In [1]:
```
import os
import sys
from tqdm import tqdm
import numpy as np
import torch
import torch.nn as nn
from torch.optim.lr_scheduler import ReduceLROnPlateau
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 [60]:
```
train_ds = PairedImageDataset(blur_dir=os.path.join(DATA_DIR, 'v_blur', 'train', 'v
                              normal_dir=os.path.join(DATA_DIR, 'v_blur', 'train',
```

```
                                transform=train_transform)

valid_ds = PairedImageDataset(blur_dir=os.path.join(DATA_DIR, 'v_blur', 'valid', 'v
                                normal_dir=os.path.join(DATA_DIR, 'v_blur', 'valid',
                                transform=valid_transform)

test_ds = ImageFolder(os.path.join(DATA_DIR, 'v_blur', 'test'), transform=valid_tra
```

In [61]:
```
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=batch_size//2, shuffle=False, num_workers=
```

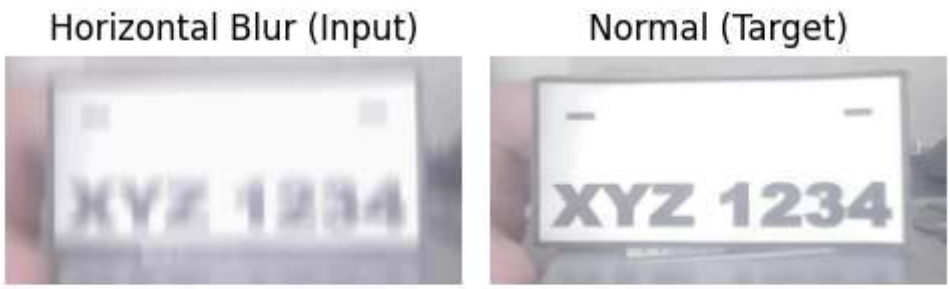
Helper functions

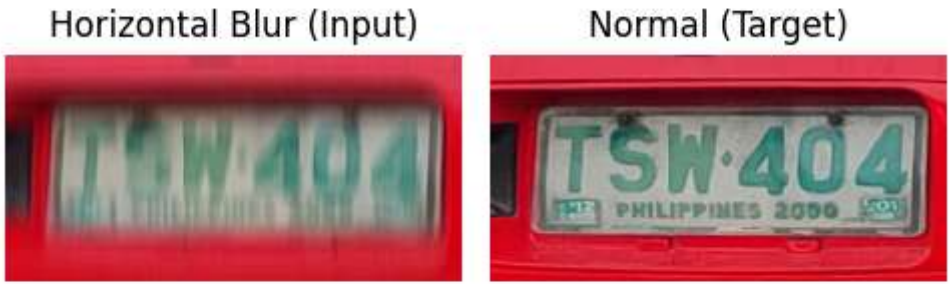
In [62]:
```
def denorm(img_tensors):
    return img_tensors * stats[1][0] + stats[0][0]

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
```

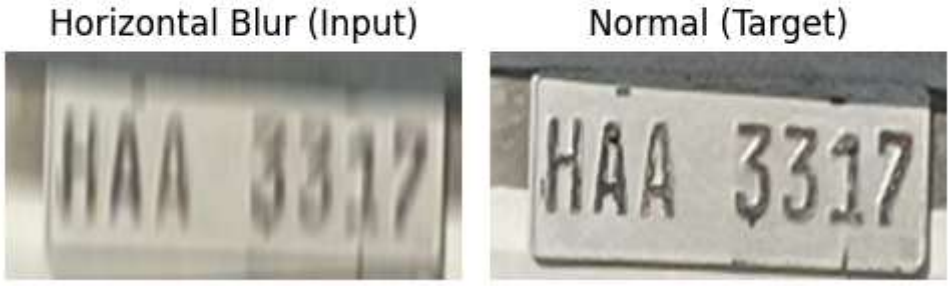
In [63]:
```
def show_paired_samples(dataset, num_samples=4):
    fig, axes = plt.subplots(num_samples, 2, figsize=(5, 2 * num_samples))

    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 [64]:
```
show_paired_samples(train_ds)
```







In [65]:
```
show_batch(valid_dl, nmax=14)
```

In [66]:
```
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 [67]:
```
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 [68]:
```
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 [69]:
```
device = get_default_device()
device
```

Out[69]:
```
device(type='cuda')
```

In [70]:
```
train_dl = DeviceDataLoader(train_dl, device)
valid_dl = DeviceDataLoader(valid_dl, device)
test_dl = DeviceDataLoader(test_dl, device)
```

Building the Model

Generator

In [16]:
```
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 [17]:
```
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 [18]:
```
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),
        ]
        self.model = nn.Sequential(*layers)

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

In [19]:
```
image1 = torch.rand((1, 3, 128, 256))

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

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

Model Parameters

In [20]:
```
total_params = sum(p.numel() for p in discriminator.parameters())
total_params
```

Out[20]: 2766529

In [21]:
```
discriminator = to_device(discriminator, device)
generator = to_device(generator, device)
```

Training

Loss Functions

In [22]:
```
comparison_loss = nn.BCEWithLogitsLoss()
L1_loss_fn = nn.L1Loss()
```

PaddleOCR

In [23]:
```
ocr_model = PaddleOCR(use_angle_cls=True, lang='en', use_gpu=True, show_log=False)

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
```

```
[2025/09/10 17:28:52] ppocr WARNING: The first GPU is used for inference by default,
GPU ID: 0
[2025/09/10 17:28:53] ppocr WARNING: The first GPU is used for inference by default,
GPU ID: 0
[2025/09/10 17:28:56] ppocr WARNING: The first GPU is used for inference by default,
GPU ID: 0
```

Discriminator Training Function

In [24]:
```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 [25]:
```python
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)[0]
        real_text = ocr_model.ocr(real_np, cls=False)[0]

    fake_str = fake_text[0][1][0] if fake_text else ''
    real_str = real_text[0][1][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 [26]:

```
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 [27]:

```
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 [28]:

```
@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 get_lr(optimizer):
    return optimizer.param_groups[0]['lr']

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 = []
    g_lrs = []
    d_lrs = []

     # 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'])

    g_scheduler = ReduceLROnPlateau(g_opt, mode='min', factor=0.5, patience=10, ver
    d_scheduler = ReduceLROnPlateau(d_opt, mode='min', factor=0.5, patience=12, ver

    # Train
    for epoch in range(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 OCRDET LR]generator-Datas
            torch.save(discriminator.state_dict(), f'[RAU-NET OCRDET LR]discriminat
            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,
                'g_lrs': g_lrs,
                'd_lrs': d_lrs,
                'epoch': epoch + 1
            }, f'[RAU-NET OCRDET LR]Checkpoint-Logs-Datasetv2-{epoch + 1} epoch.pt'

        # Print progress
        try:
            curr_glr = get_lr(g_opt)
            curr_dlr = get_lr(d_opt)

            print('Epoch [{}/{}], loss_g: {:.4f}, loss_d: {:.4f}, real_score: {:.4f
                  epoch+1, epochs, g_loss, d_loss, real_score, fake_score))
            print(f"Current LRs — Generator: {curr_glr:.6f}, Discriminator: {curr_d
        except Exception as err:
            print('Error:', str(err))

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

        g_scheduler.step(g_loss)
        d_scheduler.step(d_loss)

        # 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)
        g_lrs.append(curr_glr)
        d_lrs.append(curr_dlr)
```

```
        save_samples(generator, test_dl, epoch + 1, sample_dir='./visualization-ocr

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

Hyperparameter Configurations

In [29]:
```
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 = 0.0002
beta1 = 0.5
beta2 = 0.999

epochs = 200
```

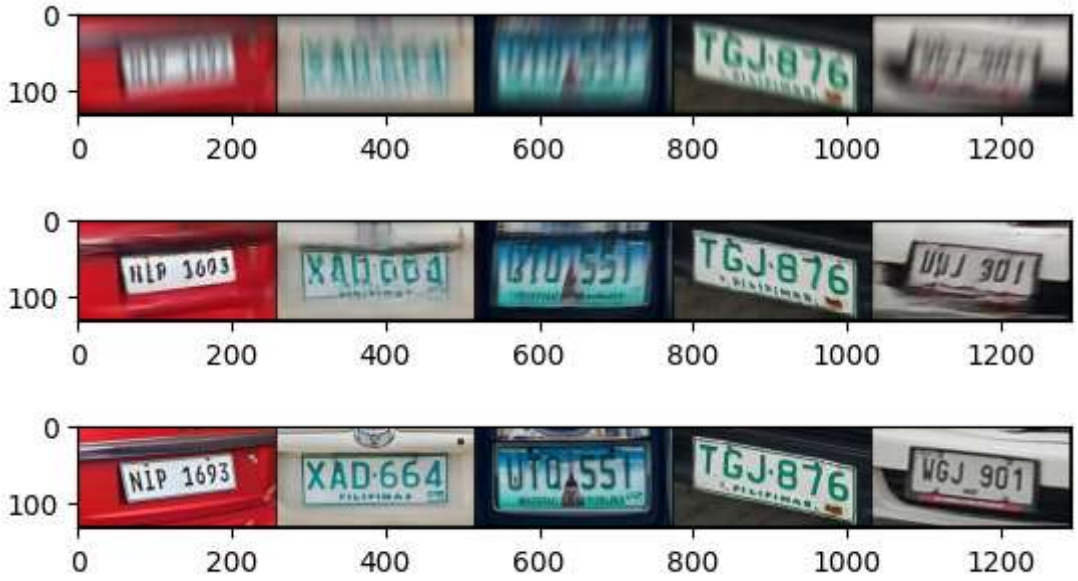
In [30]:
```
g_opt = optim.Adam(generator.parameters(), lr=lr, betas=(beta1, beta2))
d_opt = optim.Adam(discriminator.parameters(), lr=lr, betas=(beta1, beta2))
```

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

In [ ]:
```
%%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)
```

```
100%|██████████████████████████████████████████████████████████████████████████████|
1250/1250 [18:54<00:00,  1.10it/s]
Training Data
```

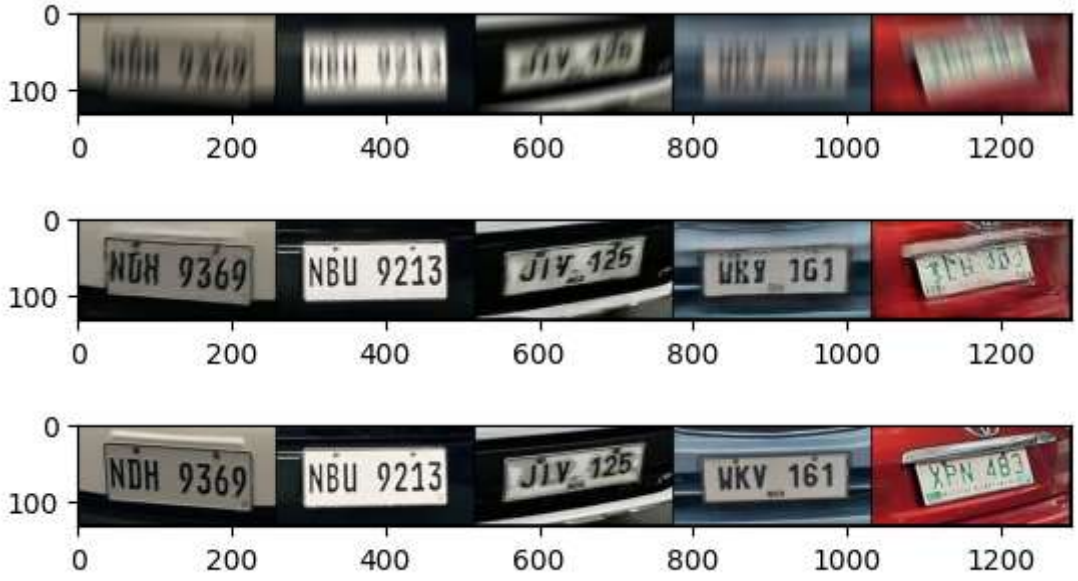
Epoch [1/200], loss_g: 21.7972, loss_d: 0.3590, real_score: 0.9407, fake_score: 0.4576
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data



Epoch [1/200] - Mean PSNR: 12.7310, Mean SSIM: 0.5079, FID: 98.25654602050781

100%|
1250/1250 [19:02<00:00, 1.09it/s]
Training Data

Epoch [2/200], loss_g: 15.5627, loss_d: 0.4916, real_score: 0.9972, fake_score: 0.5998
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data



Epoch [2/200] - Mean PSNR: 13.7796, Mean SSIM: 0.5707, FID: 86.4118423461914

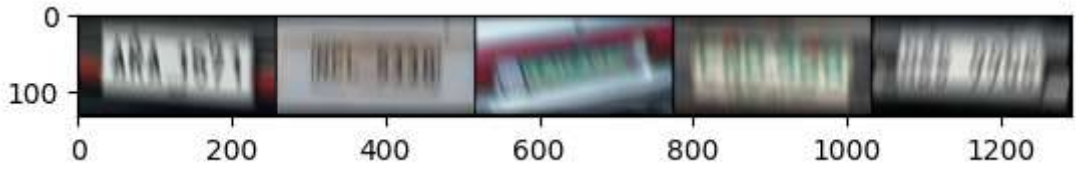
100%|
1250/1250 [18:59<00:00, 1.10it/s]
Training Data

Epoch [3/200], loss_g: 15.1074, loss_d: 1.3616, real_score: 0.0743, fake_score: 0.0242
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [3/200] - Mean PSNR: 14.5194, Mean SSIM: 0.6036, FID: 80.87458038330078

100%|
1250/1250 [19:00<00:00, 1.10it/s]
Training Data

Epoch [4/200], loss_g: 15.0530, loss_d: 0.2520, real_score: 0.9342, fake_score: 0.3437
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data



Epoch [4/200] - Mean PSNR: 14.8999, Mean SSIM: 0.6257, FID: 77.60379028320312

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

Epoch [5/200], loss_g: 15.6081, loss_d: 0.4763, real_score: 0.4459, fake_score: 0.1043
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

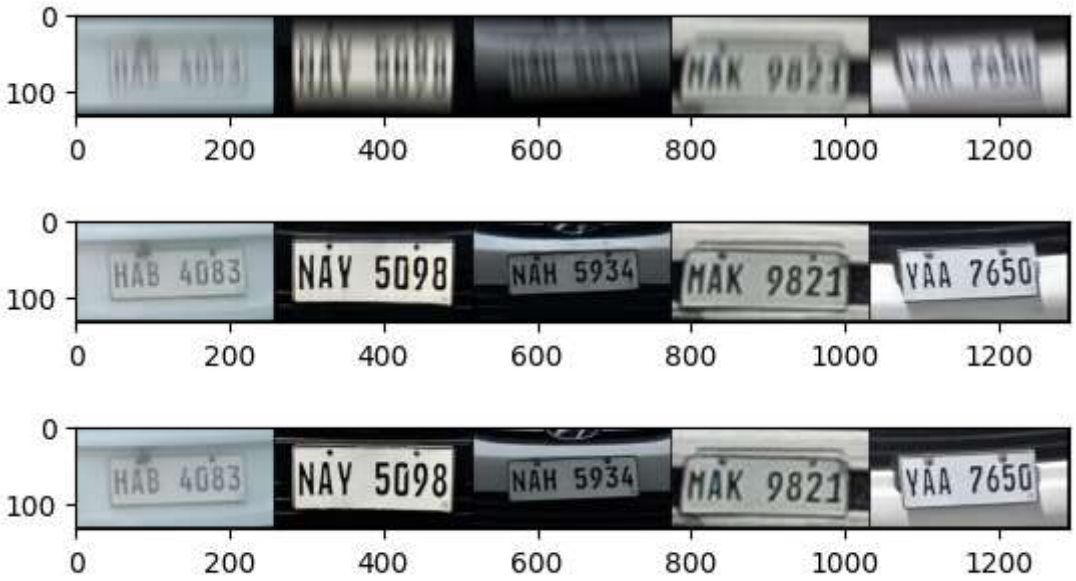
Epoch [5/200] - Mean PSNR: 15.3738, Mean SSIM: 0.6433, FID: 75.07611846923828

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

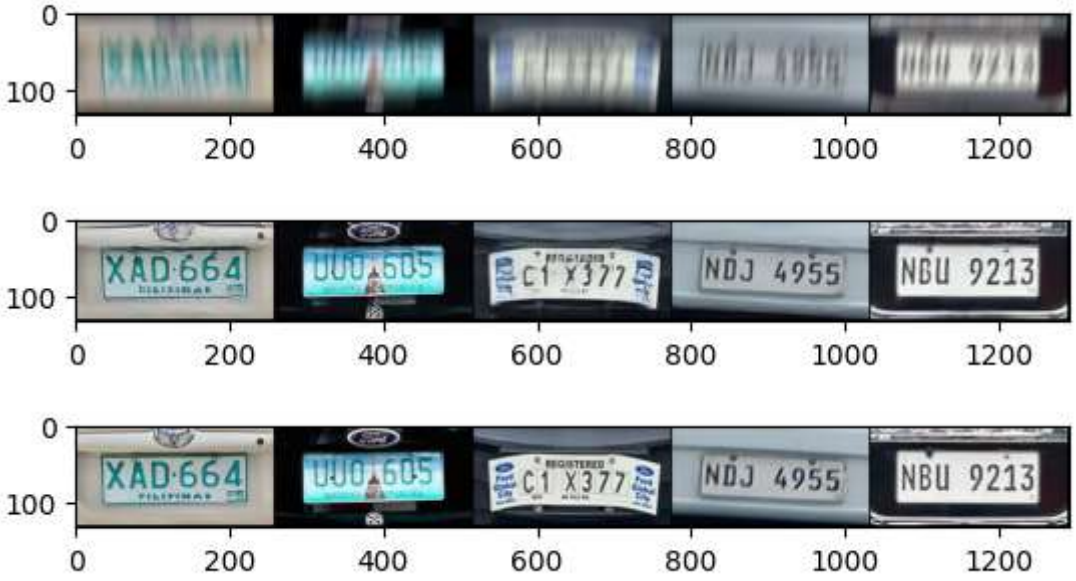
Epoch [6/200], loss_g: 12.2404, loss_d: 0.7754, real_score: 0.2958, fake_score: 0.2357
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [6/200] - Mean PSNR: 15.7332, Mean SSIM: 0.6593, FID: 74.7110595703125

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data



Epoch [7/200], loss_g: 9.9165, loss_d: 0.2968, real_score: 0.7001, fake_score: 0.1938
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data



Epoch [7/200] - Mean PSNR: 15.7611, Mean SSIM: 0.6597, FID: 73.96260833740234

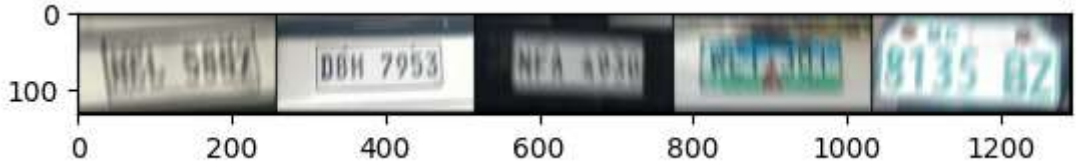
100%|
1250/1250 [18:53<00:00, 1.10it/s]
Training Data

Epoch [8/200], loss_g: 11.6706, loss_d: 0.6853, real_score: 0.7248, fake_score: 0.6163
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [8/200] - Mean PSNR: 16.0421, Mean SSIM: 0.6659, FID: 70.47030639648438

100%|
1250/1250 [18:53<00:00, 1.10it/s]
Training Data

Epoch [9/200], loss_g: 11.1505, loss_d: 0.3645, real_score: 0.9692, fake_score: 0.4768
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data



Epoch [9/200] - Mean PSNR: 16.1134, Mean SSIM: 0.6726, FID: 69.4925308227539

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

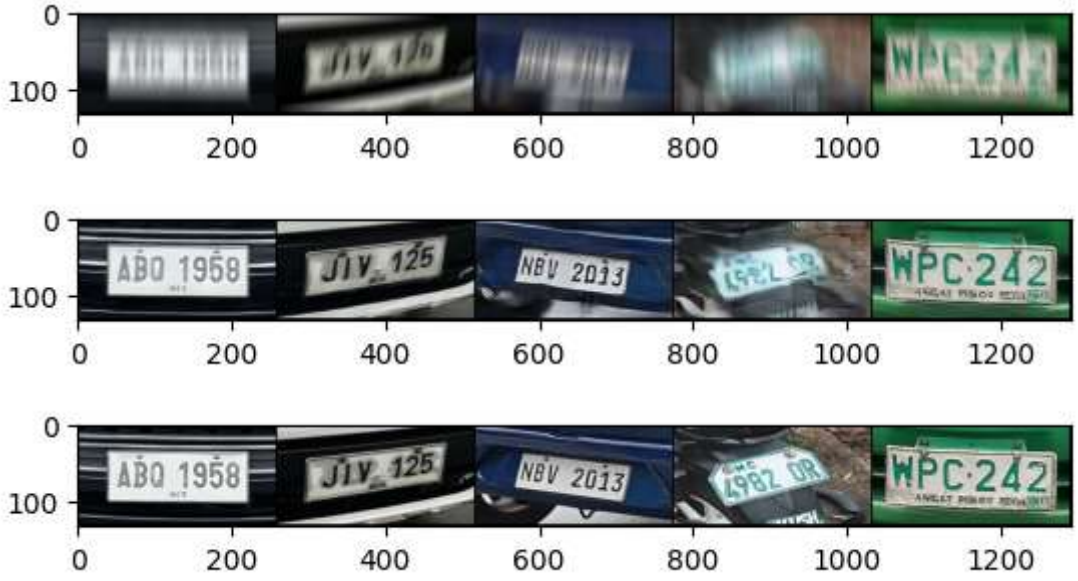
Epoch [10/200], loss_g: 13.0027, loss_d: 0.5434, real_score: 0.3987, fake_score: 0.0805
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [10/200] - Mean PSNR: 16.4206, Mean SSIM: 0.6815, FID: 66.58281707763672
Raw Test Set

Generated Test Set

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

Epoch [11/200], loss_g: 9.7814, loss_d: 0.3423, real_score: 0.9229, fake_score: 0.42
71
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data



Epoch [11/200] - Mean PSNR: 16.7512, Mean SSIM: 0.6970, FID: 65.14450073242188

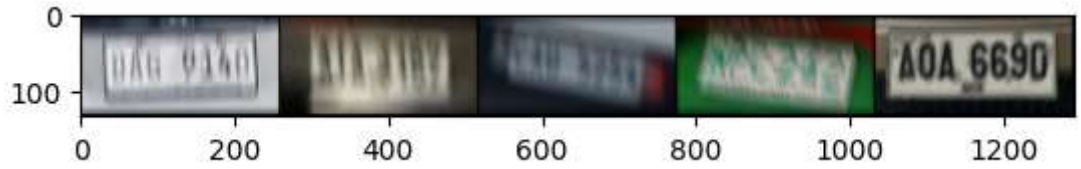
100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

Epoch [12/200], loss_g: 8.4666, loss_d: 1.1367, real_score: 0.1356, fake_score: 0.10
16
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [12/200] - Mean PSNR: 16.6780, Mean SSIM: 0.6930, FID: 64.80681610107422

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

Epoch [13/200], loss_g: 10.5010, loss_d: 0.5308, real_score: 0.4480, fake_score: 0.1809
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data



Epoch [13/200] - Mean PSNR: 17.0537, Mean SSIM: 0.7073, FID: 63.35651397705078

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

Epoch [14/200], loss_g: 12.9691, loss_d: 0.2707, real_score: 0.6972, fake_score: 0.1432
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [14/200] - Mean PSNR: 16.6548, Mean SSIM: 0.6982, FID: 62.315162658691406

100%|
1250/1250 [18:56<00:00, 1.10it/s]
Training Data

Epoch [15/200], loss_g: 10.5756, loss_d: 0.3018, real_score: 0.9042, fake_score: 0.3701
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

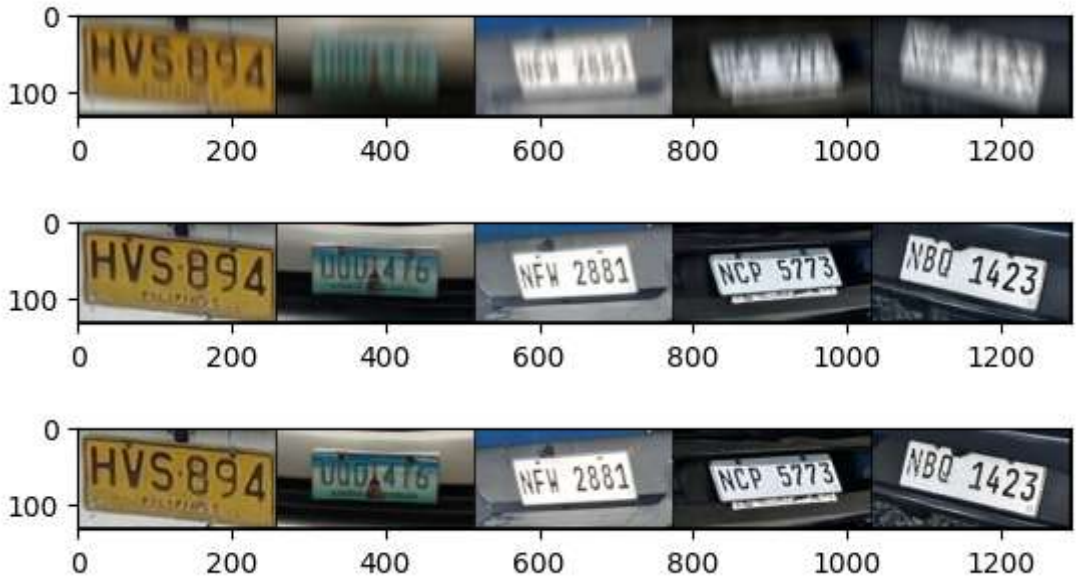
Epoch [15/200] - Mean PSNR: 16.8709, Mean SSIM: 0.7059, FID: 62.25211715698242

100%|
1250/1250 [18:57<00:00, 1.10it/s]
Training Data

Epoch [16/200], loss_g: 7.8930, loss_d: 0.5079, real_score: 0.4588, fake_score: 0.1655
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [16/200] - Mean PSNR: 17.1057, Mean SSIM: 0.7109, FID: 59.854759216308594

100%|
1250/1250 [18:57<00:00, 1.10it/s]
Training Data



Epoch [17/200], loss_g: 10.7309, loss_d: 0.2225, real_score: 0.9284, fake_score: 0.2959
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [17/200] - Mean PSNR: 17.0862, Mean SSIM: 0.7147, FID: 58.92013931274414

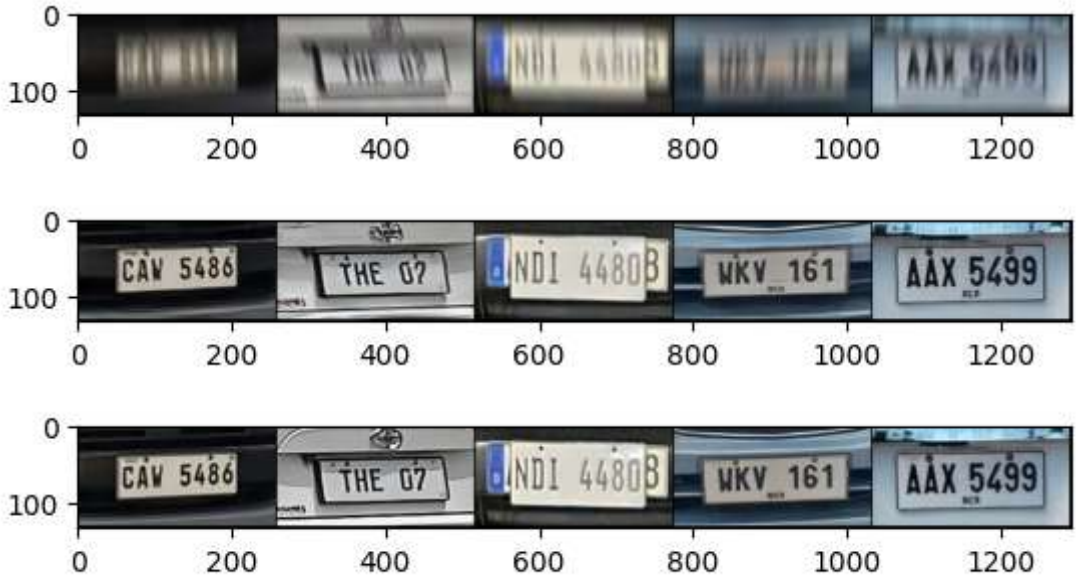
100%|
1250/1250 [18:57<00:00, 1.10it/s]
Training Data

Epoch [18/200], loss_g: 8.8675, loss_d: 0.9229, real_score: 0.1941, fake_score: 0.0491
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [18/200] - Mean PSNR: 17.3312, Mean SSIM: 0.7169, FID: 58.23484420776367

100%|
1250/1250 [18:57<00:00, 1.10it/s]
Training Data

Epoch [19/200], loss_g: 7.1087, loss_d: 0.3116, real_score: 0.8900, fake_score: 0.3719
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data



Epoch [19/200] - Mean PSNR: 17.0421, Mean SSIM: 0.7020, FID: 58.55329513549805
Raw Test Set

Generated Test Set

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

Epoch [20/200], loss_g: 8.6885, loss_d: 0.6190, real_score: 0.3975, fake_score: 0.2071
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [20/200] - Mean PSNR: 17.3240, Mean SSIM: 0.7178, FID: 56.538368225097656

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

Epoch [21/200], loss_g: 9.2200, loss_d: 0.1602, real_score: 0.9003, fake_score: 0.1846
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [21/200] - Mean PSNR: 17.5276, Mean SSIM: 0.7239, FID: 55.275970458984375

100%|
1250/1250 [18:54<00:00, 1.10it/s]
Training Data

Epoch [22/200], loss_g: 8.4237, loss_d: 0.3641, real_score: 0.9452, fake_score: 0.4655
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [22/200] - Mean PSNR: 17.5002, Mean SSIM: 0.7235, FID: 55.5517463684082

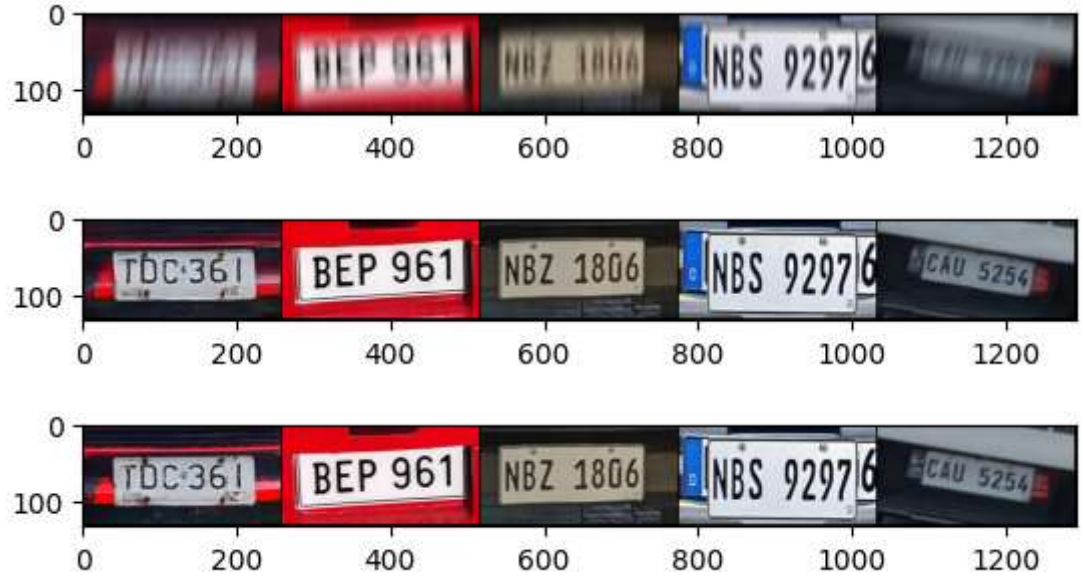
100%|
1250/1250 [18:56<00:00, 1.10it/s]
Training Data

Epoch [23/200], loss_g: 10.4220, loss_d: 0.2291, real_score: 0.8484, fake_score: 0.2371
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [23/200] - Mean PSNR: 17.5685, Mean SSIM: 0.7260, FID: 54.107852935791016

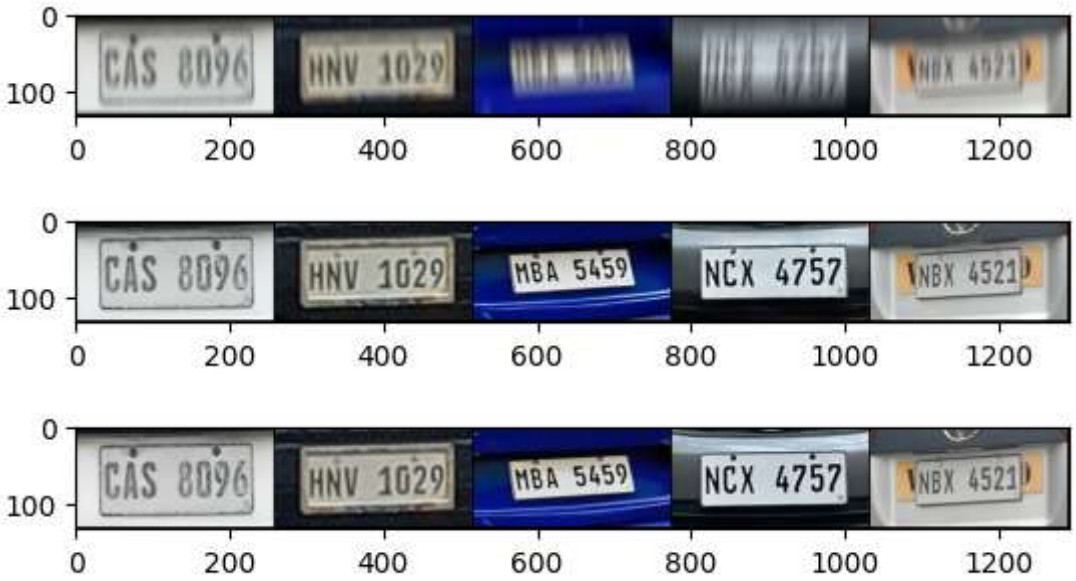
100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

Epoch [24/200], loss_g: 7.9200, loss_d: 0.4020, real_score: 0.5735, fake_score: 0.1784
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data



Epoch [24/200] - Mean PSNR: 17.4107, Mean SSIM: 0.7240, FID: 54.32588577270508

100%|
1250/1250 [18:54<00:00, 1.10it/s]
Training Data

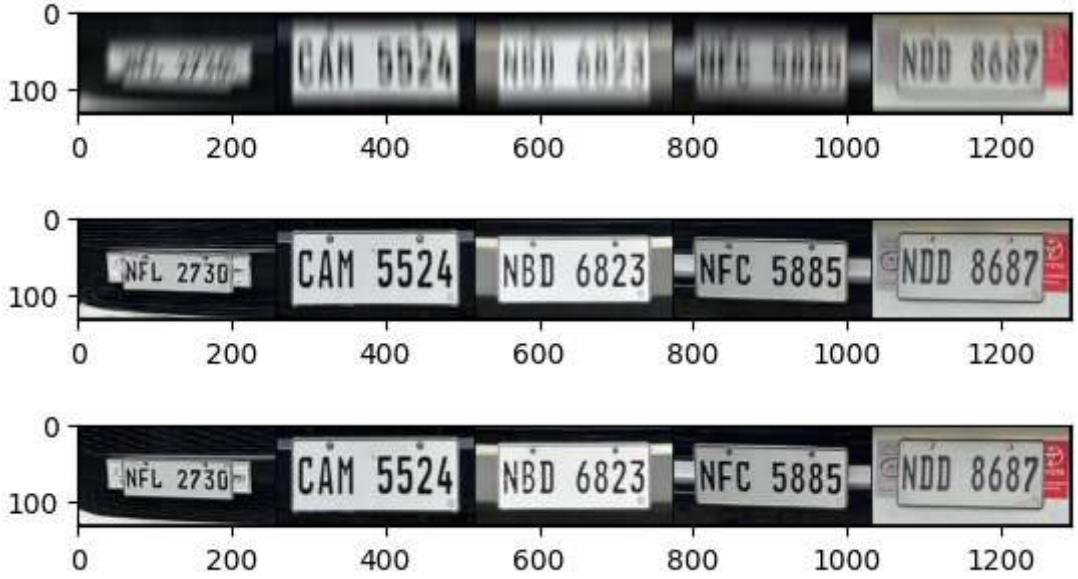


Epoch [25/200], loss_g: 8.1682, loss_d: 0.6896, real_score: 0.3011, fake_score: 0.0653
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [25/200] - Mean PSNR: 17.4726, Mean SSIM: 0.7229, FID: 55.02034378051758

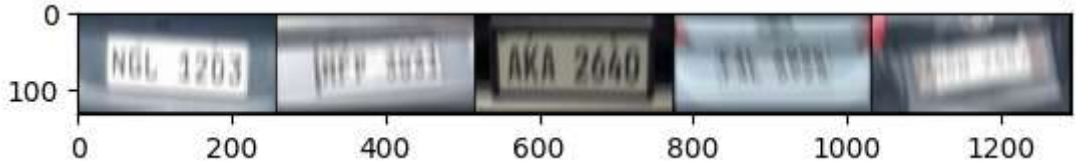
100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

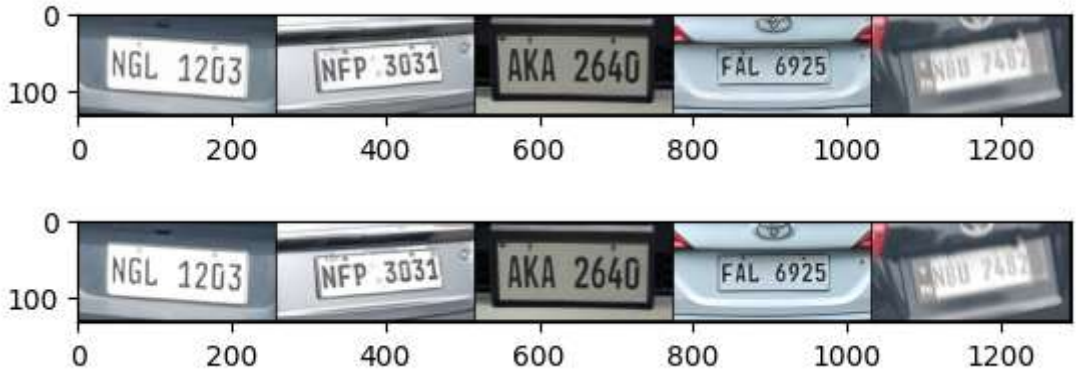
Epoch [26/200], loss_g: 78.0004, loss_d: 0.4434, real_score: 0.5355, fake_score: 0.1602
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data



Epoch [26/200] - Mean PSNR: 17.1482, Mean SSIM: 0.7182, FID: 54.91360092163086

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

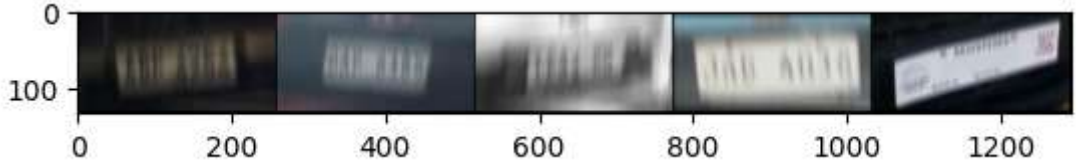




Epoch [27/200], loss_g: 7.8816, loss_d: 0.3581, real_score: 0.7307, fake_score: 0.3005
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [27/200] - Mean PSNR: 17.5700, Mean SSIM: 0.7278, FID: 53.796043395996094

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data



Epoch [28/200], loss_g: 7.8280, loss_d: 0.3021, real_score: 0.9086, fake_score: 0.3781
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

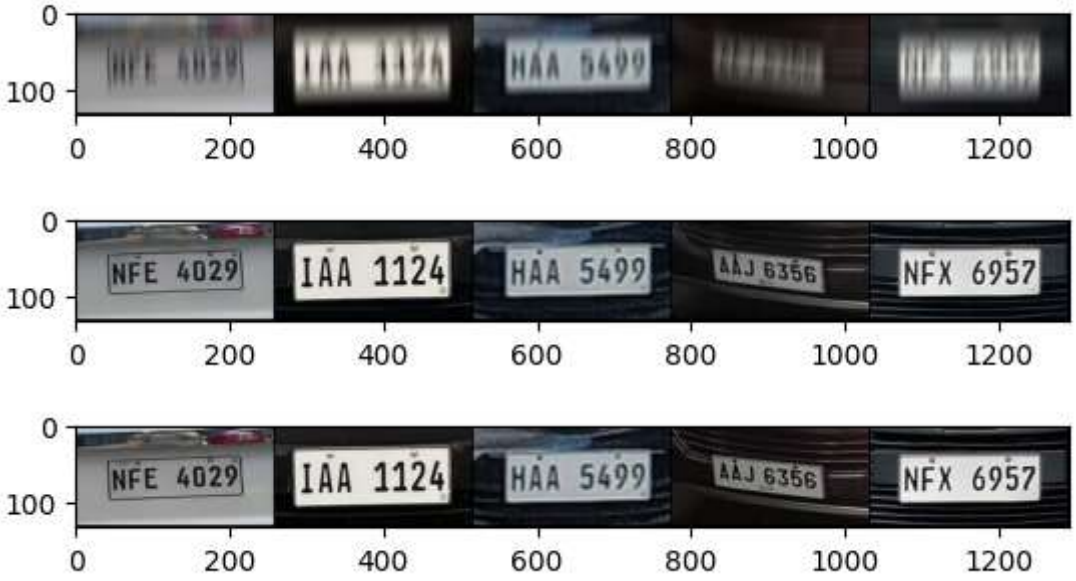
Epoch [28/200] - Mean PSNR: 17.5111, Mean SSIM: 0.7213, FID: 52.690799713134766

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

Epoch [29/200], loss_g: 7.7533, loss_d: 0.2176, real_score: 0.9296, fake_score: 0.28
45
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [29/200] - Mean PSNR: 17.7545, Mean SSIM: 0.7321, FID: 52.654701232910156

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

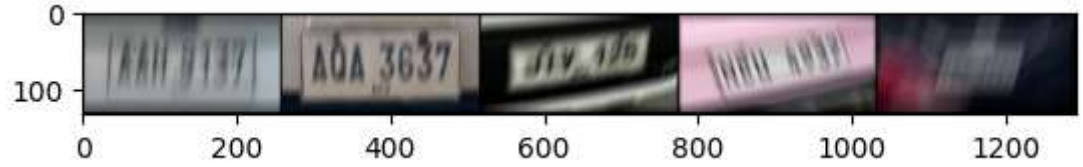


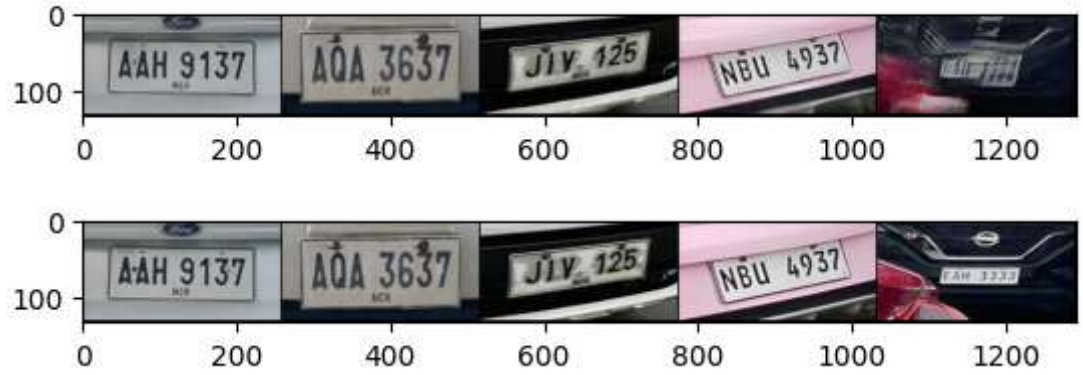
Epoch [30/200], loss_g: 7.0580, loss_d: 0.5414, real_score: 0.8560, fake_score: 0.56
78
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [30/200] - Mean PSNR: 17.6164, Mean SSIM: 0.7283, FID: 52.9904899597168

100%|
1250/1250 [18:55<00:00, 1.10it/s]
Training Data

Epoch [31/200], loss_g: 8.1779, loss_d: 0.3211, real_score: 0.8771, fake_score: 0.3775
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data





Epoch [31/200] - Mean PSNR: 17.6180, Mean SSIM: 0.7337, FID: 52.106353759765625

100%|
1250/1250 [19:26<00:00, 1.07it/s]
Training Data

Epoch [32/200], loss_g: 7.5448, loss_d: 0.4539, real_score: 0.7369, fake_score: 0.4199
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

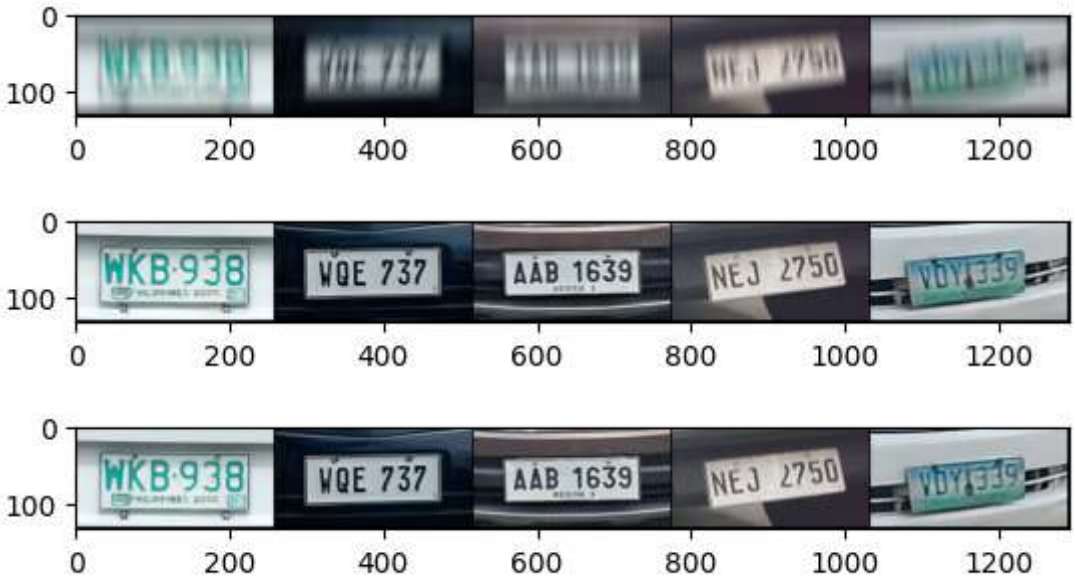
Epoch [32/200] - Mean PSNR: 17.5873, Mean SSIM: 0.7309, FID: 51.80231475830078

100%|
1250/1250 [19:47<00:00, 1.05it/s]
Training Data

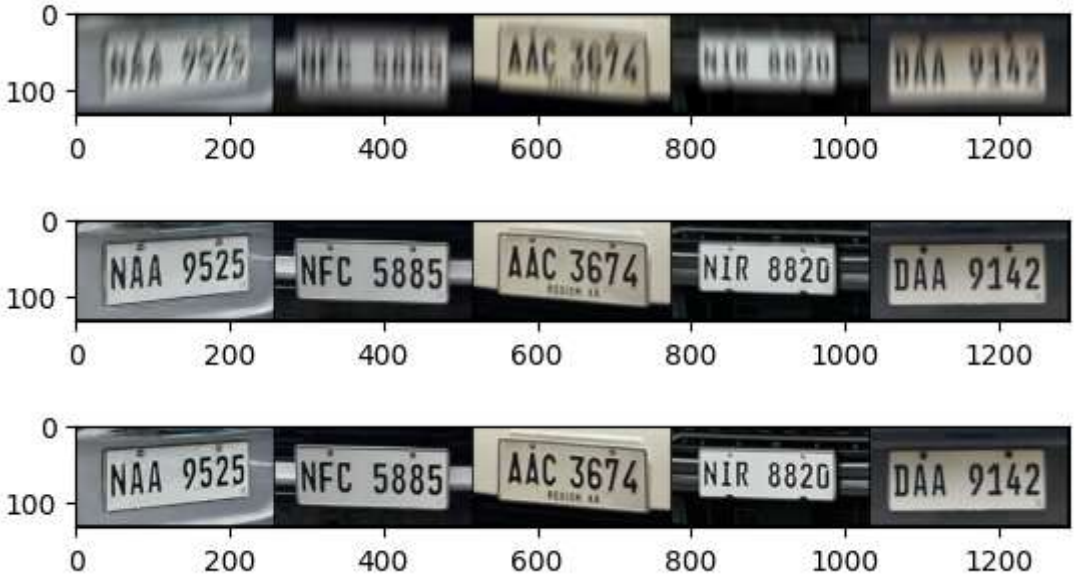
Epoch [33/200], loss_g: 8.2800, loss_d: 0.3370, real_score: 0.6413, fake_score: 0.1717
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data

Epoch [33/200] - Mean PSNR: 17.7610, Mean SSIM: 0.7363, FID: 50.74321746826172

100%|
1250/1250 [19:42<00:00, 1.06it/s]
Training Data



Epoch [34/200], loss_g: 7.7352, loss_d: 0.8898, real_score: 0.2138, fake_score: 0.0524
Current LRs — Generator: 0.000200, Discriminator: 0.000200
Validation Data



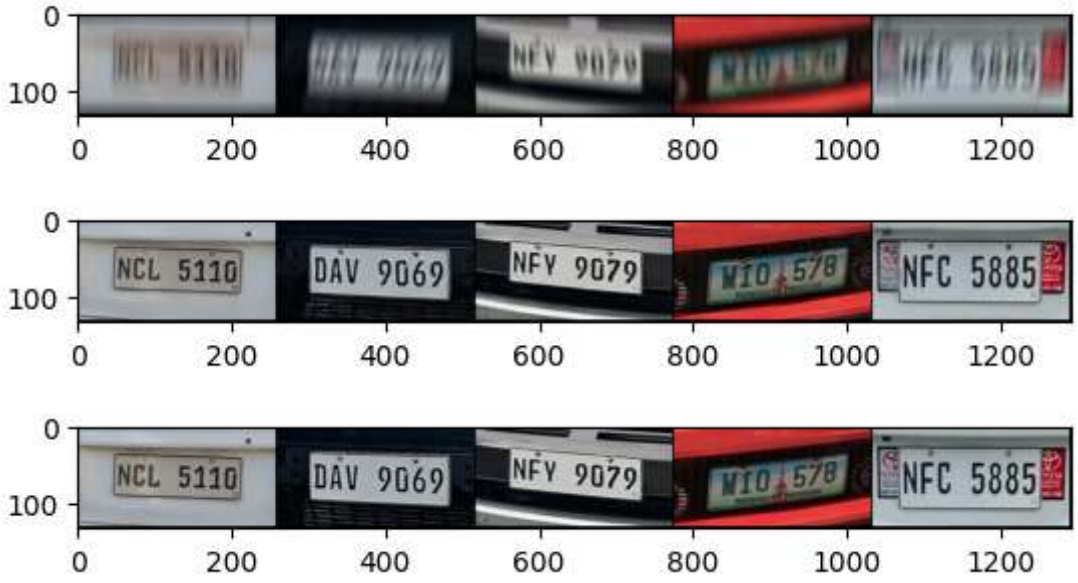
Epoch [34/200] - Mean PSNR: 17.8010, Mean SSIM: 0.7341, FID: 50.371219635009766

Epoch 00034: reducing learning rate of group 0 to 1.0000e-04.

100%|
1250/1250 [19:34<00:00, 1.06it/s]

```
Training Data
```


```
Epoch [35/200], loss_g: 7.7428, loss_d: 0.3664, real_score: 0.6519, fake_score: 0.23
62
Current LRs — Generator: 0.000200, Discriminator: 0.000100
Validation Data
```




```
Epoch [35/200] - Mean PSNR: 17.7283, Mean SSIM: 0.7344, FID: 50.69340515136719
```


```
  5%|▌
| 66/1250 [01:09<18:57,  1.04it/s]
```

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')
```

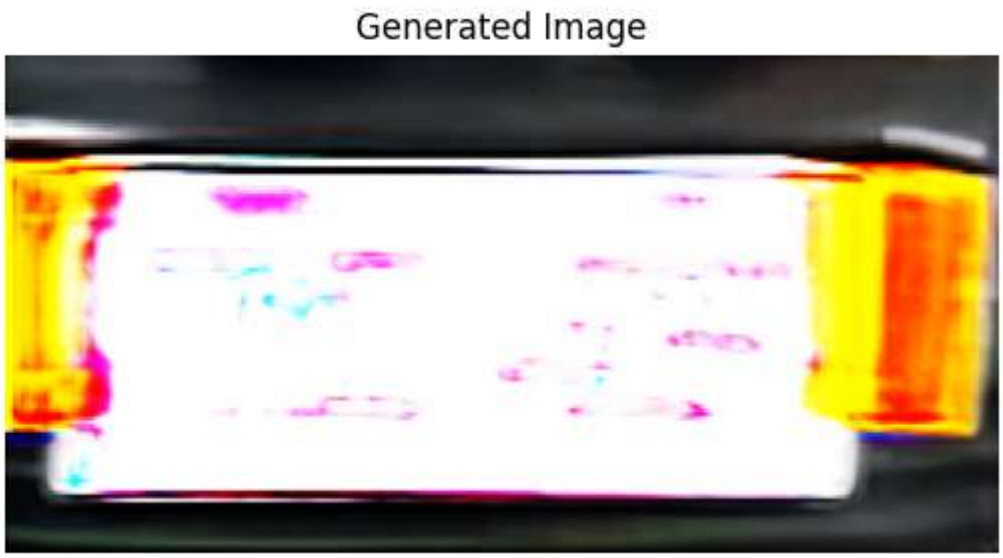
Model Evaluation

In [56]:
```
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())
```

```
Mean PSNR: 15.653532288291238
Mean SSIM: 0.6187068169767206
FID: 67.66695404052734
```

In [33]:
```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 [31]:
```python
deblur_image(os.path.join(DATA_DIR, 'h_blur', 'v2', 'test', 'h_blur', '129.jpg'), g
```



In [71]:
```python
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 [33]:
```
torch.cuda.empty_cache()
```

Load Model

In [50]:
```
MODELS_DIR = os.path.join('..', '..', '..', 'models')
```

In [55]:
```
generator = AttentionResUNetGenerator()
loaded_g_wts = torch.load("C:\\Thesis\\LiPAD with Paddle\\models\\gan_v_blur.pt")
generator.load_state_dict(loaded_g_wts)
generator = to_device(generator, device)

#discriminator = PatchDiscriminator()
#loaded_d_wts = torch.load('[RAU-NET OCRDET LR]discriminator-Datasetv2-80 epoch.pt'
#discriminator.load_state_dict(loaded_d_wts)
#discriminator = to_device(discriminator, device)
```

In [34]:
```
checkpoint = torch.load('[RAU-NET OCRDET LR]Checkpoint-Logs-Datasetv2-80 epoch.pt')
checkpoint2 = torch.load('[RAU-NET OCRDET LR 2]Checkpoint-Logs-Datasetv2-145 epoch.
#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]:
```
len(checkpoint)
```

Out[39]: 14

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

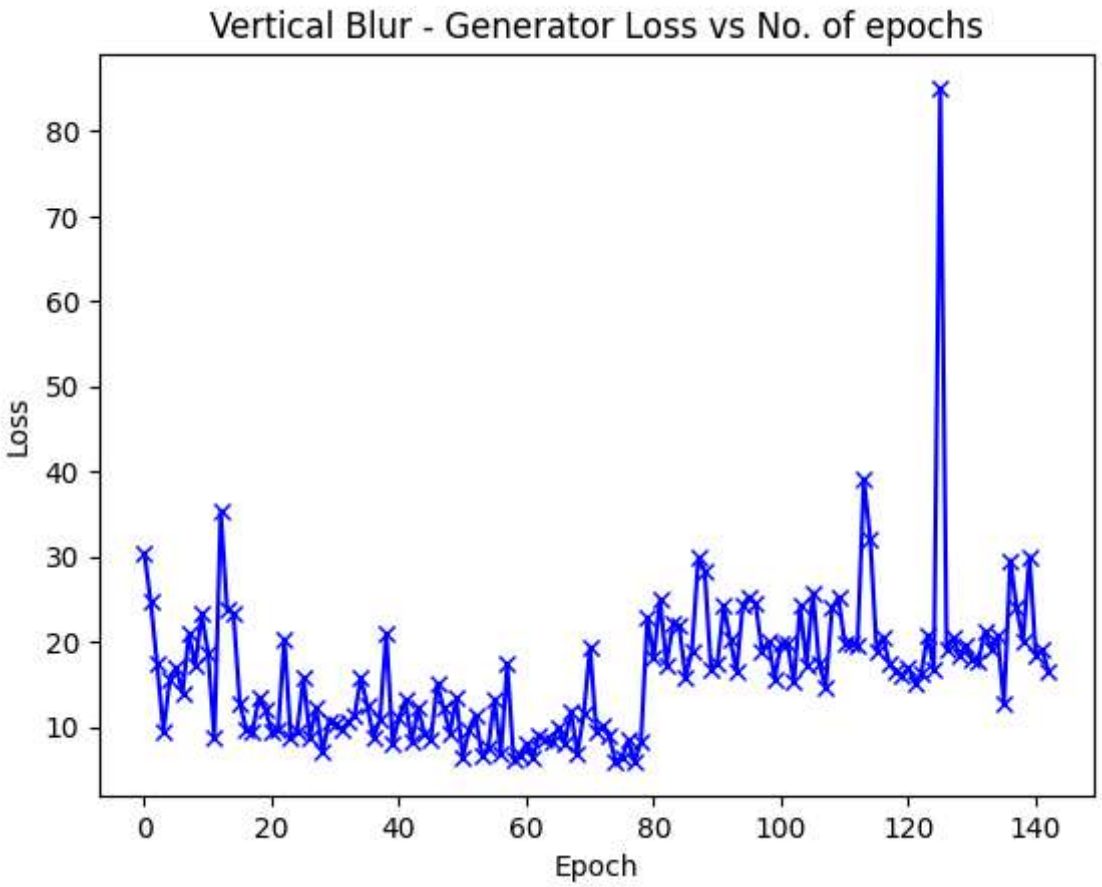
Save Reconstructed Test Set

In [39]:
```
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)
```

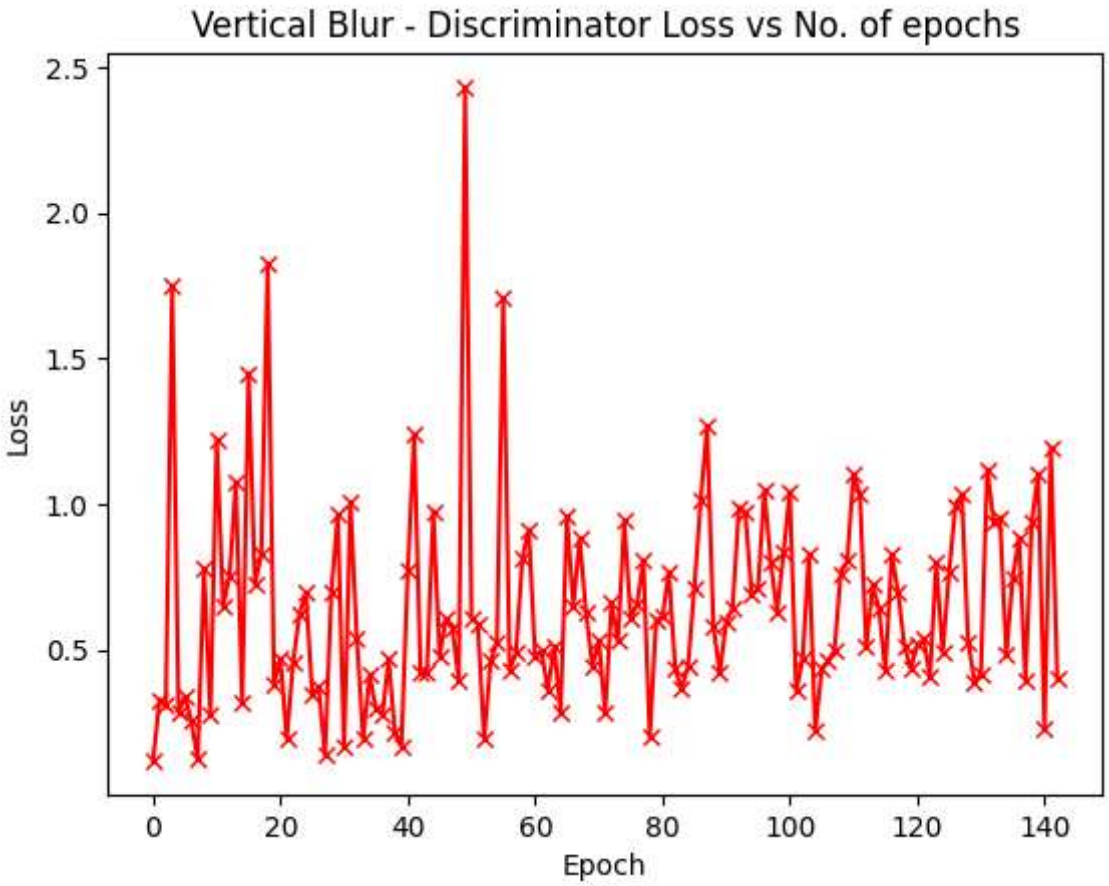
In [39]:
```
from collections import OrderedDict

for key, value in checkpoint2.items():
    if key not in checkpoint:
        checkpoint[key] = value
    else:
        if isinstance(value, list):
            checkpoint[key].extend(value)  # concatenate lists
        elif isinstance(value, OrderedDict):
            checkpoint[key].update(value)  # merge OrderedDicts
        else:
            # fallback, keep checkpoint2 value
            checkpoint[key] = value
```

In [40]:
```
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('Vertical Blur - Generator Loss vs No. of epochs')

plot_g_losses(checkpoint)
```

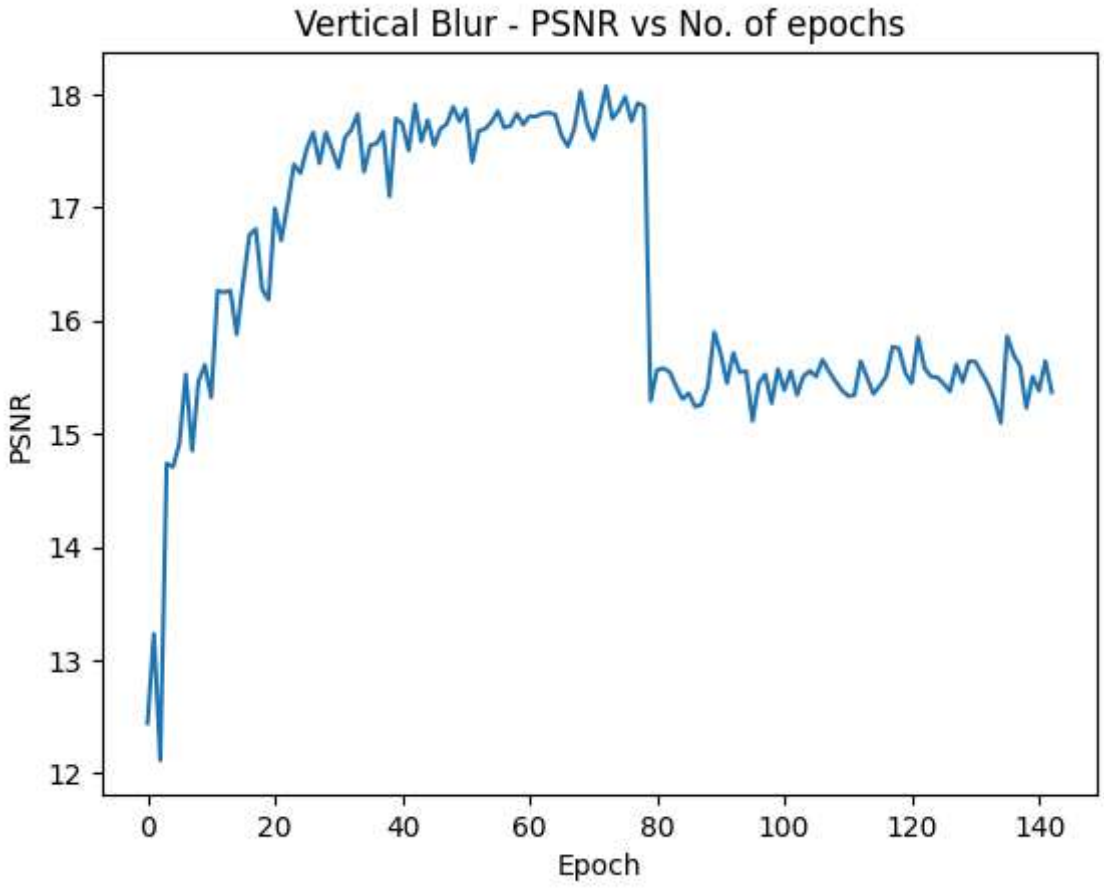


In [45]:
```
def plot_d_losses(history):
    losses_d = [x for x in history['losses_d']]
```

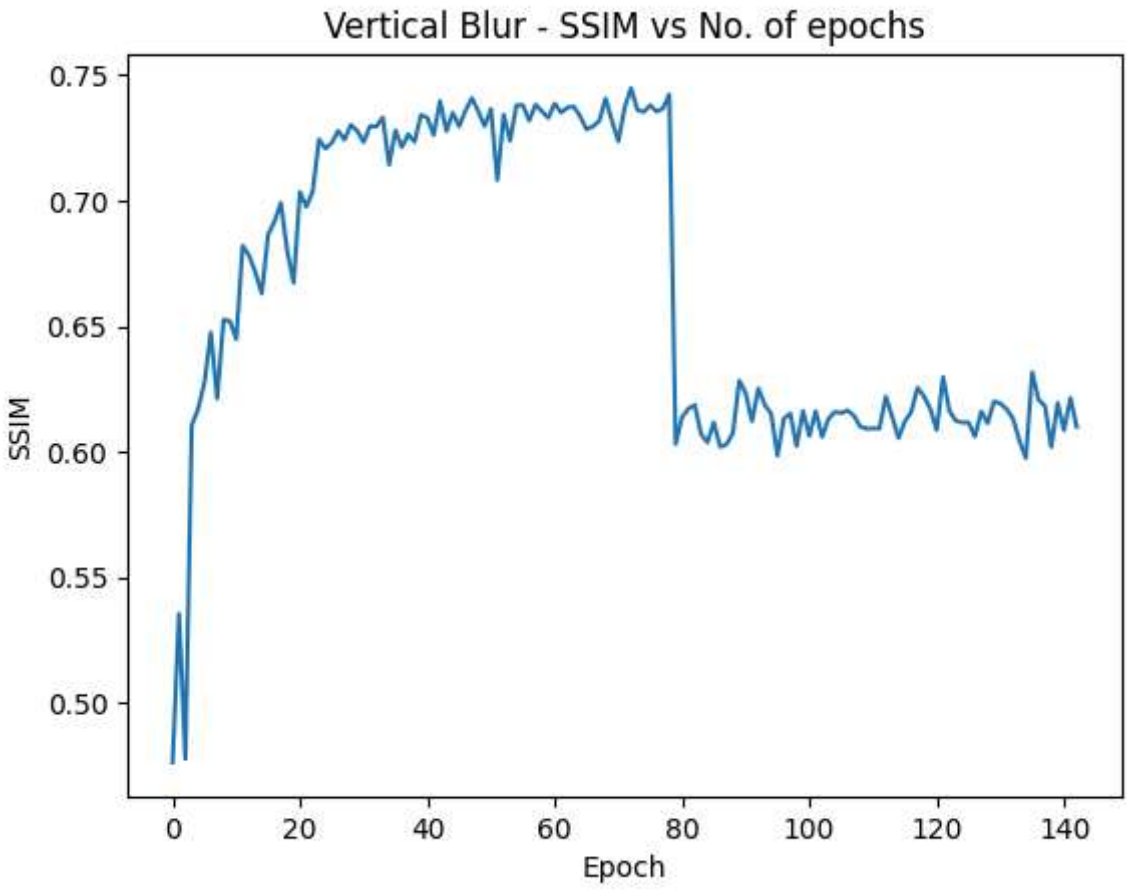
```
    plt.plot(losses_d, '-rx')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Vertical Blur - Discriminator Loss vs No. of epochs')

plot_d_losses(checkpoint)
```



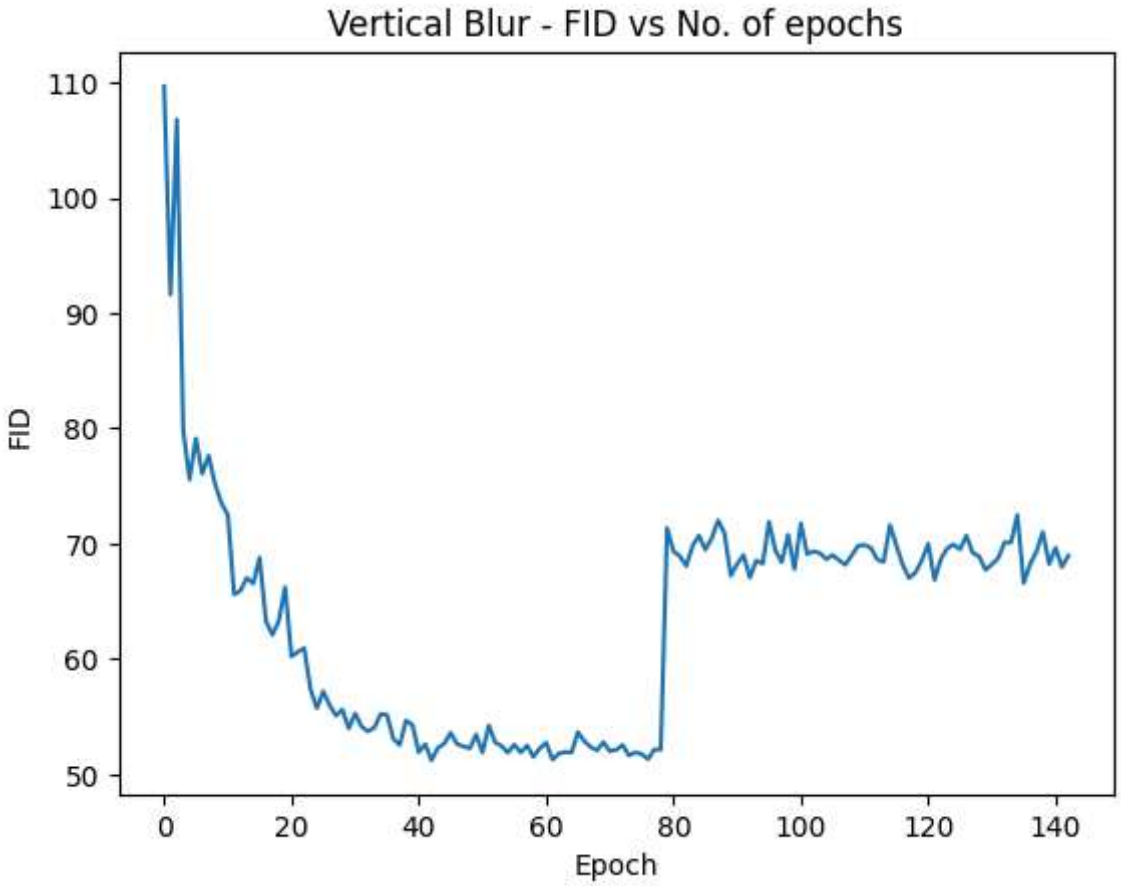
In [46]:
```
def plot_psnrs(history):
    psnrs = [x for x in history['psnrs']]
    plt.plot(psnrs)
    plt.xlabel('Epoch')
    plt.ylabel('PSNR')
    plt.title('Vertical Blur - PSNR vs No. of epochs')

plot_psnrs(checkpoint)
```



In [47]:
```
def plot_ssims(history):
    psnrs = [x for x in history['ssims']]
    plt.plot(psnrs)
    plt.xlabel('Epoch')
    plt.ylabel('SSIM')
    plt.title('Vertical Blur - SSIM vs No. of epochs')

plot_ssims(checkpoint)
```



In [48]:
```
def plot_fids(history):
    psnrs = [x for x in history['fids']]
    plt.plot(psnrs)
    plt.xlabel('Epoch')
    plt.ylabel('FID')
    plt.title('Vertical Blur - FID vs No. of epochs')

plot_fids(checkpoint)
```


Vertical Blur - FID vs No. of epochs
FID
Epoch
110
100
90
80
70
60
50
0
20
40
60
80
100
120
140

In [ ]: