

# Future Frame Prediction for Anomaly Detection – A New Baseline

DEEP LEARNING BASED ANOMALY DETECTION MODELING (GEK6207.01-00)

Coding practice

Sunghyun Ahn

[skd@yonsei.ac.kr](mailto:skd@yonsei.ac.kr)

<2023/11/23>

# 3 Future Frame Prediction

## Unsupervised Video Anomaly Detection

A method of anomaly detection that trains only on normal data and categorizing anything dissimilar to the patterns in normal data as anomalies. In supervised learning, **class imbalance** may arise as the number of normal data is much higher than anomalies. Additionally, even with effective learning, **overfitting** can occur due to the unpredictable real-world anomaly data. Thus train with **frame reconstruction** or **frame prediction** methods using unsupervised learning and attempt to detect anomalies using similarities (e.g., PSNR) between reconstructed and real frames.

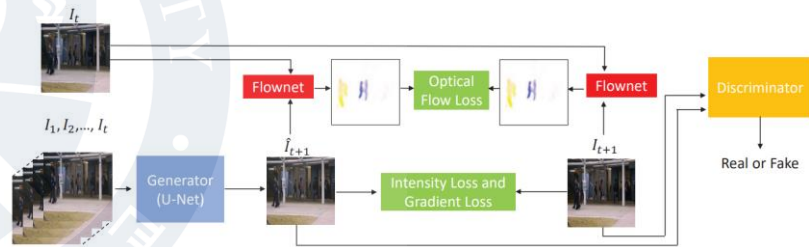
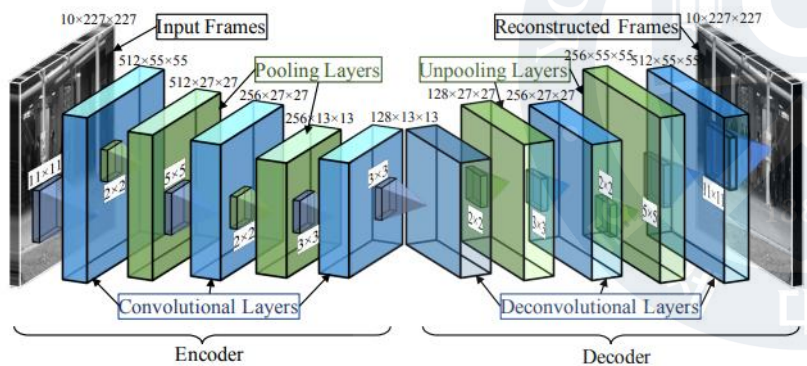


Figure 2. The pipeline of our video frame prediction network. Here we adopt U-Net as generator to predict next frame. To generate high quality image, we adopt the constraints in terms of appearance (intensity loss and gradient loss) and motion (optical flow loss). Here FlowNet is a pretrained network used to calculate optical flow. We also leverage the adversarial training to discriminate whether the prediction is real or fake.

Hasan, Mahmudul, et al. "Learning temporal regularity in video sequences." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.  
Liu, Wen, et al. "Future frame prediction for anomaly detection—a new baseline." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.

# 3 Future Frame Prediction

## Future Frame Prediction

Deep learning models derived from the assumption that abnormal future frames cannot be predicted if trained only on predicting future frames of normal data.

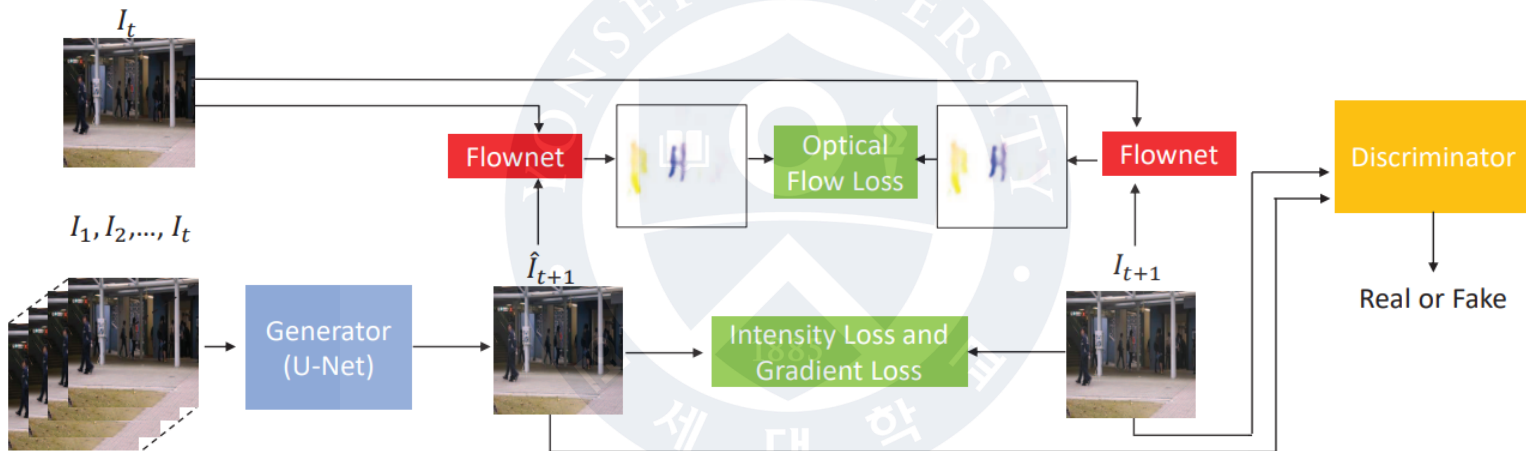
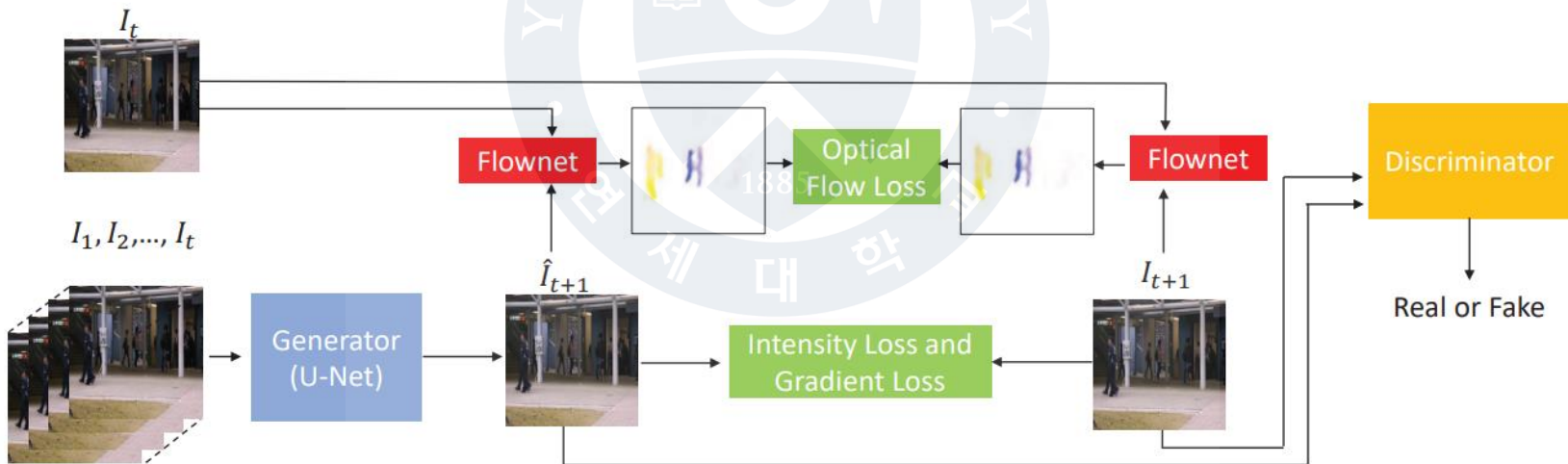


Figure 2. The pipeline of our video frame prediction network. Here we adopt U-Net as generator to predict next frame. To generate high quality image, we adopt the constraints in terms of appearance (intensity loss and gradient loss) and motion (optical flow loss). Here Flownet is a pretrained network used to calculate optical flow. We also leverage the adversarial training to discriminate whether the prediction is real or fake.

# 3 Future Frame Prediction

## Training and Testing

- I. Using the frames from 1 to  $t$  as input to the generator, the model learns to generate the  $(t+1)$ -th frame through Intensity Loss and Gradient Loss.
  - II. Generated  $(t+1)$ -th frame is forwarded into the Discriminator to learn to deceive the Discriminator. *[to make image sharply]*
  - III. Calculate the optical flow between consecutive frames and train to minimize the difference between the real optical flow and the fake optical flow. *[to make two consecutive frames appear smoothly]*
- Compute the Anomaly Score through PSNR between frames generated by the generator and real frames.



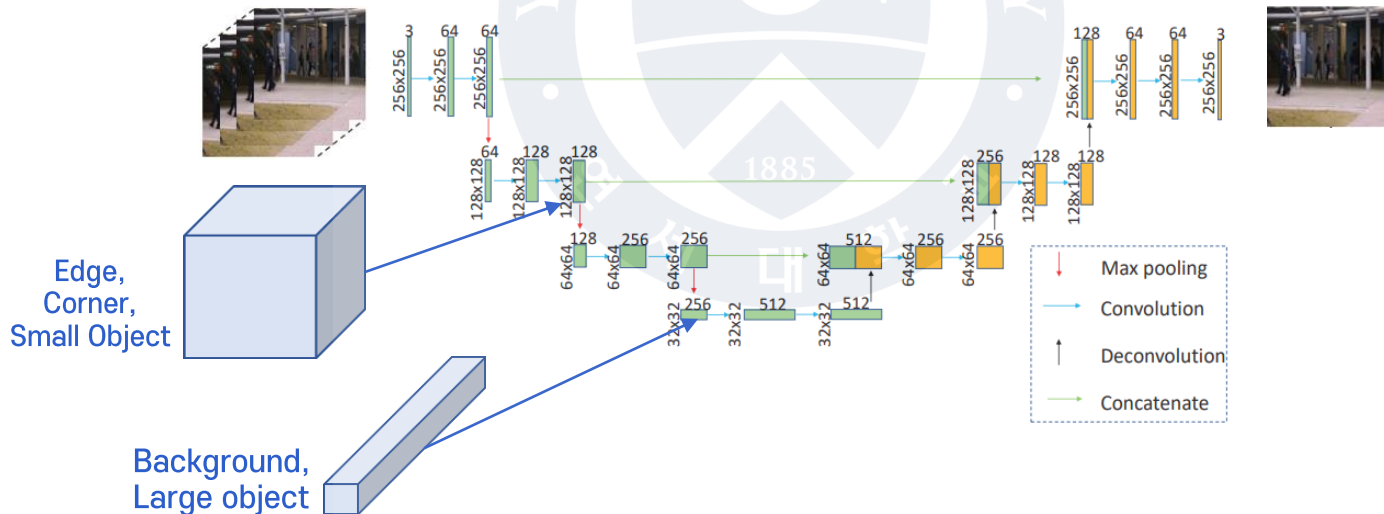
# 3 Future Frame Prediction

## Network

### Generator (U-Net)

A convolutional neural network in the form of a U-shape, composed of an encoder and a decoder.

- **Encoder:** Takes input frames from 1 to  $t$  to capture contextual information at various scales and transmits feature maps to the decoder through skip connections.
- **Decoder:** Utilizes the received feature maps to generate the detailed  $(t+1)$ -th frame.



# 3 Future Frame Prediction



## Network

### Generator (U-Net)

```
class UNet(nn.Module):
    def __init__(self, input_channels, output_channel=3):
        super(UNet, self).__init__()
        self.inc = inconv(input_channels, 64)
        self.down1 = down(64, 128)
        self.down2 = down(128, 256)
        self.down3 = down(256, 512)
        self.up1 = up(512, 256)
        self.up2 = up(256, 128)
        self.up3 = up(128, 64)
        self.outc = nn.Conv2d(64, output_channel, kernel_size=3, padding=1)

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x = self.up1(x4, x3)
        x = self.up2(x, x2)
        x = self.up3(x, x1)
        x = self.outc(x)

    return torch.tanh(x)
```

```
class inconv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super().__init__()
        self.conv = double_conv(in_ch, out_ch)
```

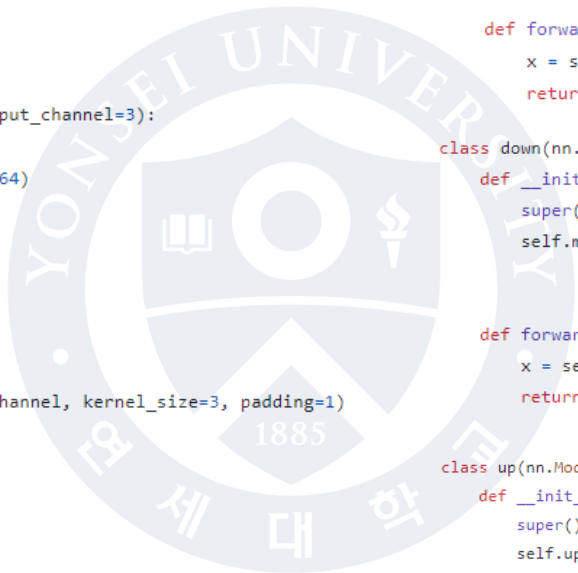
```
def forward(self, x):
    x = self.conv(x)
    return x
```

```
class down(nn.Module):
    def __init__(self, in_ch, out_ch):
        super().__init__()
        self.mpconv = nn.Sequential(nn.MaxPool2d(2),
                                     double_conv(in_ch, out_ch))
```

```
def forward(self, x):
    x = self.mpconv(x)
    return x
```

```
class up(nn.Module):
    def __init__(self, in_ch, out_ch):
        super().__init__()
        self.up = nn.ConvTranspose2d(in_ch, in_ch // 2, 2, stride=2)
        self.conv = double_conv(in_ch, out_ch)
```

```
def forward(self, x1, x2):
    x1 = self.up(x1)
    x = torch.cat([x2, x1], dim=1)
    x = self.conv(x)
    return x
```



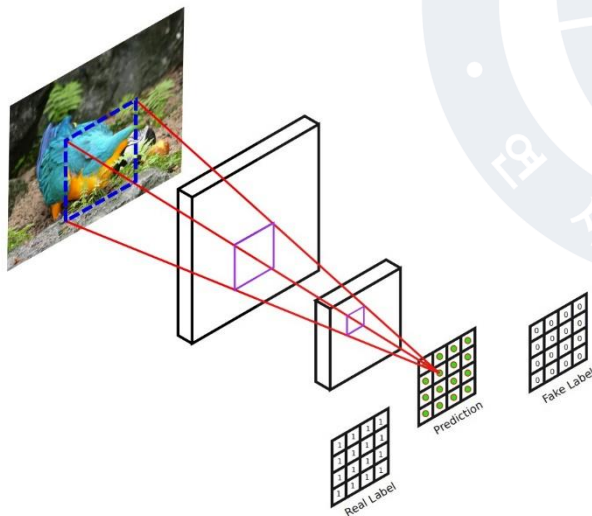
# 3 Future Frame Prediction

## Network

### Discriminator (PatchGAN)

The **size of patch (receptive field)** significantly impact the performance of the generative model. Therefore, it is crucial to train a GAN with an appropriate patch configuration.

If the receptive field is  $1 \times 1$ , each pixel is generated independently, leading to a diminished expressive power for representing continuous objects. If the receptive field is  $286 \times 286$ , only the global information of the image is emphasized, leading to a lack of detail in the image.



Discriminator receptive field	Per-pixel acc.	Per-class acc.	Class IOU
$1 \times 1$	0.39	0.15	0.10
$16 \times 16$	0.65	0.21	<b>0.17</b>
$70 \times 70$	<b>0.66</b>	<b>0.23</b>	<b>0.17</b>
$286 \times 286$	0.42	0.16	0.11



# 3 Future Frame Prediction

## Network

### Discriminator (PatchGAN)

```
class PixelDiscriminator(nn.Module):
    """Defines a PatchGAN discriminator (pixelGAN)"""

    def __init__(self, input_nc, num_filters=(128, 256, 512, 512)):
        """Construct a PatchGAN discriminator

        Parameters:
            input_nc (int) -- the number of channels in input images
            num_filters (int) -- the number of filters in the conv layer
        """
        super().__init__()

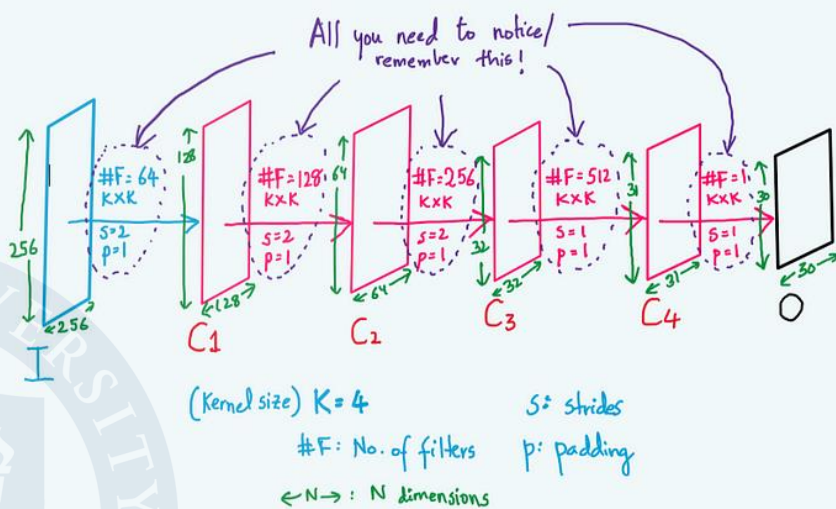
        self.conv1 = nn.Sequential(nn.Conv2d(input_nc, num_filters[0], kernel_size=4, padding=1, stride=2, bias=True),
                                   nn.LeakyReLU(0.2, True))

        self.conv2 = nn.Sequential(nn.Conv2d(num_filters[0], num_filters[1], kernel_size=4, padding=1, stride=2, bias=True),
                                   nn.LeakyReLU(0.2, True))

        self.conv3 = nn.Sequential(nn.Conv2d(num_filters[1], num_filters[2], kernel_size=4, padding=1, stride=2, bias=True),
                                   nn.LeakyReLU(0.2, True))

        self.conv4 = nn.Sequential(nn.Conv2d(num_filters[2], num_filters[3], kernel_size=4, padding=1, stride=1, bias=True),
                                   nn.LeakyReLU(0.2, True))

        self.out_conv = nn.Sequential(nn.Conv2d(num_filters[3], 1, kernel_size=4, padding=1, stride=1, bias=True))
```



```
def forward(self, x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = self.conv3(x)
    x = self.conv4(x)
    x = self.out_conv(x)
    out = torch.sigmoid(x)
    return out
```



# 3 Future Frame Prediction

## Network

### Discriminator (PatchGAN)

$$1) K_l = 1 \times 1 \leftarrow 0 \text{ layer}$$

$$K = 4 \times 4$$

$$S = 1 \times 1$$

$$K_{l-1} = \left( \frac{4 \times 4}{r \ c} \right) + \left( \frac{1 \times 1}{r \ c} \right) \left( \frac{1 \times 1}{r \ c} - 1 \right)$$

$$= 4 \times 4 \leftarrow C4 \text{ layer}$$

$$2) K_l = 4 \times 4 \leftarrow C4 \text{ layer}$$

$$K = 4 \times 4$$

$$S = 1 \times 1$$

$$K_{l-1} = \left( \frac{4 \times 4}{r \ c} \right) + \left( \frac{1 \times 1}{r \ c} \right) \left( \frac{4 \times 4}{r \ c} - 1 \right)$$

$$= 7 \times 7 \leftarrow C3 \text{ layer}$$

$$3) K_l = 7 \times 7 \leftarrow C3 \text{ layer}$$

$$K = 4 \times 4$$

$$S = 2 \times 2$$

$$K_{l-1} = \left( \frac{4 \times 4}{r \ c} \right) + \left( \frac{2 \times 2}{r \ c} \right) \left( \frac{7 \times 7}{r \ c} - 1 \right)$$

$$= 16 \times 16 \leftarrow C2 \text{ layer}$$

$$4) K_l = 16 \times 16 \leftarrow C2 \text{ layer}$$

$$K = 4 \times 4$$

$$S = 2 \times 2$$

$$K_{l-1} = \left( \frac{4 \times 4}{r \ c} \right) + \left( \frac{2 \times 2}{r \ c} \right) \left( \frac{16 \times 16}{r \ c} - 1 \right)$$

$$= 34 \times 34 \leftarrow C1 \text{ layer}$$

$$5) K_l = 34 \times 34 \leftarrow C1 \text{ layer}$$

$$K = 4 \times 4$$

$$S = 2 \times 2$$

$$K_{l-1} = \left( \frac{4 \times 4}{r \ c} \right) + \left( \frac{2 \times 2}{r \ c} \right) \left( \frac{34 \times 34}{r \ c} - 1 \right)$$

$$= 70 \times 70 \leftarrow I \text{ layer}$$

Formula

$$K_{l-1} = K + S (K_l - 1)$$

$\uparrow$  Receptive field of  $(l-1)^{\text{th}}$  layer  
 $\uparrow$  Kernel size  
 $\uparrow$  Stride value  
 $\uparrow$  Output receptive field of  $l^{\text{th}}$  layer

# 3 Future Frame Prediction

## Network

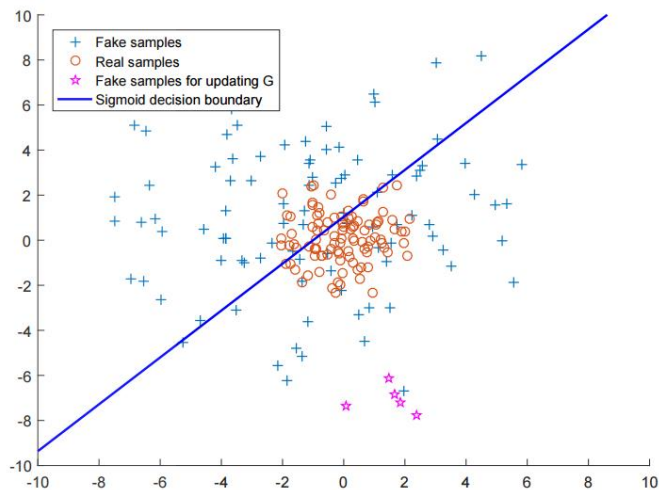
### Least Squares Generative Adversarial Networks (LSGAN)

$$\min_G \max_D V_{\text{GAN}}(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))] \quad (1)$$

$$-(y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

$$\min - 1 \log D(\mathbf{x}) \quad \min - (1 - 0) \log(1 - D(G(\mathbf{z})))$$

$$\min - 1 \log D(G(\mathbf{z}))$$



```
crit = nn.BCELoss()
# train D
loss = crit(D(x), 1) + crit(D(G(z)), 0)
loss.backward()
d_optimizer.step()

# train G
loss = crit(D(G(z)), 1)
loss.backward()
g_optimizer.step()
```

# 3 Future Frame Prediction

## Network

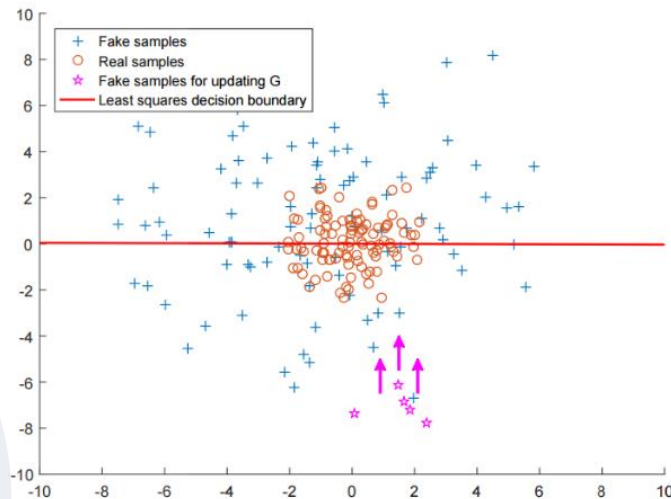
### Least Squares Generative Adversarial Networks (LSGAN)

$$\min_G \max_D V_{\text{GAN}}(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))] \quad (1)$$



$$\min_D V_{\text{LSGAN}}(D) = \frac{1}{2} \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [(D(\mathbf{x}) - b)^2] + \frac{1}{2} \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [(D(G(\mathbf{z})) - a)^2] \quad (2)$$

$$\min_G V_{\text{LSGAN}}(G) = \frac{1}{2} \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [(D(G(\mathbf{z})) - c)^2],$$



- a: Fake label (0)
- b: Real label (1)
- c: The label to deceive D (1)

# Loss of D

```
D_loss = torch.mean((D(x) - 1)**2) + torch.mean(D(G(z))**2)
```

# Loss of G

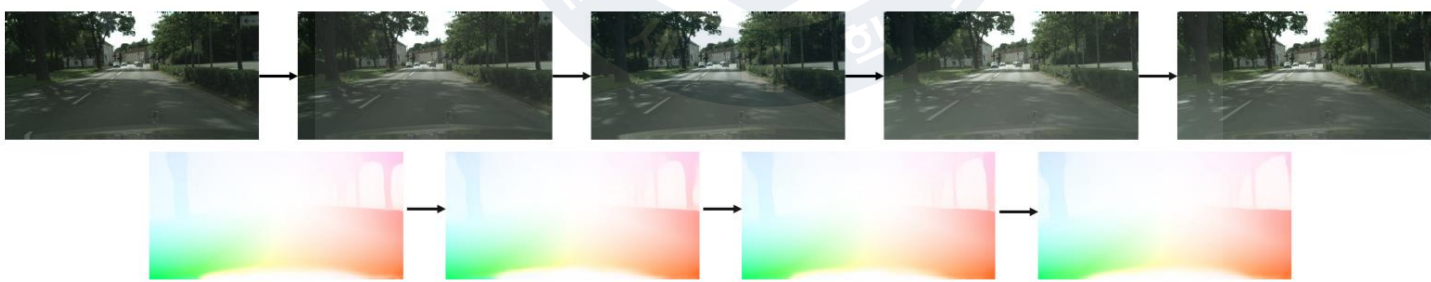
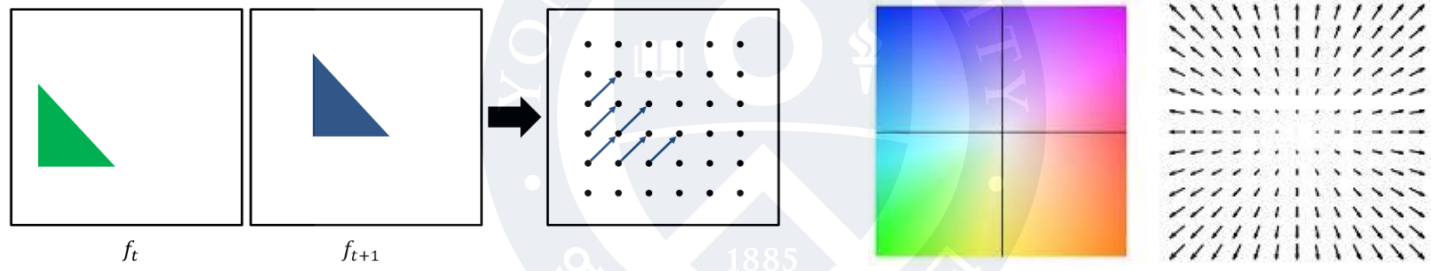
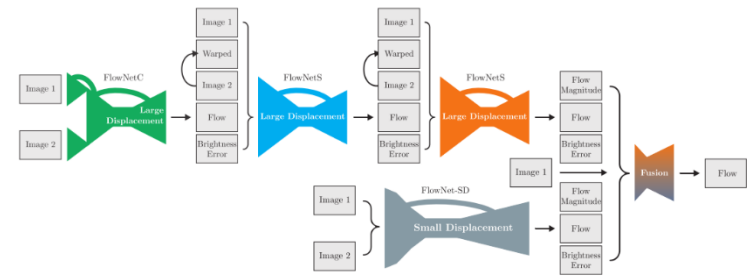
```
G_loss = torch.mean((D(G(z)) - 1)**2)
```

# 3 Future Frame Prediction

## Network

### FlowNet

A deep learning model that generates **FlowMap**, an image that represents the change in the x-direction and the change in the y-direction between the two frames per pixel.



# 3 Future Frame Prediction



## Network

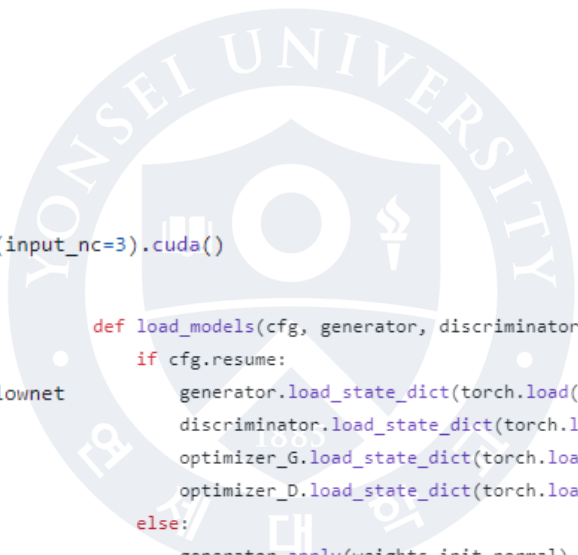
### FlowNet (Pretrained Model)

```
def def_models():
    # generator
    generator = UNet(12,3).cuda()
    # discriminator
    discriminator = PixelDiscriminator(input_nc=3).cuda()
    # flownet
    flownet = FlowNet2SD().cuda()

    return generator, discriminator, flownet

def load_models(cfg, generator, discriminator, flownet, optimizer_G, optimizer_D):
    if cfg.resume:
        generator.load_state_dict(torch.load(cfg.resume)['net_g'])
        discriminator.load_state_dict(torch.load(cfg.resume)['net_d'])
        optimizer_G.load_state_dict(torch.load(cfg.resume)['optimizer_g'])
        optimizer_D.load_state_dict(torch.load(cfg.resume)['optimizer_d'])
    else:
        generator.apply(weights_init_normal)
        discriminator.apply(weights_init_normal)

    # flownet
    flownet.load_state_dict(torch.load('flownet/pretrained/FlowNet2-SD.pth')['state_dict'])
    flownet.eval()
```



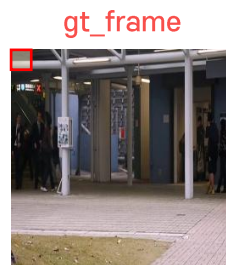
# 3 Future Frame Prediction

## Loss Function

Intensity loss: increase similarity between pixels.

$$L_{int}(\hat{I}, I) = \|\hat{I} - I\|_2^2$$

```
class Intensity_Loss(nn.Module):  
    def __init__(self):  
        super().__init__()  
  
    def forward(self, gen_frames, gt_frames):  
        return torch.mean(torch.abs((gen_frames - gt_frames) ** 2))
```



# 3 Future Frame Prediction

$$L_{gd}(\hat{I}, I) = \sum_{i,j} \left( \left| \hat{I}_{i,j} - \hat{I}_{i-1,j} \right| - \left| I_{i,j} - I_{i-1,j} \right| \right)_1 + \left( \left| \hat{I}_{i,j} - \hat{I}_{i,j-1} \right| - \left| I_{i,j} - I_{i,j-1} \right| \right)_1$$

where  $i, j$  denote the spatial index of a frame

## Loss Function

Gradient loss: make the differences between consecutive pairs of pixels similar, thereby smoothing out the generated images.

```
class Gradient_Loss(nn.Module):
    def __init__(self, channels):
        super().__init__()

        pos = torch.from_numpy(np.identity(channels, dtype=np.float32))
        neg = -1 * pos
        # Note: when doing conv2d, the channel order is different from tensorflow, so do permutation.
        self.filter_x = torch.stack((neg, pos)).unsqueeze(0).permute(3, 2, 0, 1).cuda()
        self.filter_y = torch.stack((pos.unsqueeze(0), neg.unsqueeze(0))).permute(3, 2, 0, 1).cuda()

    def forward(self, gen_frames, gt_frames):
        # Do padding to match the result of the original tensorflow implementation
        gen_frames_x = nn.functional.pad(gen_frames, [0, 1, 0, 0])
        gen_frames_y = nn.functional.pad(gen_frames, [0, 0, 0, 1])
        gt_frames_x = nn.functional.pad(gt_frames, [0, 1, 0, 0])
        gt_frames_y = nn.functional.pad(gt_frames, [0, 0, 0, 1])

        gen_dx = torch.abs(nn.functional.conv2d(gen_frames_x, self.filter_x))
        gen_dy = torch.abs(nn.functional.conv2d(gen_frames_y, self.filter_y))
        gt_dx = torch.abs(nn.functional.conv2d(gt_frames_x, self.filter_x))
        gt_dy = torch.abs(nn.functional.conv2d(gt_frames_y, self.filter_y))

        grad_diff_x = torch.abs(gt_dx - gen_dx)
        grad_diff_y = torch.abs(gt_dy - gen_dy)

        return torch.mean(grad_diff_x + grad_diff_y)
```

gen\_frame

```
tensor([[[[1., 2., 3.],
          [4., 5., 6.],
          [7., 8., 9.]]]]])
```

gt\_frame

```
tensor([[[[2., 4., 6.],
          [6., 8., 9.],
          [3., 2., 1.]]]]])
```

```
tensor([[[[-1., 1.]]]], device='cuda:0')
tensor([[[[ 1.,
          [-1.]]]]], device='cuda:0')
```

filter\_x,  
filter\_y

```
tensor([[[[1., 2., 3., 0.],
          [4., 5., 6., 0.],
          [7., 8., 9., 0.]]]], device='cuda:0')
tensor([[[[1., 2., 3.],
          [4., 5., 6.],
          [7., 8., 9.],
          [0., 0., 0.]]]]], device='cuda:0')
```

gen\_frame\_x,  
gen\_frame\_y

```
tensor([[[[1., 1., 3.],
          [1., 1., 6.],
          [1., 1., 9.]]]], device='cuda:0')
tensor([[[[3., 3., 3.],
          [3., 3., 3.],
          [7., 8., 9.]]]], device='cuda:0')
```

gen\_dx,  
gen\_dy

# 3 Future Frame Prediction

## Loss Function

- Discriminator adversarial loss: distinguish real and fake future frame.

$$L_{adv}^D(\hat{I}, I) = \sum_{i,j} \frac{1}{2} L_{MSE}(D(I)_{i,j}, 1) + \sum_{i,j} \frac{1}{2} L_{MSE}(D(\hat{I})_{i,j}, 0)$$

```
class Discriminate_Loss(nn.Module):  
    def __init__(self):  
        super().__init__()  
  
    def forward(self, real_outputs, fake_outputs):  
        return torch.mean((real_outputs - 1) ** 2 / 2) + torch.mean(fake_outputs ** 2 / 2)
```





# 3 Future Frame Prediction

## Loss Function

- Generator adversarial loss: predict future frames well enough to deceive Discriminator

$$L_{adv}^G(\hat{f}) = \sum_{i,j} \frac{1}{2} L_{MSE}(D(\hat{f})_{i,j}, 1)$$

```
class Adversarial_Loss(nn.Module):  
    def __init__(self):  
        super().__init__()  
  
    def forward(self, fake_outputs):  
        # TODO: compare with torch.nn.MSELoss ?  
        return torch.mean((fake_outputs - 1) ** 2 / 2)
```

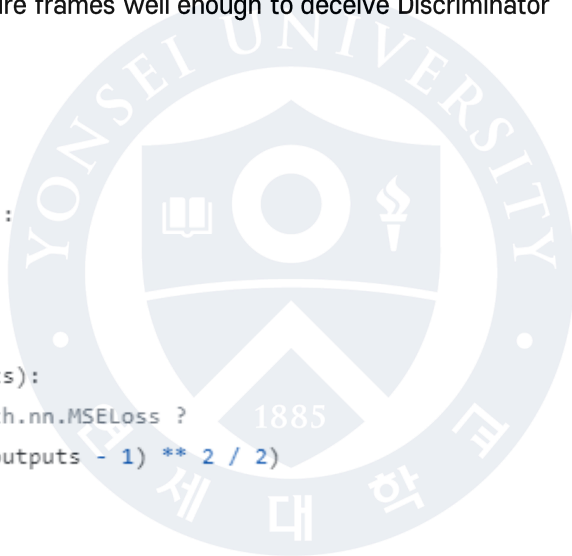
Generator  
(U-Net)

gen\_frame



Discriminator

Real



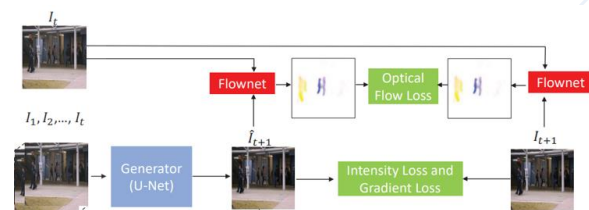
# 3 Future Frame Prediction

## Loss Function

- Optical Flow loss: minimize the difference between the real the fake optical flow, thereby making two consecutive frames appear smoothly.

$$L_{op}(\hat{I}_{t+1}, I_{t+1}, I_t) = \|f(\hat{I}_{t+1}, I_t) - f(I_{t+1}, I_t)\|_1$$

```
class Flow_Loss(nn.Module):  
    def __init__(self):  
        super().__init__()  
  
    def forward(self, gen_flows, gt_flows):  
        return torch.mean(torch.abs(gen_flows - gt_flows))
```



# 3 Future Frame Prediction

## Loss Function

### 3.5. Objective Function

We combine all these constraints regarding appearance, motion, and adversarial training, into our objective function, and arrive at the following objective function:

$$L_G = \lambda_{int} L_{int}(\hat{I}_{t+1}, I_{t+1}) + \lambda_{gd} L_{gd}(\hat{I}_{t+1}, I_{t+1}) + \lambda_{op} L_{op}(\hat{I}_{t+1}, I_{t+1}, I_t) + \lambda_{adv} L_{adv}^G(\hat{I}_{t+1}) \quad (7)$$

When we train  $\mathcal{D}$ , we use the following loss function:

$$L_{\mathcal{D}} = L_{adv}^D(\hat{I}_{t+1}, I_{t+1}) \quad (8)$$

To train the network, the intensity of pixels in all frames are normalized to  $[-1, 1]$  and the size of each frame is resized to  $256 \times 256$ . Similar to [27], we set  $t = 4$  and randomly clip 5 sequential frames. Adam [19] based Stochastic Gradient Descent method is used for parameter optimization. The mini-batch size is 4. For gray scale datasets, the learning rate of generator and discriminator are set to 0.0001 and 0.00001, separately. While for color scale datasets, they start from 0.0002 and 0.00002, respectively.  $\lambda_{int}$ ,  $\lambda_{gd}$ ,  $\lambda_{op}$  and  $\lambda_{adv}$  slightly vary from datasets and an easy way is to set them as 1.0, 2.0 and 0.05, respectively.

```
discriminate_loss = losses['discriminate_loss']
intensity_loss = losses['intensity_loss']
gradient_loss = losses['gradient_loss']
adversarial_loss = losses['adversarial_loss']
flow_loss = losses['flow_loss']

coefs = [1, 1, 0.05, 2] # inte_l, grad_l, adv_l, flow_l

# future frame prediction and get loss
pred = generator(input)
inte_l = intensity_loss(pred, target)
grad_l = gradient_loss(pred, target)
adv_l = adversarial_loss(discriminator(pred))

# flowmap prediction and get loss
gt_flow_input = torch.cat([input_last.unsqueeze(2), target.unsqueeze(2)], 2)
pred_flow_input = torch.cat([input_last.unsqueeze(2), pred.unsqueeze(2)], 2)
flow_gt = (flownet(gt_flow_input * 255.) / 255.).detach() # Input for flownet2sd is in (0, 255).
flow_pred = (flownet(pred_flow_input * 255.) / 255.).detach()
flow_l = flow_loss(flow_pred, flow_gt)

loss_gen = coefs[0] * inte_l + \
           coefs[1] * grad_l + \
           coefs[2] * adv_l + \
           coefs[3] * flow_l

# discriminator
loss_dis = discriminate_loss(discriminator(target),
                             discriminator(pred.detach()))
```

(train\_func.py)

# 3 Future Frame Prediction

gen\_frame

gt\_frame



PSNR

## Testing

### Peak Signal-to-noise ratio (PSNR)

Represents the [quality of an image](#) by normalizing Mean Squared Error (MSE) to pixel information scale; higher values indicate better quality.

$$PSNR(I, \hat{I}) = 10 \log_{10} \frac{[\max_{\hat{I}}]^2}{\frac{1}{N} \sum_{i=0}^N (I_i - \hat{I}_i)^2}$$

```
def psnr_error(gen_frames, gt_frames):  
    """  
    Computes the Peak Signal to Noise Ratio error between the generated images and the ground  
    truth images.  
    @param gen_frames: A tensor of shape [batch_size, 3, height, width]. The frames generated by the  
    generator model.  
    @param gt_frames: A tensor of shape [batch_size, 3, height, width]. The ground-truth frames for  
    each frame in gen_frames.  
    @return: A scalar tensor. The mean Peak Signal to Noise Ratio error over each frame in the  
    batch.  
    """  
    shape = list(gen_frames.shape)  
    num_pixels = (shape[1] * shape[2] * shape[3])  
    gt_frames = (gt_frames + 1.0) / 2.0 # if the generate ouput is sigmoid output, modify here.  
    gen_frames = (gen_frames + 1.0) / 2.0  
    square_diff = (gt_frames - gen_frames) ** 2  
    batch_errors = 10 * log10(1. / ((1. / num_pixels) * torch.sum(square_diff, [1, 2, 3])))  
    return torch.mean(batch_errors)
```

JPEG



Original image

PSNR 34.8227dB

PSNR 30.9394dB

PSNR 25.8699dB

# 3 Future Frame Prediction

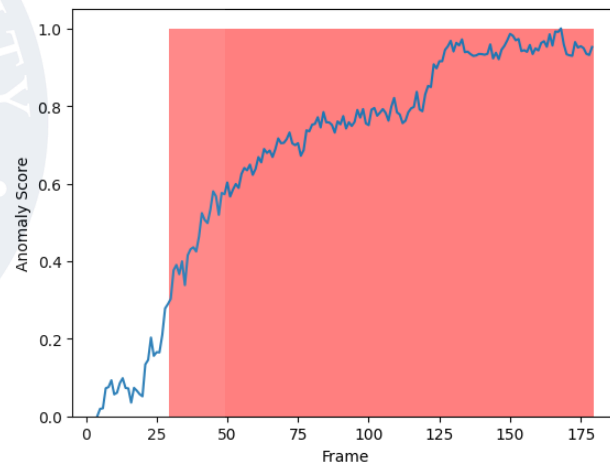
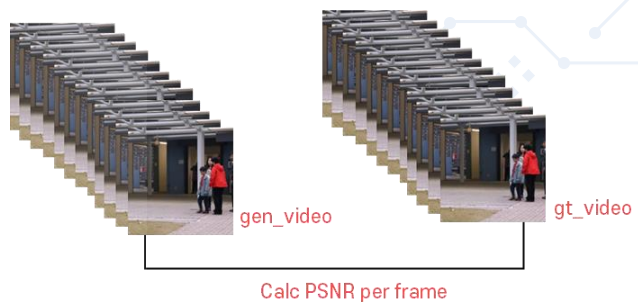
## Testing

### Regular Score

The PSNR is min-max normalized to calculate the Regular Score, also known as the Anomaly Score.

$$S(t) = 1 - \frac{PSNR(I_t, \hat{I}_t) - \min PSNR(I_t, \hat{I}_t)}{\max PSNR(I_t, \hat{I}_t) - \min PSNR(I_t, \hat{I}_t)}$$

```
def min_max_normalize(arr, eps=1e-8):  
    min_val = np.min(arr)  
    max_val = np.max(arr)  
    denominator = max_val - min_val + eps # Avoid division by zero  
  
    normalized_arr = (arr - min_val) / denominator  
    return normalized_arr
```



# 3 Future Frame Prediction

## Testing

### Area Under the ROC Curve (AUROC) [1]

AUROC is an [evaluation metric for anomaly detection](#), and it is calculated based on the [Anomaly Score](#).

```
import numpy as np
from sklearn import metrics
import matplotlib.pyplot as plt
```

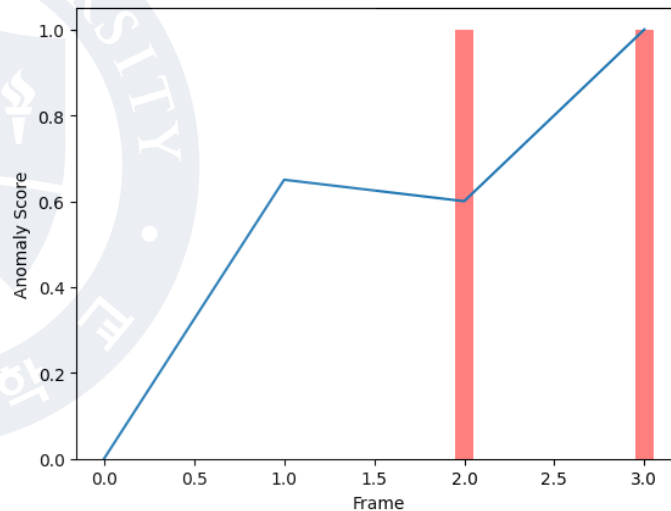
```
y = np.array([0, 0, 1, 1])
scores = np.array([0, 0.65, 0.6, 1.0])
```

```
length = len(scores)
anomalies_idx = [i for i, l in enumerate(y) if l==1]
```

```
print("length:", length)
print("anomalies_idx", anomalies_idx)
```

```
length: 4
anomalies_idx [2, 3]
```

```
plt.clf()
plt.plot([num for num in range(length)], [score for score in scores]) # plotting
plt.bar(anomalies_idx, max(scores), width=0.1, color='r', alpha=0.5) # check answer
plt.xlabel("Frame")
plt.ylabel("Anomaly Score")
```



This is an example where abnormal is considered positive(1) and normal is considered negative(0).

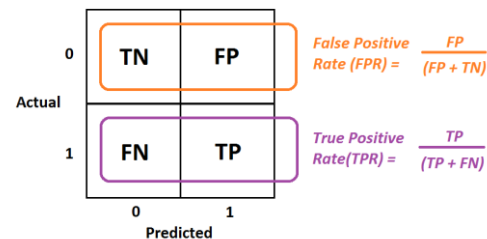
# 3 Future Frame Prediction

## Testing

### Area Under the ROC Curve (AUROC) [2]

The ROC curve is a graph that records True Positive Rate (TPR) and False Positive Rate (FPR) while changing the threshold.

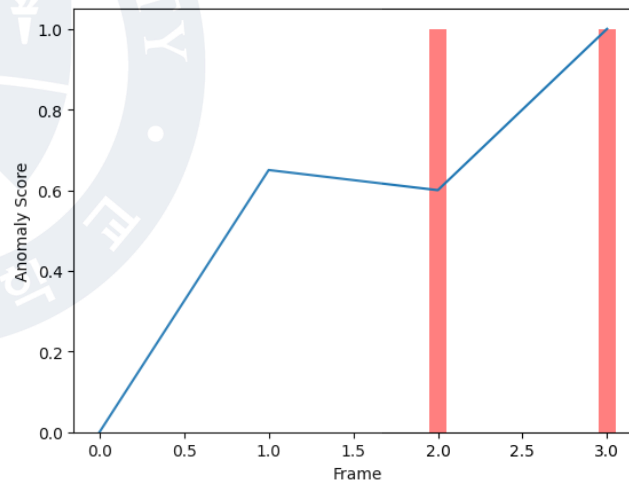
- **TPR (True Positive Rate):** The proportion of data correctly predicted as positive in actual positive data.
- **FPR (False Positive Rate):** The proportion of data incorrectly predicted as positive in actual negative data.



```
fpr, tpr, thresholds = metrics.roc_curve(y, scores, pos_label=1)
auc = metrics.auc(fpr, tpr)

print("thresholds", thresholds)
print("tpr:", tpr)
print("fpr", fpr)
print("auc", auc)
```

```
thresholds [2.  1.  0.65 0.6  0. ]
tpr: [0.  0.5 0.5 1.  1. ]
fpr [0.  0.  0.5 0.5 1. ]
auc 0.75
```



This is an example where abnormal is considered positive(1) and normal is considered negative(0).

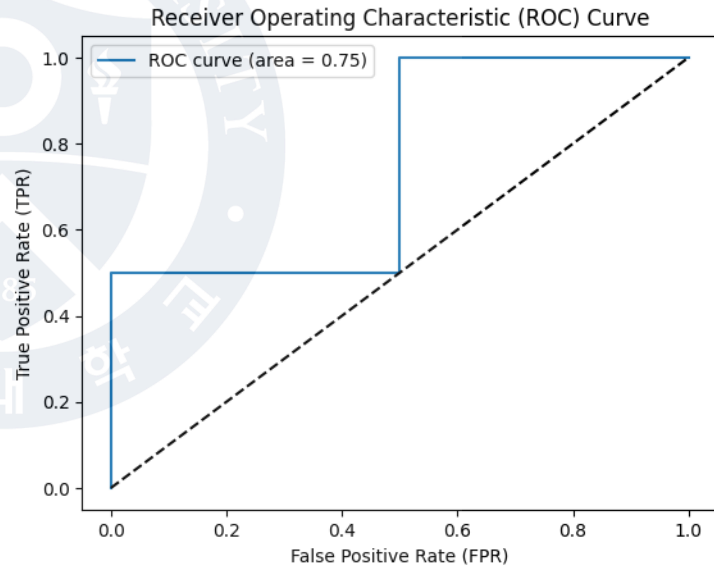
# 3 Future Frame Prediction

## Testing

### Area Under the ROC Curve (AUROC) [3]

The AUC (Area Under the Curve) of the ROC curve represents the overall performance of the model, with a higher AUC value indicating better performance, approaching 1.

```
plt.clf()
plt.plot(fpr, tpr, label=f'ROC curve (area = {auc:.2f})')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()
```



This is an example where abnormal is considered positive(1) and normal is considered negative(0).



# 3 Future Frame Prediction



## Experiments [Dataset]

### UCSD Ped2

UCSD Ped1



UCSD Ped2



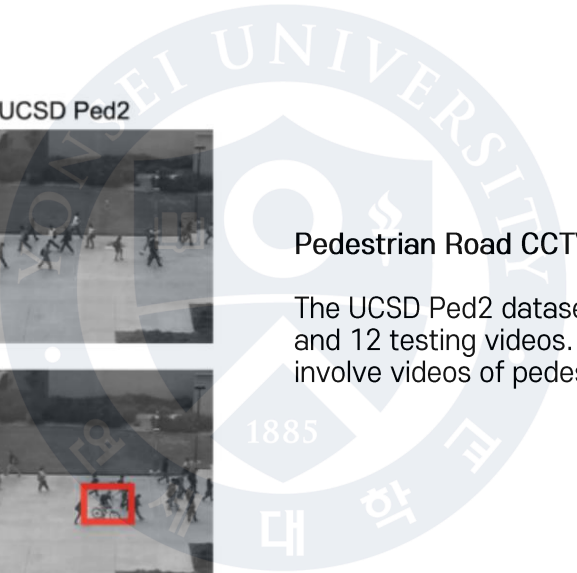
Biker



Biker

### Pedestrian Road CCTV Dataset

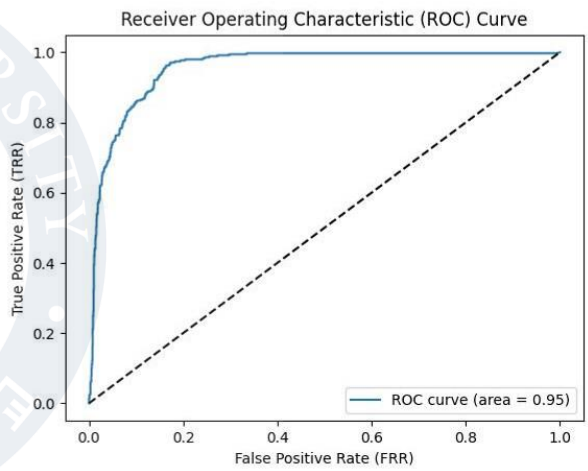
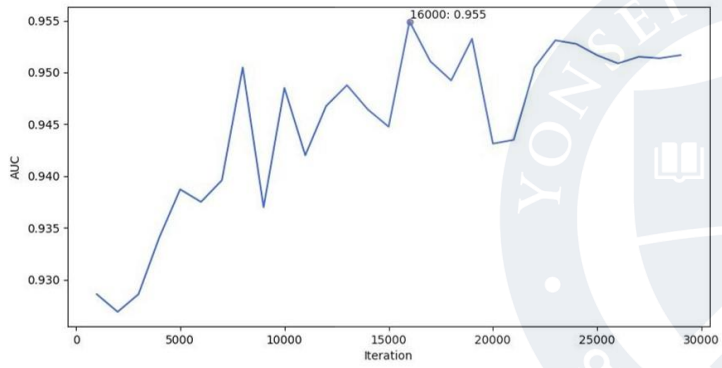
The UCSD Ped2 dataset consists of 16 training videos and 12 testing videos. Abnormal situations in this dataset involve videos of pedestrians or vehicles moving on roads.



# 3 Future Frame Prediction

## Experiments [Results]

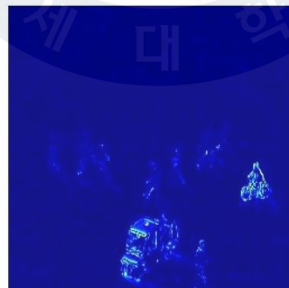
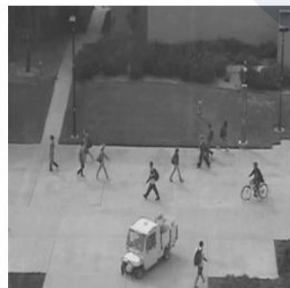
### AUC & Heatmap



Real frame

Fake frame

Heatmap



# 3 Future Frame Prediction



## Appendix

### Dataset loading

```
class train_dataset(Dataset):
    def __init__(self, cfg):
        self.img_h = cfg.img_size[0]
        self.img_w = cfg.img_size[1]
        self.clip_length = 5

        self.videos = []
        self.all_seqs = []

        for folder in sorted(glob.glob(f'{cfg.train_data}/*')):
            all_imgs = glob.glob(f'{folder}/*.jpg')
            all_imgs.sort()
            self.videos.append(all_imgs)

            random_seq = list(range(len(all_imgs) - 4))
            random.shuffle(random_seq)
            self.all_seqs.append(random_seq)

    def __len__(self): # This decide the indice range of the PyTorch Dataloader.
        return len(self.videos)

    def __getitem__(self, indice): # Indice decide which video folder to be loaded.
        one_folder = self.videos[indice]

        video_clip = []
        start = self.all_seqs[indice][-1] # Always use the last index in self.all_seqs.
        for i in range(start, start + self.clip_length):
            video_clip.append(np_load_frame(one_folder[i], self.img_h, self.img_w))

        video_clip = np.array(video_clip).reshape((-1, self.img_h, self.img_w))
        video_clip = torch.from_numpy(video_clip)

        return indice, video_clip
```

(dataset.py)

```
for indice, clips in dataloader:
    # define frame 1 to 4
    frame_1 = clips[:, 0:3, :, :].cuda() # (n, 3, 256, 256)
    frame_2 = clips[:, 3:6, :, :].cuda() # (n, 3, 256, 256)
    frame_3 = clips[:, 6:9, :, :].cuda() # (n, 3, 256, 256)
    frame_4 = clips[:, 9:12, :, :].cuda() # (n, 3, 256, 256)

    # pop() the used video index
    for index in indice:
        dataset.all_seqs[index].pop()
        if len(dataset.all_seqs[index]) == 0:
            dataset.all_seqs[index] = list(range(len(dataset.videos[index]) - 4))
            random.shuffle(dataset.all_seqs[index])
```

(train\_func.py)

# 3 Future Frame Prediction



## Appendix

### Anomaly Scoring

```
for i, folder in enumerate(video_folders):
    # Testing Log
    if not os.path.exists(f"results/{dataset_name}/{i}/f{i+1}"):
        os.makedirs(f"results/{dataset_name}/{i}/f{i+1}")
        os.makedirs(f"results/{dataset_name}/{i}/f{i+1}/real")
        os.makedirs(f"results/{dataset_name}/{i}/f{i+1}/fake")
        os.makedirs(f"results/{dataset_name}/{i}/f{i+1}/diff")

    one_video = dataset.test_dataset(cfg, folder)
    psnrs = []

    for j, clip in enumerate(one_video):
        frame_1 = clip[0:3, :, :].unsqueeze(0).cuda() # (1, 3, 256, 256)
        frame_2 = clip[3:6, :, :].unsqueeze(0).cuda() # (1, 3, 256, 256)
        frame_3 = clip[6:9, :, :].unsqueeze(0).cuda() # (1, 3, 256, 256)
        frame_4 = clip[9:12, :, :].unsqueeze(0).cuda() # (1, 3, 256, 256)

        input = torch.cat([frame_1, frame_2, frame_3, frame_4], 1).cuda() # (1, 12, 256, 256)
        target_frame = clip[12:15, :, :].unsqueeze(0).cuda() # (1, 3, 256, 256)

        F_frame = generator(input)
        test_psnr = psnr_error(F_frame, target_frame).cpu().detach().numpy()
        psnrs.append(float(test_psnr))

    gt_loader = Label_loader(cfg, video_folders) # Get gt labels.
    gt = gt_loader()
    assert len(psnr_group) == len(gt), f'Ground truth has {len(gt)} videos, but got {len(psnr_group)} detected videos.'

    scores = np.array([], dtype=np.float32)
    labels = np.array([], dtype=np.int8)

    for i in range(len(psnr_group)):
        psnrs = psnr_group[i]
        distance = min_max_normalize(psnrs)

        label = gt[i][4:]
        scores = np.concatenate((scores, distance), axis=0)
        labels = np.concatenate((labels, label), axis=0)

    # calc total auc
    fpr, tpr, _ = metrics.roc_curve(labels, scores, pos_label=0)
    auc = metrics.auc(fpr, tpr)
    print('auc:', auc)
```

(test\_eval.py)

# 3 Future Frame Prediction



## Github Code

The screenshot shows a GitHub repository page for 'Future Frame Prediction'. The repository is owned by 'SkiddieAhn' and is currently on the 'main' branch. The repository description is '[CVPR 2018] Future Frame Prediction for Anomaly Detection (pytorch implementation)'. The repository has 0 stars, 0 watching, and 1 fork. The repository is licensed under the MIT license. The repository contains the following files and folders:

File/Folder	File Type	Last Commit
data/ped2	edit files	last month
evaluation	edit files	last month
flownet	edit files	last month
model	edit files	last month
results	edit files	last month
training	edit files	last month
weights	edit files	last month
README.md	Update README.md	last month
config.py	edit files	last month
dataset.py	edit files	last month
eval.py	edit files	last month
train.py	edit files	last month
utils.py	edit files	last month

The repository also has a 'Releases' section with no releases published and a 'Packages' section with no packages published. The repository is primarily written in Python (79.1%) and Jupyter Notebook (20.9%).

The README.md file is displayed below, showing the title 'Future Frame Prediction' and the description: 'Pytorch implementation of video anomaly detection paper for CVPR 2018: [Future Frame Prediction for Anomaly Detection – A New Baseline.](#)'

Suggested Workflows are shown at the bottom, including a workflow for 'Pylint' with a 'Configure' button.

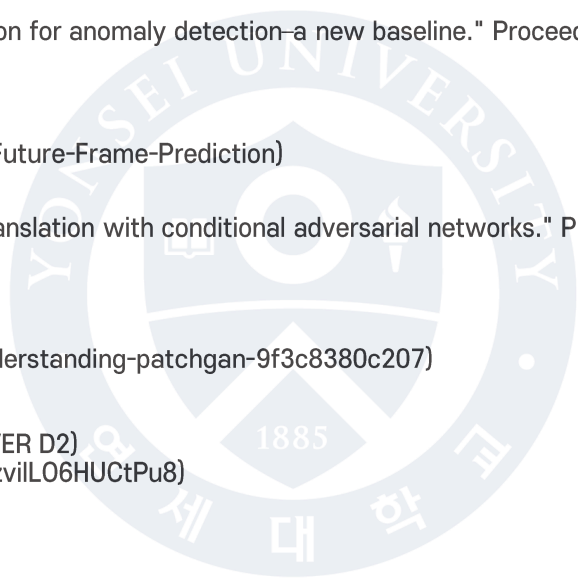
<https://github.com/ad-yonsei/Code-Future-Frame-Prediction>

# 3 Future Frame Prediction



## References

- ➡ Liu, Wen, et al. "Future frame prediction for anomaly detection—a new baseline." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.
- ➡ Implementing FFP in PyTorch  
(<https://github.com/ad-yonsei/Code-Future-Frame-Prediction>)
- ➡ Isola, Phillip, et al. "Image-to-image translation with conditional adversarial networks." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.
- ➡ Understanding PatchGAN  
(<https://sahiltinky94.medium.com/understanding-patchgan-9f3c8380c207>)
- ➡ Generative Adversarial Networks (NAVER D2)  
([https://youtu.be/odpjk7\\_tGY0?si=d3zviiLO6HUCtPu8](https://youtu.be/odpjk7_tGY0?si=d3zviiLO6HUCtPu8))





**Thank You**

Future Frame Prediction - coding practice

Sunghyun Ahn  
[skd@yonsei.ac.kr](mailto:skd@yonsei.ac.kr)

<2023/11/23>