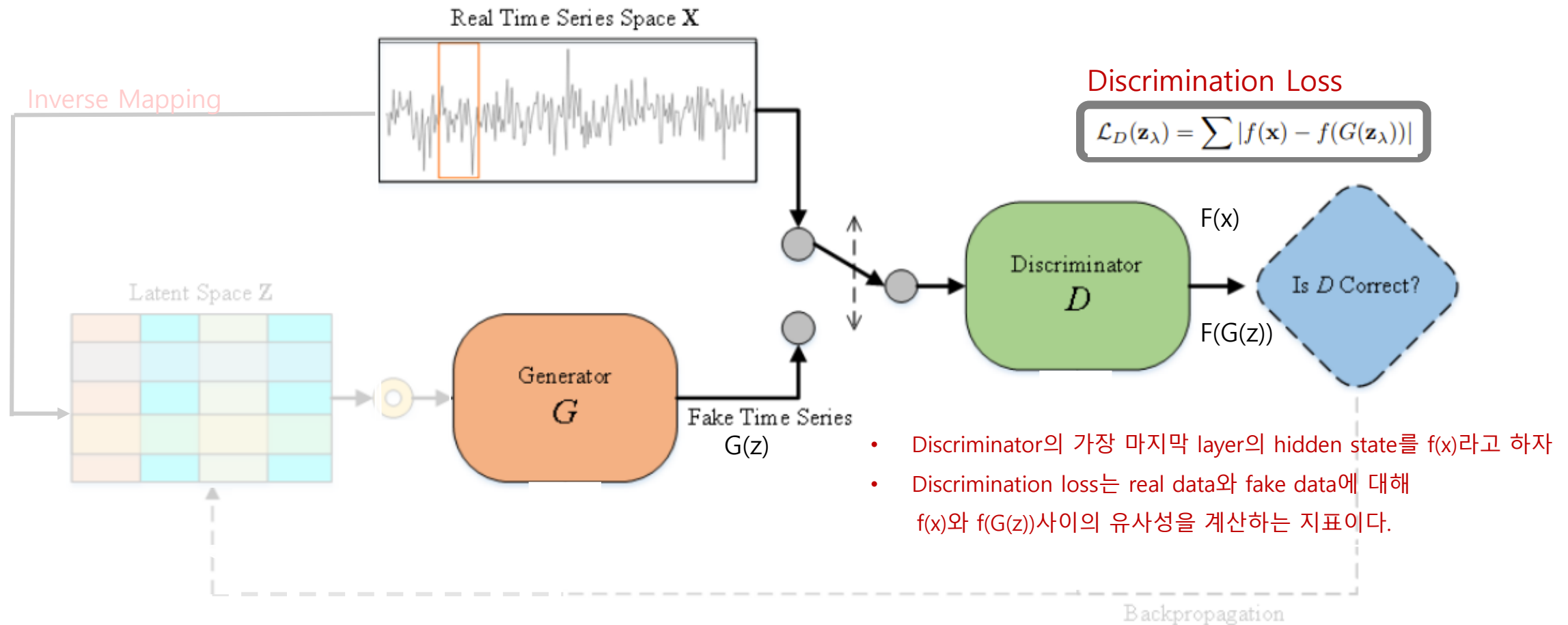
# 4. Anomaly Score

비정상 점수 계산: 1. Inverse mapping to latent space and Residual Loss



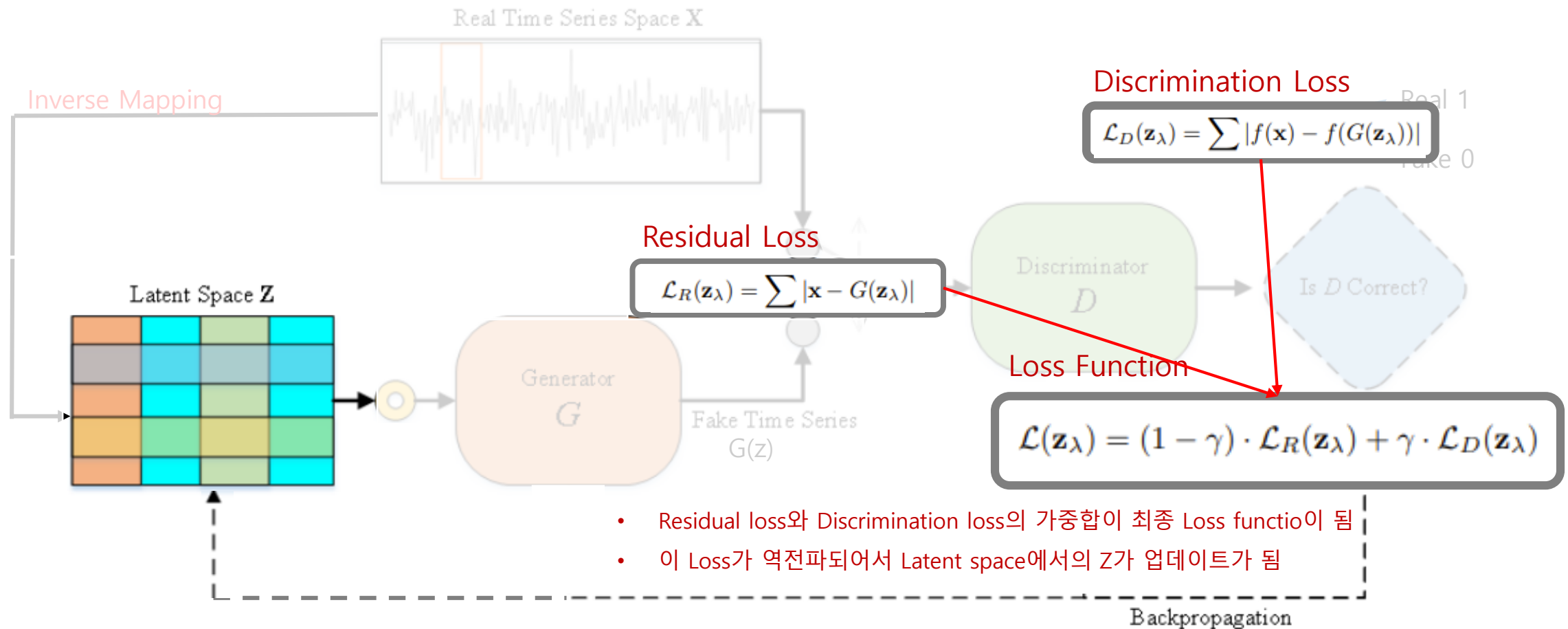
# 4. Anomaly Score

## 비정상 점수 계산: 2. Discrimination Loss



# 4. Anomaly Score

## 비정상 점수 계산: 3. Loss Function and Backpropagation



## 4. Anomaly Score

### Code: Inverse mapping to latent space and reconstruction of data

loss\_list : 테스트셋에 대해서 모든 행마다 50번의 iteration(업데이트)을 통해 latent space상에서의 z를 확정하고 해당데이터의 loss를 저장

In [2]:

```
loss_list = []

for i, (x,y) in enumerate(test_dataloader):
    print(i, y)

    z = Variable(init.normal(torch.zeros(opt_test.batch_size,
                                         test_dataset.window_length,
                                         test_dataset.n_feature),mean=0,std=0.1),
                 requires_grad=True)

    #z = x
    z_optimizer = torch.optim.Adam([z],lr=1e-2)

    loss = None
    for j in range(50): # set your iteration range
        gen_fake,_ = generator(z)
        loss = Anomaly_score(Variable(x), gen_fake)
        loss.backward()
        z_optimizer.step()

    loss_list.append(loss)
    print('~~~~~loss={}, y={}' ~~~~~'.format(loss, y))
```

[tensor(465.1875, grad\_fn=<AddBackward0>),  
tensor(560.8277, grad\_fn=<AddBackward0>),  
tensor(508.7813, grad\_fn=<AddBackward0>),

- ① 처음에는 랜덤하게 noise vector z를 생성
- ② Loss를 통해 업데이트할 대상은 z이다.
- ③ 각 iteration에서의 z를 generator에 넣어서 fake데이터를 생성
- ④ Anomaly\_score 함수를 이용해 fake data와 real data x의 loss를 반환
- ⑤ Loss를 역전파하여 z를 업데이트
- ⑥ 마지막 iteration을 돌았을 때 나오는 latent space상에서의 데이터 포인트 z를 확정하고 해당 데이터의 loss를 저장

## 4. Anomaly Score

### Code: Anomaly\_Score 함수

predict값과 real값의 loss를 담은 loss\_list의 평균과 표준편차를 통해 이상치 점수를 산출

```
In [2]: # Lambda = 0.1 according to paper
# x is new data, G_z is closely regenerated data

def Anomaly_score(x, G_z, Lambda=0.1):
    residual_loss = torch.sum(torch.abs(x-G_z)) # Residual Loss

    output, x_feature = discriminator(x.to(device))
    output, G_z_feature = discriminator(G_z.to(device))

    # Discrimination Loss
    discrimination_loss = torch.sum(torch.abs(x_feature-G_z_feature))

    total_loss = (1-Lambda)*residual_loss.to(device) + Lambda*discrimination_loss
    return total_loss
```

① Residual Loss

② Real x와 generator가 생성한 fake data인 G\_z의 L1\_loss

③ Discrimination Loss

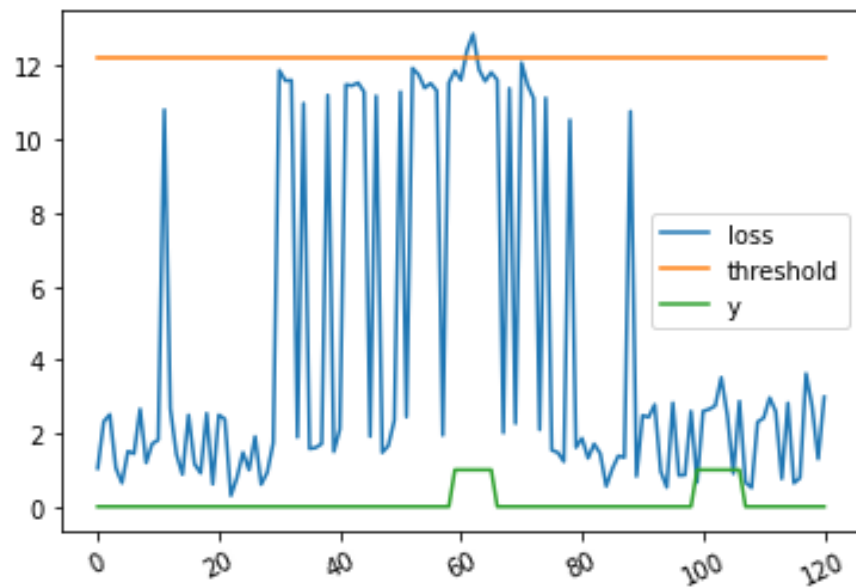
④ Real data와 fake data에 대해 Discriminator의 가장 마지막 layer를 통과시킨  $f(x)$ 와  $f(G(z))$ 의 L1\_loss

⑤ Lambda는 residual loss와 discrimination loss의 trade\_off를 조정하는 가중치로 논문에서는 0.1로 설정하였다.

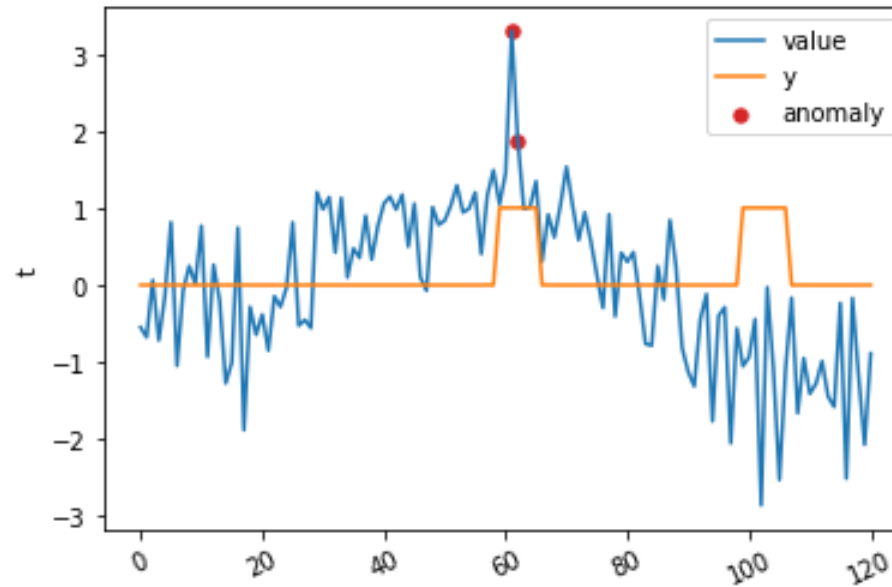


## 5. Visualize Result

Loss\_list Graph



Real data Graph




- 앞서 구한 Loss\_list를 시각화하고 threshold를 12.5로 두어 loss가 12.5이상인 경우 이상치로 탐지
- 오른쪽 그래프는 실제 test data의 values를 시각화한 것으로 loss를 통해 탐지한 이상치가 실제 이상치(주황색 그래프)와 얼마나 맞는지를 비교
- 2구간의 이상치 중 한 구간의 이상치를 탐지한 것으로 보여짐

# 3 / **Appendix**

# Labels Data

pytorch의 Dataset을 상속받아 데이터를 구성

## Class NabDataset(Dataset):



```
def read_data(self, data_file=None, label_file=None, key=None, BASE=''):
    with open(BASE+label_file) as FI:
        j_label = json.load(FI)
        ano_spans = j_label[key]
    self.ano_span_count = len(ano_spans)
    df_x = pd.read_csv(BASE+data_file)
    df_x, df_y = self.assign_ano(ano_spans, df_x)

    return df_x, df_y
```

```
{'artificialNoAnomaly/art_daily_no_noise.csv': [],
 'artificialNoAnomaly/art_daily_perfect_square_wave.csv': [],
 'artificialNoAnomaly/art_daily_small_noise.csv': [],
 'artificialNoAnomaly/art_flatline.csv': [],
 'artificialNoAnomaly/art_noisy.csv': [],
 'artificialWithAnomaly/art_daily_flatmiddle.csv': [['2014-04-10 07:15:00.000000',
 '2014-04-11 16:45:00.000000']],
 'artificialWithAnomaly/art_daily_jumpsdown.csv': [['2014-04-10 16:15:00.000000',
 '2014-04-12 01:45:00.000000']],
 'artificialWithAnomaly/art_daily_jumpsup.csv': [['2014-04-10 16:15:00.000000',
 '2014-04-12 01:45:00.000000']],
 'artificialWithAnomaly/art_daily_nojump.csv': [['2014-04-10 16:15:00.000000',
 '2014-04-12 01:45:00.000000']],
```

- ① 다음은 j\_label의 실제모습을 보여주고 있다.
- ② NabDataset의 50가지 데이터 Label 정보는 labels/ combined\_windows.json에 집합형태로 저장되어 있다.
- ③ 위의 사전처럼 각각의 csv파일에 대해 이상치에 해당하는 날짜의 정보가 리스트에 담겨져 있다.
- ④ []처럼 빈칸으로 표시된것은 이상치가 없음을 의미하고 ["2014-04-10", "2014-04-12"] 와 같이 표시된것은 4-10일부터 4/12사이에 해당하는 데이터 모두 이상치임을 나타낸다.