In [1]:
```
import os
import sys
from tqdm import tqdm
import numpy as np
import torch
import torch.nn as nn
from torch import optim
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder
from torchvision.utils import make_grid
import torchvision.utils as vutils
import torchvision.transforms as T
import torchvision.transforms.functional as TF
import torch.nn.functional as F
from torchvision.utils import save_image
from PIL import Image
import matplotlib.pyplot as plt
%matplotlib inline
from paddleocr import PaddleOCR, draw_ocr
import paddle
from Levenshtein import distance as levenshtein_distance
from pytorch_msssim import ssim as ssim_fn
from lpips import LPIPS
from torchmetrics.image.psnr import PeakSignalNoiseRatio
from torchmetrics.image.ssim import StructuralSimilarityIndexMeasure
from torchmetrics.image.fid import FrechetInceptionDistance
```

In [2]:
```
sys.path.append(os.path.abspath('../../../src/dataset'))
from paired_image_dataset import PairedImageDataset

DATA_DIR = os.path.join('..', '..', '..', 'data')
print(os.listdir(DATA_DIR))
```

```
['h_blur', 'low_light', 'low_qual', 'test', 'v_blur']
```

In [3]:
```
image_width = 256
image_height = 128
batch_size = 16
stats = (0.5, 0.5, 0.5), (0.5, 0.5, 0.5)
```

In [4]:
```
train_transform = T.Compose([
    T.Resize((image_height, image_width)),
    T.ToTensor(),
    T.Normalize(*stats)])

valid_transform = T.Compose([
    T.Resize((image_height, image_width)),
    T.ToTensor(),
    T.Normalize(*stats)
])
```

In [14]:
```
train_ds = PairedImageDataset(blur_dir=os.path.join(DATA_DIR, 'h_blur', 'v2', 'trai
                              normal_dir=os.path.join(DATA_DIR, 'h_blur', 'v2', 'tr
                              transform=train_transform)
```

```
valid_ds = PairedImageDataset(blur_dir=os.path.join(DATA_DIR, 'h_blur', 'v2', 'vali
                              normal_dir=os.path.join(DATA_DIR, 'h_blur', 'v2', 'va
                              transform=valid_transform)

test_ds = ImageFolder(os.path.join(DATA_DIR, 'h_blur', 'v2', 'test v2'), transform=
```

In [15]:
```
train_dl = DataLoader(train_ds, batch_size=batch_size, shuffle=True, num_workers=4)
valid_dl = DataLoader(valid_ds, batch_size=batch_size*2, shuffle=True, num_workers=
test_dl = DataLoader(test_ds, batch_size=2, shuffle=False, num_workers=4)
```

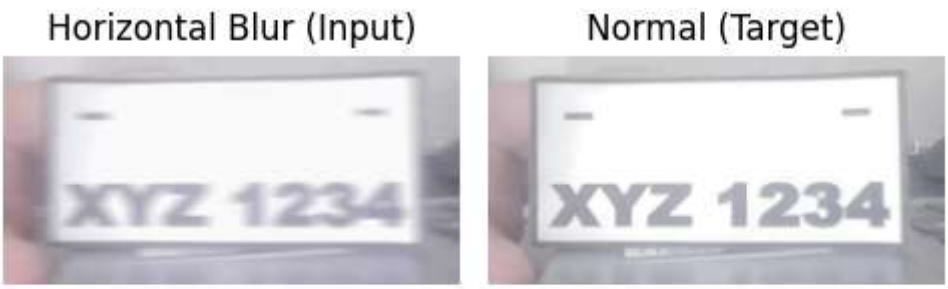
Helper functions

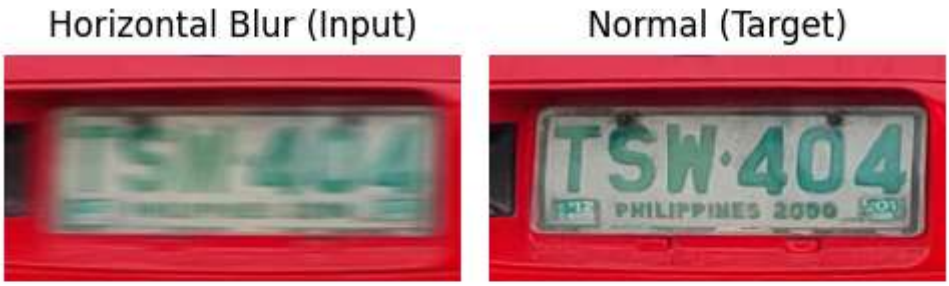
In [16]:
```
def denorm(img_tensors):
    return img_tensors * stats[1][0] + stats[0][0]

def show_images(input_images, target_images, nmax=16):
    input_images = denorm(input_images.detach()[:nmax])
    target_images = denorm(target_images.detach()[:nmax])

    # Combine into a single tensor for visualization
    combined = torch.cat((input_images, target_images), 0)
    grid = make_grid(combined, nrow=nmax)

    plt.figure(figsize=(nmax, 4))
    plt.imshow(grid.permute(1, 2, 0))
    plt.axis("off")
    plt.title("Top: Horizontal Blur / Blurred | Bottom: Normal / Target")
    plt.show()

def show_batch(dl, nmax=16):
    for input_batch, target_batch in dl:
        show_images(input_batch, target_batch, nmax)
        break
```

In [17]:
```
def show_paired_samples(dataset, num_samples=4):
    fig, axes = plt.subplots(num_samples, 2, figsize=(5, 2 * num_samples))

    for i in range(num_samples):
        input_img, target_img = dataset[i]

        input_img = denorm(input_img)
        target_img = denorm(target_img)

        if isinstance(input_img, torch.Tensor):
            input_img = input_img.permute(1, 2, 0).numpy()
            target_img = target_img.permute(1, 2, 0).numpy()

        axes[i, 0].imshow(input_img)
        axes[i, 0].set_title("Horizontal Blur (Input)")
        axes[i, 0].axis("off")

        axes[i, 1].imshow(target_img)
```

```
        axes[i, 1].set_title("Normal (Target)")
        axes[i, 1].axis("off")

    plt.tight_layout()
    plt.show()

def show_generated_images(images, nrow=4):
    images = denorm(images.detach().cpu())

    # Make grid
    grid_img = vutils.make_grid(images, nrow=nrow, normalize=False)

    # Show image
    plt.figure(figsize=(nrow * 2, 2 * (len(images) // nrow)))
    plt.axis('off')
    plt.imshow(grid_img.permute(1, 2, 0))
    plt.show()
```

In [18]:
```
show_paired_samples(train_ds)
```





In [19]:
```
show_batch(valid_dl, nmax=14)
```

In [20]:
```
def print_images(image_tensor, num_images):
    images = denorm(image_tensor)
    images = images.detach().cpu()
    image_grid = make_grid(images[:num_images], nrow=5)
```

```
    plt.imshow(image_grid.permute(1, 2, 0).squeeze())
    plt.show()
```

In [21]:
```
def save_samples(generator, test_dl, epoch, sample_dir='./generated_test_outputs'):
    generator.eval()
    os.makedirs(sample_dir, exist_ok=True)

    with torch.no_grad():
        global_idx = 1  # running index for image IDs

        for inputs, _ in test_dl:
            inputs = inputs.to(device)

            fake_images = generator(inputs)
            fake_images = fake_images.clamp(-1, 1)

            batch_size = fake_images.size(0)

            for b in range(batch_size):
                img = fake_images[b]

                # create folder per image
                folder_path = os.path.join(sample_dir, str(global_idx))
                os.makedirs(folder_path, exist_ok=True)

                file_name = f"{global_idx}_{epoch}epoch.jpg"
                save_path = os.path.join(folder_path, file_name)

                save_image(denorm(img).unsqueeze(0), save_path, nrow=1)
                global_idx += 1
```

In [22]:
```
def get_default_device():
    if torch.cuda.is_available():
        return torch.device('cuda')
    else:
        return torch.device('cpu')

def to_device(data, device):
    if isinstance(data, (list, tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)

class DeviceDataLoader():
    def __init__(self, dl, device):
        self.dl = dl
        self.device = device

    def __iter__(self):
        for b in self.dl:
            yield to_device(b, self.device)

    def __len__(self):
        return len(self.dl)
```

In [23]:
```
device = get_default_device()
device
```

Out[23]:
```
device(type='cuda')
```

In [24]:
```
train_dl = DeviceDataLoader(train_dl, device)
valid_dl = DeviceDataLoader(valid_dl, device)
test_dl = DeviceDataLoader(test_dl, device)
```

Building the Model

Generator

In [2]:
```
class ResConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=3, dropout=0.0, batch
        super().__init__()
        padding = kernel_size // 2

        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size, padding=padd
        self.bn1 = nn.BatchNorm2d(out_channels) if batch_norm else nn.Identity()

        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size, padding=pad
        self.bn2 = nn.BatchNorm2d(out_channels) if batch_norm else nn.Identity()

        self.drop = nn.Dropout(p=dropout) if dropout > 0 else nn.Identity()

        self.shortcut = nn.Conv2d(in_channels, out_channels, kernel_size=1, padding
        self.bn_sc = nn.BatchNorm2d(out_channels) if batch_norm else nn.Identity()

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = F.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.drop(out)

        shortcut = self.shortcut(residual)
        shortcut = self.bn_sc(shortcut)

        out += shortcut
        out = F.relu(out)
        return out


class GatingSignal(nn.Module):
    def __init__(self, in_channels, out_channels, batch_norm=False):
        super().__init__()
```

```
        layers = [nn.Conv2d(in_channels, out_channels, kernel_size=1, padding=0)]
        if batch_norm:
            layers.append(nn.BatchNorm2d(out_channels))
        layers.append(nn.ReLU(inplace=True))
        self.gate = nn.Sequential(*layers)

    def forward(self, x):
        return self.gate(x)

class AttentionBlock(nn.Module):
    def __init__(self, in_channels, gating_channels, inter_channels):
        super().__init__()

        self.theta_x = nn.Conv2d(in_channels, inter_channels, kernel_size=2, stride
        self.phi_g = nn.Conv2d(gating_channels, inter_channels, kernel_size=1, padd

        self.upsample_g = nn.ConvTranspose2d(inter_channels, inter_channels, kernel
        self.combine = nn.Sequential(
            nn.ReLU(inplace=True),
            nn.Conv2d(inter_channels, 1, kernel_size=1),
            nn.Sigmoid())

        self.final_conv = nn.Sequential(
            nn.Conv2d(in_channels, in_channels, kernel_size=1),
            nn.BatchNorm2d(in_channels))

    def forward(self, x, g):
        # x: skip connection feature map
        # g: gating signal (decoder feature map)
        theta_x = self.theta_x(x)
        phi_g = self.phi_g(g)

        # Upsample gating to match theta_x
        upsample_g = self.upsample_g(phi_g)

        if upsample_g.shape != theta_x.shape:
            upsample_g = F.interpolate(upsample_g, size=theta_x.shape[2:], mode='bi

        psi = self.combine(theta_x + upsample_g)
        psi = F.interpolate(psi, size=x.shape[2:], mode='bilinear', align_corners=T
        psi = psi.expand(-1, x.shape[1], -1, -1)

        y = x * psi
        return self.final_conv(y)


class AttentionResUNetGenerator(nn.Module):
    def __init__(self, in_channels=3, out_channels=3, base_filters=64, dropout=0.0,
        super().__init__()

        # Encoder
        self.down1 = ResConvBlock(in_channels, base_filters, dropout=dropout, batch
        self.pool1 = nn.MaxPool2d(2)
        self.down2 = ResConvBlock(base_filters, base_filters * 2, dropout=dropout,
        self.pool2 = nn.MaxPool2d(2)
        self.down3 = ResConvBlock(base_filters * 2, base_filters * 4, dropout=dropo
```

```
        self.pool3 = nn.MaxPool2d(2)
        self.down4 = ResConvBlock(base_filters * 4, base_filters * 8, dropout=dropo
        self.pool4 = nn.MaxPool2d(2)

        # Bottleneck
        self.bottleneck = ResConvBlock(base_filters * 8, base_filters * 16, dropout

        # Gating and Attention
        self.gate4 = GatingSignal(base_filters * 16, base_filters * 8, batch_norm)
        self.attn4 = AttentionBlock(base_filters * 8, base_filters * 8, base_filter

        self.gate3 = GatingSignal(base_filters * 8, base_filters * 4, batch_norm) #
        self.attn3 = AttentionBlock(base_filters * 4, base_filters * 4, base_filter

        self.gate2 = GatingSignal(base_filters * 4, base_filters * 2, batch_norm) #
        self.attn2 = AttentionBlock(base_filters * 2, base_filters * 2, base_filter

        self.gate1 = GatingSignal(base_filters * 2, base_filters, batch_norm) # 128
        self.attn1 = AttentionBlock(base_filters, base_filters, base_filters) # 64

        # Decoder
        self.up4 = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
        self.dec4 = ResConvBlock(base_filters * 16 + base_filters * 8, base_filters

        self.up3 = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
        self.dec3 = ResConvBlock(base_filters * 8 + base_filters * 4, base_filters

        self.up2 = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
        self.dec2 = ResConvBlock(base_filters * 4 + base_filters * 2, base_filters

        self.up1 = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
        self.dec1 = ResConvBlock(base_filters * 2 + base_filters, base_filters, dro

        # Output Layer
        self.final_conv = nn.Sequential(
            nn.Conv2d(base_filters, out_channels, kernel_size=1),
            nn.Tanh())

    def forward(self, x):
        d1 = self.down1(x)
        p1 = self.pool1(d1)
        d2 = self.down2(p1)
        p2 = self.pool2(d2)
        d3 = self.down3(p2)
        p3 = self.pool3(d3)
        d4 = self.down4(p3)
        p4 = self.pool4(d4)

        bn = self.bottleneck(p4)

        g4 = self.gate4(bn)
        a4 = self.attn4(d4, g4)
        u4 = self.up4(bn)
        u4 = torch.cat([u4, a4], dim=1)
        d5 = self.dec4(u4)
```

```
        g3 = self.gate3(d5)
        a3 = self.attn3(d3, g3)
        u3 = self.up3(d5)
        u3 = torch.cat([u3, a3], dim=1)
        d6 = self.dec3(u3)

        g2 = self.gate2(d6)
        a2 = self.attn2(d2, g2)
        u2 = self.up2(d6)
        u2 = torch.cat([u2, a2], dim=1)
        d7 = self.dec2(u2)

        g1 = self.gate1(d7)
        a1 = self.attn1(d1, g1)
        u1 = self.up1(d7)
        u1 = torch.cat([u1, a1], dim=1)
        d8 = self.dec1(u1)

        return self.final_conv(d8)
```

In [3]:
```
generator = AttentionResUNetGenerator()
x = torch.randn(4, 3, 128, 256)  # Batch of horizontal motion blurred images
y = generator(x)  # Deblurred output
print(y.shape)  # Should be (4, 3, 128, 256)
```

```
torch.Size([4, 3, 128, 256])
```

Discriminator

In [27]:
```
class PatchDiscriminator(nn.Module):
    def __init__(self, in_channels=3, base_filters=64):
        super().__init__()
        layers = [
            nn.Conv2d(in_channels, base_filters, kernel_size=4, stride=2, padding=1
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(base_filters, base_filters * 2, kernel_size=4, stride=2, padd
            nn.BatchNorm2d(base_filters * 2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(base_filters * 2, base_filters * 4, kernel_size=4, stride=2,
            nn.BatchNorm2d(base_filters * 4),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(base_filters * 4, base_filters * 8, kernel_size=4, stride=2,
            nn.BatchNorm2d(base_filters * 8),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(base_filters * 8, 1, kernel_size=4, stride=2, padding=1),
            nn.Sigmoid()
        ]
        self.model = nn.Sequential(*layers)
```

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

In [28]:
```
image1 = torch.rand((1, 3, 128, 256))

discriminator = PatchDiscriminator()
output = discriminator(image1)
print(output.shape)
```

```
torch.Size([1, 1, 4, 8])
```

Model Parameters

In [4]:
```
total_params = sum(p.numel() for p in generator.parameters())
total_params
```

Out[4]:
```
39068999
```

In [30]:
```
discriminator = to_device(discriminator, device)
generator = to_device(generator, device)
```

Training

Loss Functions

In [31]:
```
comparison_loss = nn.BCEWithLogitsLoss()
L1_loss_fn = nn.L1Loss()
```

PaddleOCR

In [32]:
```
ocr_model = PaddleOCR(use_angle_cls=False, lang='en', det=False, rec=True, use_gpu=

def tensor_to_bgr_numpy(tensor_img, mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)):
    if tensor_img.dim() == 4:
        tensor_img = tensor_img[0]

    if tensor_img.shape[0] != 3:
         raise ValueError(f"Expected 3 channels, got shape {tensor_img.shape}")

    unnorm = torch.zeros_like(tensor_img)
    for c in range(3):
        unnorm[c] = tensor_img[c] * float(std[c]) + float(mean[c])
    tensor_img = unnorm.clamp(0, 1)

    pil_img = TF.to_pil_image(tensor_img.cpu())
    np_img = np.array(pil_img)[:, :, ::-1]

    return np_img
```

```
[2025/09/01 06:55:05] ppocr DEBUG: Namespace(help='==SUPPRESS==', use_gpu=True, use_
xpu=False, use_npu=False, use_mlu=False, use_gcu=False, ir_optim=True, use_tensorrt=
False, min_subgraph_size=15, precision='fp32', gpu_mem=500, gpu_id=0, image_dir=Non
e, page_num=0, det_algorithm='DB', det_model_dir='C:\\Users\\ADMIN/.paddleocr/whl\\d
et\\en\\en_PP-OCRv3_det_infer', det_limit_side_len=960, det_limit_type='max', det_bo
x_type='quad', det_db_thresh=0.3, det_db_box_thresh=0.6, det_db_unclip_ratio=1.5, ma
x_batch_size=10, use_dilation=False, det_db_score_mode='fast', det_east_score_thresh
=0.8, det_east_cover_thresh=0.1, det_east_nms_thresh=0.2, det_sast_score_thresh=0.5,
det_sast_nms_thresh=0.2, det_pse_thresh=0, det_pse_box_thresh=0.85, det_pse_min_area
=16, det_pse_scale=1, scales=[8, 16, 32], alpha=1.0, beta=1.0, fourier_degree=5, rec
_algorithm='SVTR_LCNet', rec_model_dir='C:\\Users\\ADMIN/.paddleocr/whl\\rec\\en\\en
_PP-OCRv4_rec_infer', rec_image_inverse=True, rec_image_shape='3, 48, 320', rec_batc
h_num=6, max_text_length=25, rec_char_dict_path='C:\\Thesis\\LiPAD with Paddle\\lipa
dvenv3.9\\lib\\site-packages\\paddleocr\\ppocr\\utils\\en_dict.txt', use_space_char=
True, vis_font_path='./doc/fonts/simfang.ttf', drop_score=0.5, e2e_algorithm='PGNe
t', e2e_model_dir=None, e2e_limit_side_len=768, e2e_limit_type='max', e2e_pgnet_scor
e_thresh=0.5, e2e_char_dict_path='./ppocr/utils/ic15_dict.txt', e2e_pgnet_valid_set
='totaltext', e2e_pgnet_mode='fast', use_angle_cls=False, cls_model_dir='C:\\Users
\\ADMIN/.paddleocr/whl\\cls\\ch_ppocr_mobile_v2.0_cls_infer', cls_image_shape='3, 4
8, 192', label_list=['0', '180'], cls_batch_num=6, cls_thresh=0.9, enable_mkldnn=Fal
se, cpu_threads=10, use_pdserving=False, warmup=False, sr_model_dir=None, sr_image_s
hape='3, 32, 128', sr_batch_num=1, draw_img_save_dir='./inference_results', save_cro
p_res=False, crop_res_save_dir='./output', use_mp=False, total_process_num=1, proces
s_id=0, benchmark=False, save_log_path='./log_output/', show_log=True, use_onnx=Fals
e, onnx_providers=False, onnx_sess_options=False, return_word_box=False, output='./o
utput', table_max_len=488, table_algorithm='TableAttn', table_model_dir=None, merge_
no_span_structure=True, table_char_dict_path=None, formula_algorithm='LaTeXOCR', for
mula_model_dir=None, formula_char_dict_path=None, formula_batch_num=1, layout_model_
dir=None, layout_dict_path=None, layout_score_threshold=0.5, layout_nms_threshold=0.
5, kie_algorithm='LayoutXLM', ser_model_dir=None, re_model_dir=None, use_visual_back
bone=True, ser_dict_path='../train_data/XFUND/class_list_xfun.txt', ocr_order_method
=None, mode='structure', image_orientation=False, layout=True, table=True, formula=F
alse, ocr=True, recovery=False, recovery_to_markdown=False, use_pdf2docx_api=False,
invert=False, binarize=False, alphacolor=(255, 255, 255), lang='en', det=False, rec=
True, type='ocr', savefile=False, ocr_version='PP-OCRv4', structure_version='PP-Stru
ctureV2')
[2025/09/01 06:55:05] ppocr WARNING: The first GPU is used for inference by default,
GPU ID: 0
[2025/09/01 06:55:07] ppocr WARNING: The first GPU is used for inference by default,
GPU ID: 0
```

Discriminator Training Function

In [33]:

```python
def train_discriminator(discriminator, generator, inputs, targets, d_opt):
    discriminator.train()

    real_images = targets
    fake_images = generator(inputs).detach()

    pred_real = discriminator(real_images)
    pred_fake = discriminator(fake_images)

    real_labels = torch.ones_like(pred_real)
    fake_labels = torch.zeros_like(pred_fake)
```

```
    real_score = torch.sigmoid(pred_real).mean().item()
    fake_score = torch.sigmoid(pred_fake).mean().item()

    loss_real = comparison_loss(pred_real, real_labels)
    loss_fake = comparison_loss(pred_fake, fake_labels)

    d_loss = (loss_real + loss_fake) / 2

    d_opt.zero_grad()
    d_loss.backward()
    d_opt.step()

    return d_loss.item(), real_score, fake_score
```

Generator Training Function

In [34]:

```
def ssim(pred, target):
    return 1 - ssim_fn(pred, target, data_range=1.0, size_average=True)

lpips_loss = LPIPS(net='vgg').to(device)
lpips_loss.eval()
for param in lpips_loss.parameters():
    param.requires_grad = False

def train_generator(generator, discriminator, inputs, targets, g_opt, adv_lambda=1.
    generator.train()

    fake_images = generator(inputs)
    pred_fake = discriminator(fake_images)

    real_labels = torch.ones_like(pred_fake)

    adv_loss = comparison_loss(pred_fake, real_labels)
    l1_loss = L1_loss_fn(fake_images, targets)
    perceptual_loss = lpips_loss(fake_images.clamp(-1, 1), targets.clamp(-1, 1)).me
    ssim_loss = ssim(fake_images, targets)

    # Text Loss
    fake_np = tensor_to_bgr_numpy(fake_images)
    real_np = tensor_to_bgr_numpy(targets)

    with torch.no_grad():
        fake_text = ocr_model.ocr(fake_np, cls=False, det=False)[0]
        real_text = ocr_model.ocr(real_np, cls=False, det=False)[0]

    fake_str = fake_text[0][0] if fake_text else ''
    real_str = real_text[0][0] if real_text else ''

    fake_str = fake_str.strip().replace(' ', '')
    real_str = real_str.strip().replace(' ', '')

    # Use normalized edit distance as loss
    edit_dist = levenshtein_distance(fake_str, real_str)
    norm_edit_dist = edit_dist / max(len(real_str), 1)
    text_loss = torch.tensor(norm_edit_dist, device=device)
```

```
    g_loss = (
        adv_loss * adv_lambda +
        l1_loss * l1_lambda +
        ssim_loss * ssim_lambda +
        perceptual_loss * perceptual_lambda +
        text_loss * text_lambda
    )

    g_opt.zero_grad()
    g_loss.backward()
    g_opt.step()

    return g_loss.item(), fake_images, {
        "total_loss": g_loss.item(),
        "adv": adv_loss.item(),
        "l1": l1_loss.item(),
        "ssim": ssim_loss.item(),
        "perceptual": perceptual_loss.item(),
        "text": text_loss.item()
    }
```

```
Setting up [LPIPS] perceptual loss: trunk [vgg], v[0.1], spatial [off]
Loading model from: C:\Thesis\LiPAD with Paddle\lipadvenv3.9\lib\site-packages\lpips
\weights\v0.1\vgg.pth
```

In [35]:
```
psnr_metric = PeakSignalNoiseRatio(data_range=1.0).to(device)
ssim_metric = StructuralSimilarityIndexMeasure(data_range=1.0).to(device)
fid_metric = FrechetInceptionDistance(feature=2048).to(device)

def to_uint8(tensor):
    # Clamp to [0, 1], scale to [0, 255], then convert to uint8
    tensor = torch.clamp(tensor, 0.0, 1.0)
    tensor = (tensor * 255.0).to(torch.uint8)
    return tensor

def evaluate_test(generator):
    raw_grid_img, gen_grid_img = create_image_grid_from_loader(generator, test_dl,

    fig, axes = plt.subplots(1, 2, figsize=(10, 8))  # 1 row, 2 columns

    # Raw Test Set
    axes[0].imshow(raw_grid_img)
    axes[0].axis("off")
    axes[0].set_title("Raw Test Set")

    # Generated Test Set
    axes[1].imshow(gen_grid_img)
    axes[1].axis("off")
    axes[1].set_title("Generated Test Set")

    plt.tight_layout()
    plt.show()
```

In [36]:
```
def create_image_grid_from_loader(generator, dataloader, device, num_images=16, ima
    generator.eval()
```

```
    generator.to(device)

    raw_images = []
    gen_images = []
    gen_count = 0
    raw_count = 0

    for batch in dataloader:
        # Assume batch is either just images or (images, labels)
        if isinstance(batch, (tuple, list)):
            inputs = batch[0]
        else:
            inputs = batch

        inputs = inputs.to(device)
        outputs = generator(inputs)

        for img in inputs:
            if denormalize:
                img = denorm(img)

            img_resized = TF.resize(img, image_size)
            raw_images.append(img_resized)

            raw_count += 1
            if raw_count >= num_images:
                break

        for out_img in outputs:
            if denormalize:
                out_img = denorm(out_img)

            out_img_resized = TF.resize(out_img, image_size)
            gen_images.append(out_img_resized)

            gen_count += 1
            if gen_count >= num_images:
                break

        if raw_count >= num_images:
            break

    raw_grid = make_grid(raw_images, nrow=4)
    gen_grid = make_grid(gen_images, nrow=4)

    raw_ndarr = raw_grid.mul(255).byte().cpu().permute(1, 2, 0).numpy()
    gen_ndarr = gen_grid.mul(255).byte().cpu().permute(1, 2, 0).numpy()

    raw_grid_image = Image.fromarray(raw_ndarr)
    gen_grid_image = Image.fromarray(gen_ndarr)

    if save_path:
        gen_grid_image.save(save_path)
        print(f"Saved image grid to {save_path}")

    return raw_grid_image, gen_grid_image
```

In [37]:

```
@torch.no_grad()
def evaluate_generator(generator, valid_dl, epoch):
    generator.eval()
    psnr_vals = []
    ssim_vals = []
    fid_metric.reset()
    shown = False

    for val_inputs, val_targets in valid_dl:
        val_inputs = val_inputs.to(device)
        val_targets = val_targets.to(device)

        val_outputs = generator(val_inputs)

        if not shown:
            print('Validation Data')
            print_images(val_inputs, 5)
            print_images(val_outputs, 5)
            print_images(val_targets, 5)
            shown = True

        # Convert to uint8 for FID
        gen_uint8 = to_uint8(val_outputs)
        target_uint8 = to_uint8(val_targets)

        fid_metric.update(gen_uint8, real=False)
        fid_metric.update(target_uint8, real=True)

        # Compute PSNR/SSIM
        psnr = psnr_metric(val_outputs, val_targets).item()
        ssim = ssim_metric(val_outputs, val_targets).item()
        psnr_vals.append(psnr)
        ssim_vals.append(ssim)

    mean_psnr = sum(psnr_vals) / len(psnr_vals)
    mean_ssim = sum(ssim_vals) / len(ssim_vals)
    fid = fid_metric.compute().item()

    print(f"Epoch [{epoch+1}/{epochs}] - Mean PSNR: {mean_psnr:.4f}, Mean SSIM: {me

    evaluate_test(generator)
    return mean_psnr, mean_ssim, fid

def fit(generator, discriminator, epochs, lr, adv_lambda=1.0, l1_lambda=100.0, ssim
        perceptual_lambda=5.0, text_lambda=7.5, g_opt=None, d_opt=None, checkpoint=

    torch.cuda.empty_cache()

    losses_g = []
    losses_d = []
    real_scores = []
    fake_scores = []
    psnrs = []
    ssims = []
    fids = []
```

```
    g_losses_trace = []

     # Create optimizers
    if d_opt is None:
        d_opt = optim.Adam(discriminator.parameters(), lr=lr, betas=(beta1, beta2))
    if g_opt is None:
        g_opt = optim.Adam(generator.parameters(), lr=lr, betas=(beta1, beta2))

    # Load checkpoint state if resuming
    if checkpoint is not None:
        print("Resuming training from checkpoint...")
        d_opt.load_state_dict(checkpoint['opt_d'])
        g_opt.load_state_dict(checkpoint['opt_g'])

    # Train
    for epoch in range(80, epochs):
        torch.cuda.empty_cache()
        generator.train()
        discriminator.train()
        for inputs, targets in tqdm(train_dl):
            inputs = inputs.to(device)
            targets = targets.to(device)

            # Train discriminator
            d_loss, real_score, fake_score = train_discriminator(discriminator, gen

            # Train generator
            g_loss, fake_images, g_loss_trace = train_generator(generator, discrimi
                                                    l1_lambda, ssim_lambda, perce

        print('Training Data')
        print_images(inputs, 5)
        print_images(fake_images, 5)
        print_images(targets, 5)

        if (epoch + 1) % 5 == 0:
            torch.save(generator.state_dict(), f'[RAU-NET]generator-Datasetv2-{epoc
            torch.save(discriminator.state_dict(), f'[RAU-NET]discriminator-Dataset
            torch.save({
                'generator': generator.state_dict(),
                'discriminator': discriminator.state_dict(),
                'opt_g': g_opt.state_dict(),
                'opt_d': d_opt.state_dict(),
                'losses_g': losses_g,
                'losses_d': losses_d,
                'real_score': real_scores,
                'fake_score': fake_scores,
                'psnrs': psnrs,
                'ssims': ssims,
                'fids': fids,
                'epoch': epoch + 1
            }, f'[RAU-NET]Checkpoint-Logs-Datasetv2-{epoch + 1} epoch.pt')

        # Print progress
        try:
            print('Epoch [{}/{}], loss_g: {:.4f}, loss_d: {:.4f}, real_score: {:.4f
```

```
                    epoch+1, epochs, g_loss, d_loss, real_score, fake_score))
        except Exception as err:
            print('Error:', str(err))

        mean_psnr, mean_ssim, fid = evaluate_generator(generator, valid_dl, epoch)

        # Record losses and scores
        losses_g.append(g_loss)
        losses_d.append(d_loss)
        real_scores.append(real_score)
        fake_scores.append(fake_score)
        psnrs.append(mean_psnr)
        ssims.append(mean_ssim)
        fids.append(fid)
        g_losses_trace.append(g_loss_trace)

        save_samples(generator, test_dl, epoch + 1, sample_dir='./visualization-2')

    return losses_g, losses_d, real_scores, fake_scores, psnrs, ssims, fids, g_loss
```

Hyperparameter Configurations

In [38]:
```
adv_lambda = 0.25 # Change to 0.25
l1_lambda = 100
ssim_lambda = 5 # Change to 5
perceptual_lambda = 5.0
text_lambda = 10 # Change to 10

lr = 0.00005
beta1 = 0.5
beta2 = 0.999

epochs = 100
```

In [30]:
```
g_opt = optim.Adam(generator.parameters(), lr=lr, betas=(beta1, beta2))
d_opt = optim.Adam(discriminator.parameters(), lr=lr, betas=(beta1, beta2))
```

In [31]:
```
history = []
```

In [32]:
```
%%time
history += fit(generator,
               discriminator,
               epochs,
               lr,
               adv_lambda=adv_lambda,
               l1_lambda=l1_lambda,
               ssim_lambda=ssim_lambda,
               perceptual_lambda=perceptual_lambda,
               text_lambda=text_lambda,
               g_opt=g_opt,
               d_opt=d_opt,
               checkpoint=checkpoint)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
File <timed exec>:12

NameError: name 'checkpoint' is not defined
```

Post Training

In [34]:
```
torch.cuda.empty_cache()
```

Save Model

In [34]:
```
torch.save(generator.state_dict(), "[RAU-NET]generator-Datasetv2-5epoch.pt")
torch.save(discriminator.state_dict(), "[RAU-NET]discriminator-Datasetv2-5epoch.pt"
```

In [35]:
```
torch.save({
    'generator_state_dict': generator.state_dict(),
    'discriminator_state_dict': discriminator.state_dict(),
    'opt_g_state_dict': g_opt.state_dict(),
    'opt_d_state_dict': d_opt.state_dict(),
    'history': history
}, f'[RAU-NET]Checkpoint-Datasetv2-30epoch.pt')
```

Model Evaluation

In [ ]:
```
psnr_vals = []
ssim_vals = []

fid_metric = FrechetInceptionDistance(feature=2048).to(device)
fid_metric.reset()

generator.eval()
for val_input, val_target in valid_dl:
    val_input = val_input.to(device)
    val_target = val_target.to(device)

    with torch.no_grad():
        gen_output = generator(val_input)

    # Convert to uint8 for FID
    gen_uint8 = to_uint8(gen_output)
    target_uint8 = to_uint8(val_target)

    fid_metric.update(gen_uint8, real=False)
    fid_metric.update(target_uint8, real=True)

    show_generated_images(gen_output)

    # Compute PSNR/SSIM
```

```
    psnr = psnr_metric(gen_output, val_target).item()
    ssim = ssim_metric(gen_output, val_target).item()
    psnr_vals.append(psnr)
    ssim_vals.append(ssim)

print("Mean PSNR:", sum(psnr_vals) / len(psnr_vals))
print("Mean SSIM:", sum(ssim_vals) / len(ssim_vals))
print("FID:", fid_metric.compute().item())
```


DAZ 9945
NBK 7762
AIA 3157
TGJ·876
DCP 9851
NDG 4508
0401-0
DCP 9984
DAG 9140
NAV 70
534NDE
TGJ·876
TGJ·876
NAP 9734
NBP 3657
JIX 125
AYK·684
WLI 301
WPC·242
DAL 7387
ZDA 908
NIP 498
NDS 5916
UCI 227
ZRU 201
CAD·968
VDAA 9607
UIB 652
NLQ 302
DAG 9140
NDR 7414
DAA 9142
359DPK
AAA 6660
NBZ 4064
DAG 9140
WPC·242

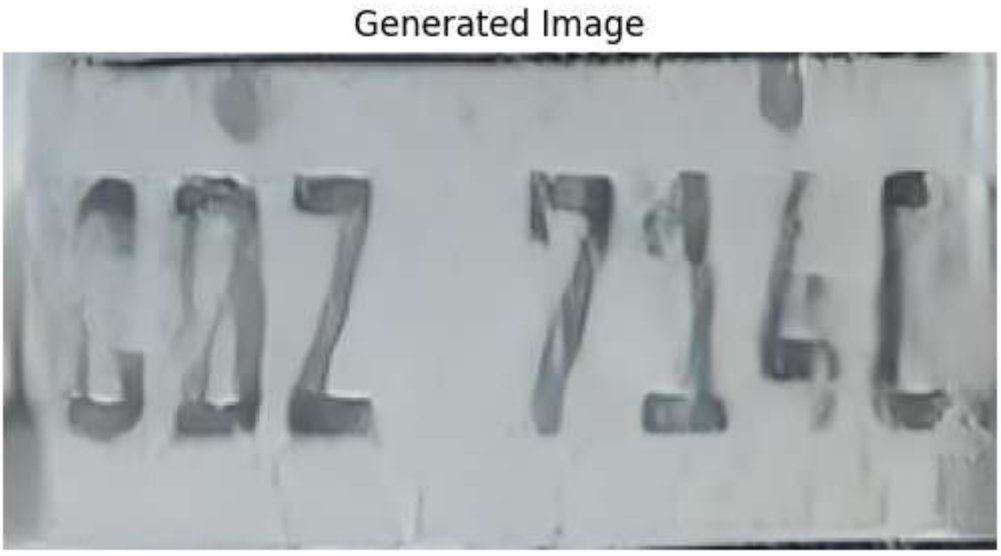

NCC 1984
NBS 9297
NCR 6979
NAA 9525
XRJ 395
UHJ 230
DAG 9140
DCP 9851
DAP 1878
E1 D960
NDD 4538
PRW 348
NDD 2923
DAA 9142
CAM 5524
WQR 579
ZDH 705
THE 07
NEO 8835
NBA 2005
NAF 7510
DAL 7387
DAZ 4722
XKZ 642
AHH 758
AHA 5851
NBQ 5239
NCJ 6149
NCZ 4304
ATA 5018
GEB 4887
NBU 4937
AIA 3157
NBZ 4064
DCP 9851
PNB 439
236DAJ
TGJ 876


ATA 5018
NDL 7075
NAF 7510
ABO 1958
ZDA 908
NFC 5885
NDD 8687
CAM 5524
NBZ 1806
NAC 6182
BEP 961
WPC·242
UUO 605
AAA 5409
NFY 9079
ZJM 772
WUT 131
NIP 498
ZLW 733
NDJ 4955
NFY 9079
WPC·242
ABO 1958
GAO 1276
DAG 9140
NDS 5916
TGJ·876
WLT 301
NGK 6104
CAU 5254
359DPK
DAG 9140
ARA 2976
359DPK
NEF 3315
WKV 161
NEG 5445
359DPK

In [41]:
```python
def deblur_image(image_path, generator, device):
    image = Image.open(image_path).convert('RGB')
    input_tensor = valid_transform(image).unsqueeze(0).to(device)

    generator.eval()
    with torch.no_grad():
        gen_image = generator(input_tensor)

    output = denorm(gen_image.squeeze(0).cpu())
```

```
    plt.imshow(output.permute(1, 2, 0))  # CHW → HWC
    plt.axis('off')
    plt.title("Generated Image")
    plt.show()
```

In [43]:
```
deblur_image(os.path.join(DATA_DIR, 'h_blur', 'v2', 'test', 'h_blur', '135.jpg'), g
```



In [42]:
```
raw_grid_img, gen_grid_img = create_image_grid_from_loader(generator, test_dl, devi

fig, axes = plt.subplots(1, 2, figsize=(15, 15))  # 1 row, 2 columns

# Raw Test Set
axes[0].imshow(raw_grid_img)
axes[0].axis("off")
axes[0].set_title("Raw Test Set")

# Generated Test Set
axes[1].imshow(gen_grid_img)
axes[1].axis("off")
axes[1].set_title("Generated Test Set")

plt.tight_layout()
plt.show()
```

In [31]:
```
torch.cuda.empty_cache()
```

Load Model

In [39]:
```
MODELS_DIR = os.path.join('..', '..', '..', 'models', 'h_blur')
```

In [40]:
```
generator = AttentionResUNetGenerator()
loaded_g_wts = torch.load(os.path.join(MODELS_DIR, 'BEST MODELS', '0.25 Adv Loss Mo
generator.load_state_dict(loaded_g_wts)
generator = to_device(generator, device)

discriminator = PatchDiscriminator()
loaded_d_wts = torch.load(os.path.join(MODELS_DIR, 'BEST MODELS', '0.25 Adv Loss Mo
discriminator.load_state_dict(loaded_d_wts)
discriminator = to_device(discriminator, device)
```

In [30]:
```
checkpoint = torch.load(os.path.join(MODELS_DIR, 'BEST MODELS', '0.25 Adv Loss Mode
#g_opt.load_state_dict(checkpoint['opt_g_state_dict'])
#d_opt.load_state_dict(checkpoint['opt_d_state_dict'])
#history = checkpoint['losses_g']
```

In [39]:
```
def are_models_identical(model1, model2):
    # Check if state dictionaries have the same keys
    if model1.state_dict().keys() != model2.state_dict().keys():
        return False

    # Compare each parameter's values
    for key in model1.state_dict():
        if not torch.equal(model1.state_dict()[key], model2.state_dict()[key]):
            return False
    return True
```

In [45]:
```
if are_models_identical(generator1, generator2):
    print("Models have identical weights and biases")
else:
    print("Models have different weights or biases")
```

```
Models have different weights or biases
```

In [9]:
```
len(history)
```

Out[9]: 79

Save Reconstructed Test Set

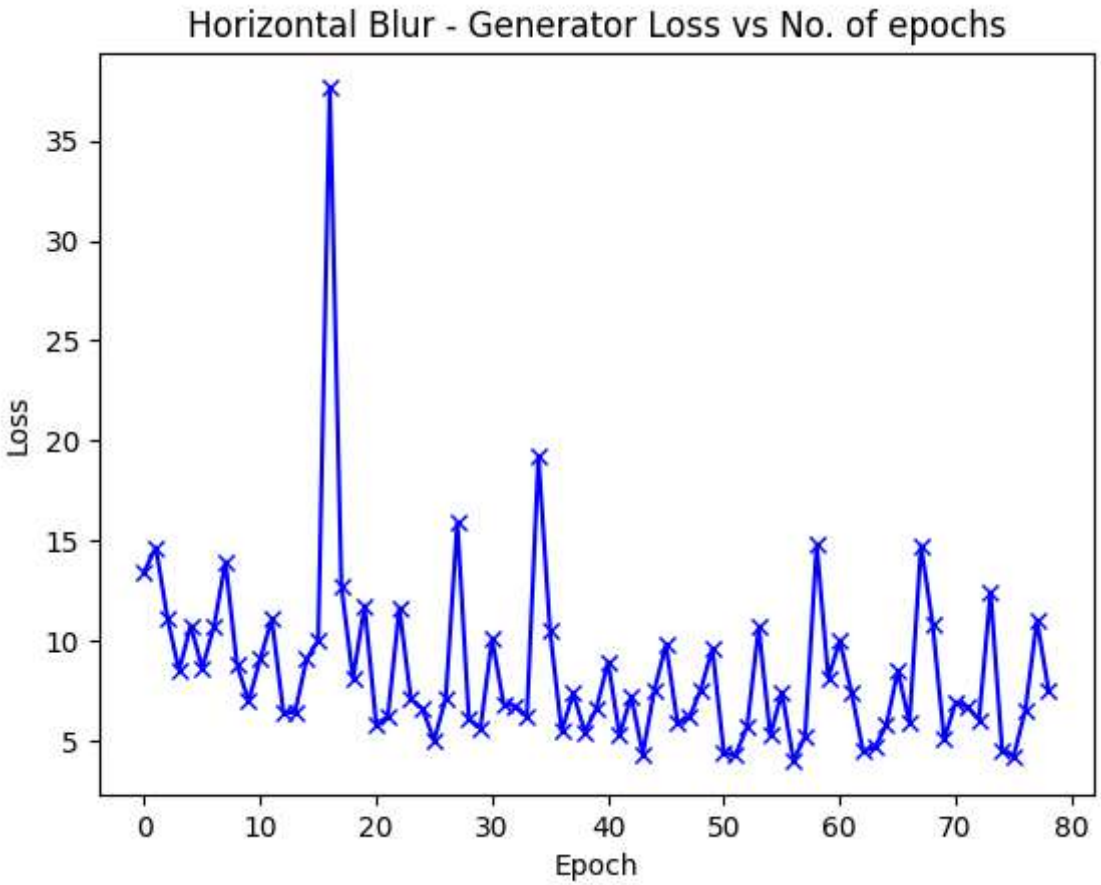
In [12]:
```
def save_generated_tests(generator, test_dl, sample_dir='./results'):
    generator.eval()
    os.makedirs(sample_dir, exist_ok=True)

    with torch.no_grad():
        global_idx = 1  # running index for image IDs

        for inputs, _ in test_dl:
            inputs = inputs.to(device)

            fake_images = generator(inputs)
            fake_images = fake_images.clamp(-1, 1)

            batch_size = fake_images.size(0)
```

```
            for b in range(batch_size):
                img = fake_images[b]

                file_name = f"{global_idx}.jpg"
                save_path = os.path.join(sample_dir, file_name)

                save_image(denorm(img).unsqueeze(0), save_path, nrow=1)
                global_idx += 1
```

In [40]:
```
save_generated_tests(generator, test_dl)
```

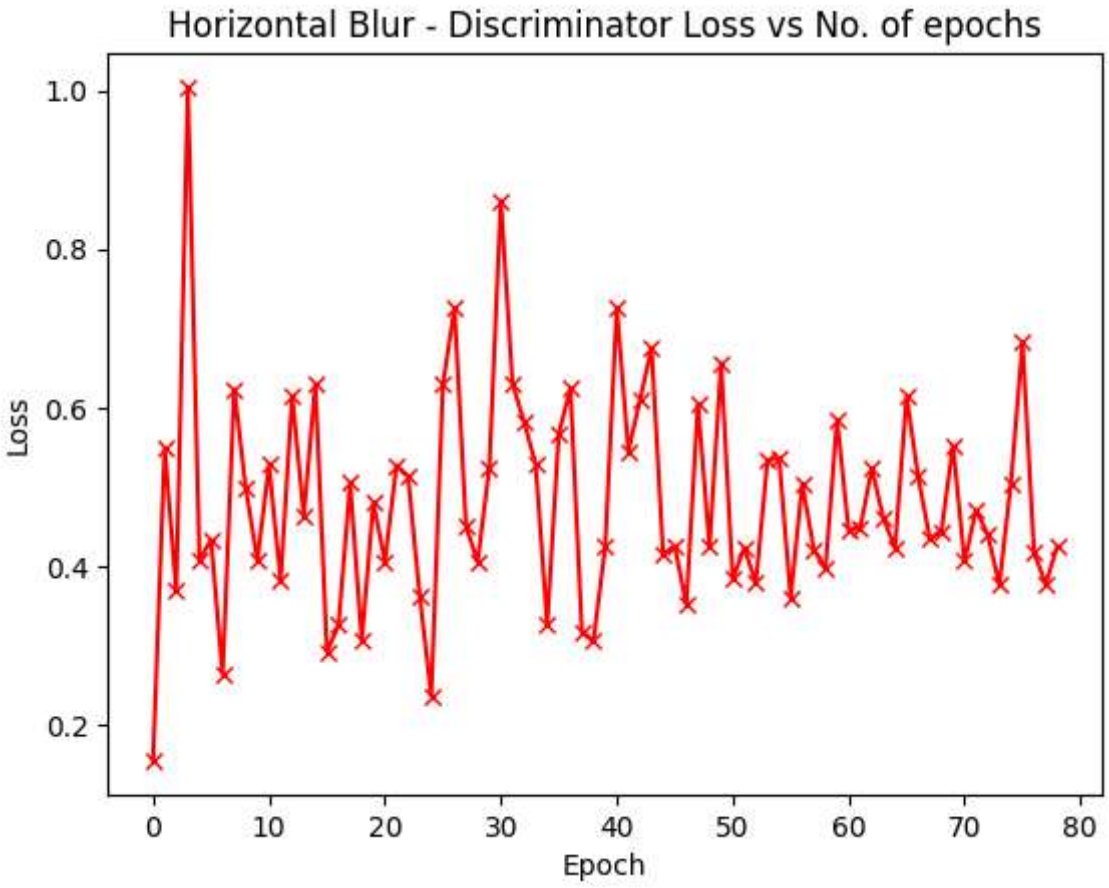
Plots

In [68]:
```
def plot_g_losses(history):
    losses_g = [x for x in history['losses_g']]
    plt.plot(losses_g, '-bx')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Horizontal Blur - Generator Loss vs No. of epochs')

plot_g_losses(checkpoint)
```

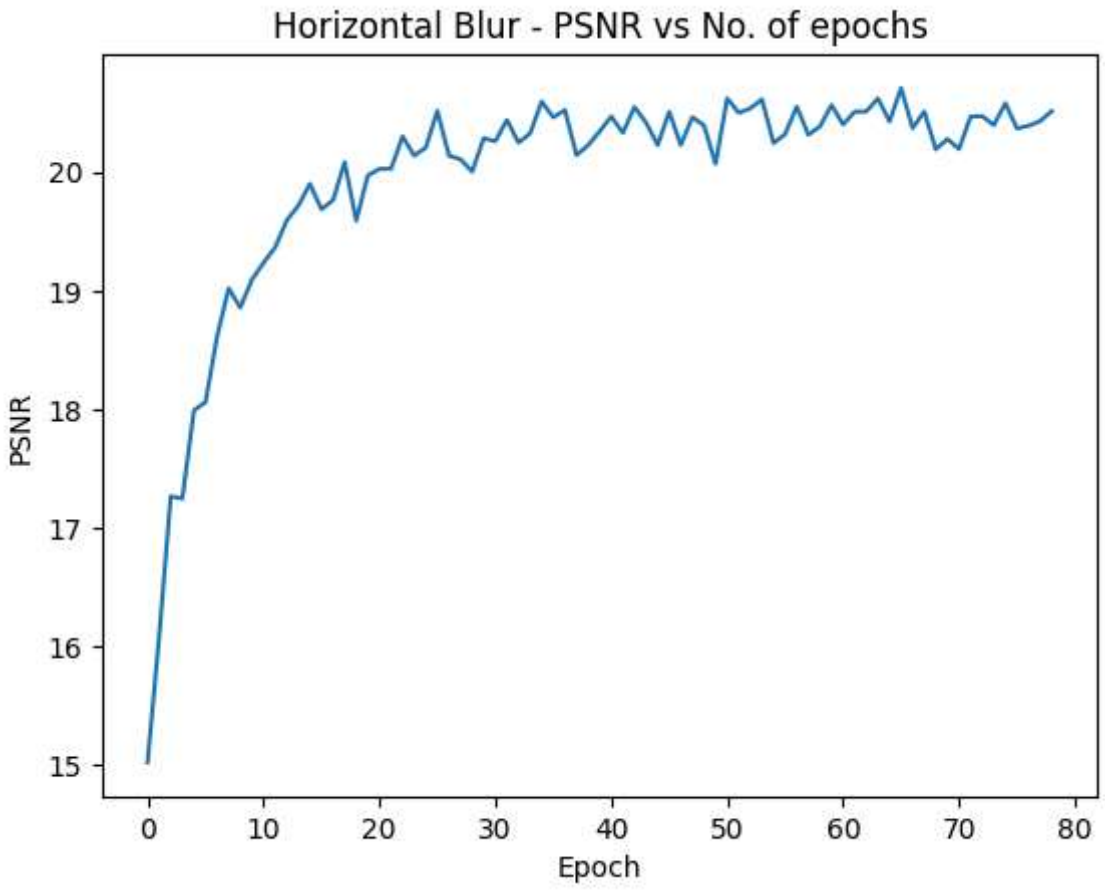


In [71]:
```
def plot_d_losses(history):
    losses_d = [x for x in history['losses_d']]
    plt.plot(losses_d, '-rx')
```

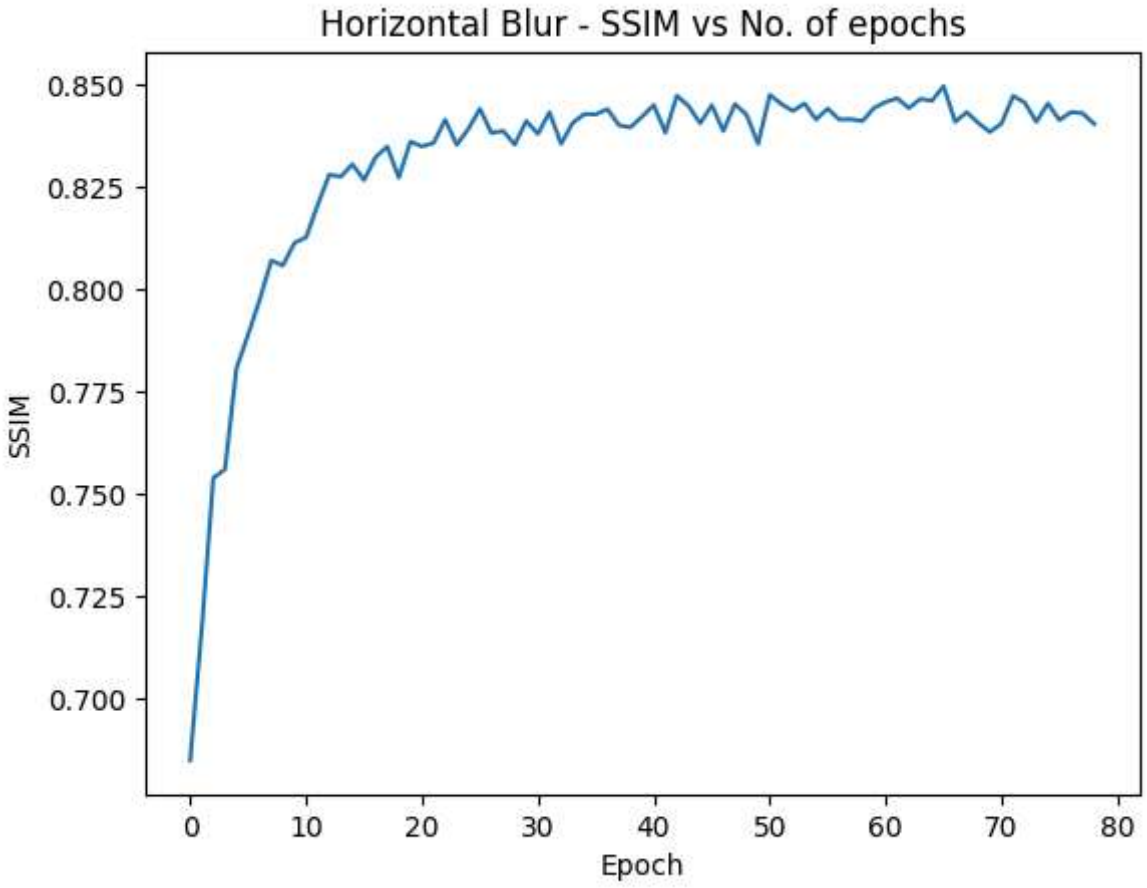
```
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Horizontal Blur - Discriminator Loss vs No. of epochs')

plot_d_losses(checkpoint)
```



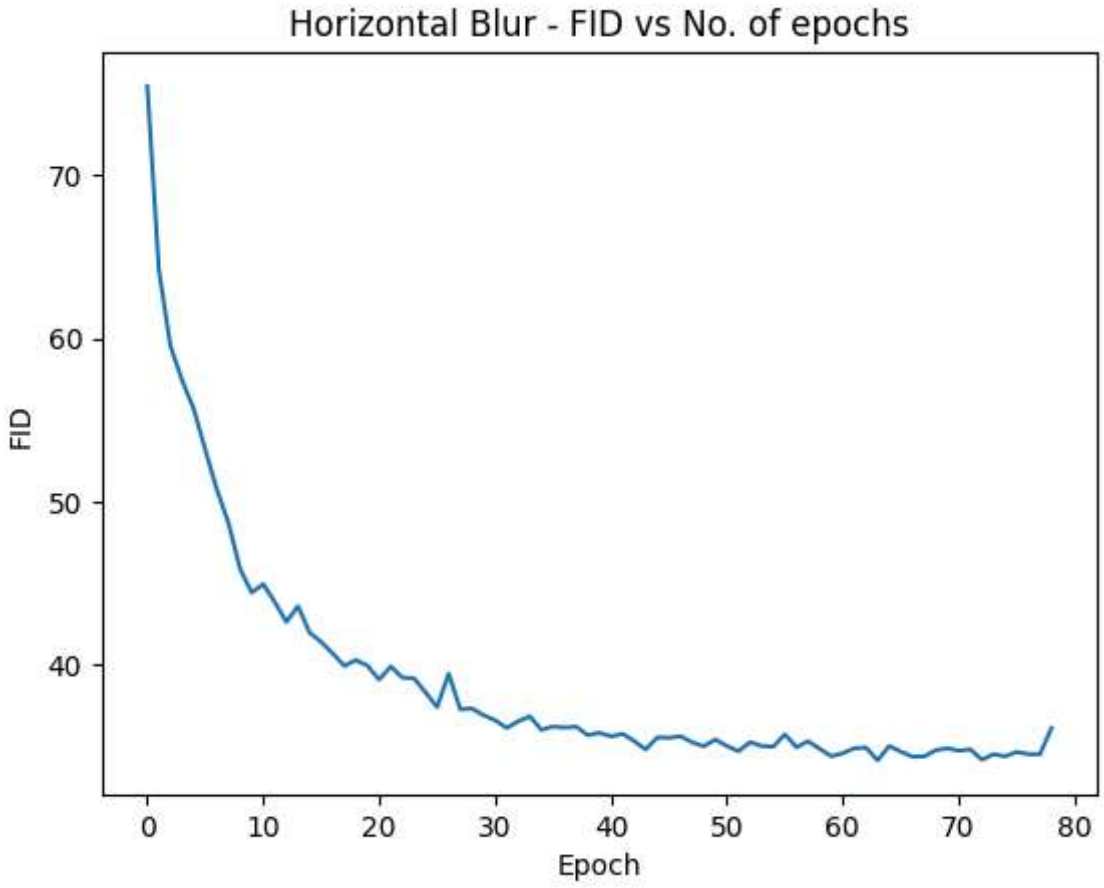
In [62]:
```
def plot_psnrs(history):
    psnrs = [x for x in history['psnrs']]
    plt.plot(psnrs)
    plt.xlabel('Epoch')
    plt.ylabel('PSNR')
    plt.title('Horizontal Blur - PSNR vs No. of epochs')

plot_psnrs(checkpoint)
```



In [65]:
```
def plot_ssims(history):
    psnrs = [x for x in history['ssims']]
    plt.plot(psnrs)
    plt.xlabel('Epoch')
    plt.ylabel('SSIM')
    plt.title('Horizontal Blur - SSIM vs No. of epochs')

plot_ssims(checkpoint)
```



In [67]:
```
def plot_fids(history):
    psnrs = [x for x in history['fids']]
    plt.plot(psnrs)
    plt.xlabel('Epoch')
    plt.ylabel('FID')
    plt.title('Horizontal Blur - FID vs No. of epochs')

plot_fids(checkpoint)
```


Horizontal Blur - FID vs No. of epochs
FID
70
60
50
40
0
10
20
30
40
50
60
70
80
Epoch

In [ ]:

In [ ]: