Object Removal and Inpainting using Generative Adverserial Network

Project by: Anna Wang, Jimmy Yang and Philip Chen

Abstract

This project aims to explore the applications of using a Generative Adversarial Network (GANs) to remove objects and inpaint over specific regions in images. The art of object inpainting is a technique used to seamlessly insert or remove objects within an image, often used for image editing or restoration. Given a masked area, we wish to fill in the masked-area with new content that blends realistically with the rest of the image.

Object removal and image inpainting are inherently challenging tasks in computer vision. To help us tackle this project, we employ the use of a special variant of GANs, a Deep Convolutional Generative Adversarial Network (DCGANs). With the help of convolutional layers for image processing, we architected a generator to produce convincing image completions, while the discriminator learns to distinguish between authentic images and the generator's images. As the two models compete in an adversarial process, they both improve their respective abilities, leading to a better trained generator to inpaint masked images.

To evaluate performance, a curated dataset of cat images was used, with manually masked holes of the same size inserted in the middle of all images. This choice of dataset allows for testing the model's ability to handle fine textures like fur and preserve anatomical consistency. The generator performs reasonably well at reconstructing altered parts of the cats body, but struggles when it comes to facial features.

Ultimately, this project highlighted the complexity of the inpainting task. We conclude with several observations and potential directions for improving the model's performance in future work.

Team Members and Contributions

- Anna Wang (20898072, aj2wang@uwaterloo.ca): I researched and found the dataset of cats and dogs from hugging-face, incorporating, importing and setting up the data for further processing in our notebook. I worked alongside Jimmy to split our dataset into cats and dogs and then the cat dataset into our training, validation and test splits. I also researched with Philip on the GANs architecture and wrote the code for the Discriminator. Furthermore, I worked alongside Philip to design and write the training function four our model. As a group, we collectively worked on fine-tuning our models, analyzing our results, iterating on improvements as well as the creation of the reports within our notebook.
- Jimmy Yang (20890430, jj7yang@uwaterloo.ca): I helped curate the dataset of cats to be used in model training. I worked with Anna to split our original dataset into images of cats and defined a custom CatDataset class to help us insert square masks for training purposes. I also explored various model training methods against our dataset to help decide what image resolution to use and how to architect our models. Then, together with the group I contributed to fine-tuning our models, analyzing our results, and iterating on improvements.
- Philip Chen (20902971 , p242chen@uwaterloo.ca): I helped write the function for creating the masks. I also researched and helped design the architecture for our GAN model, as well as writing the code for the Generator. I also worked with Anna to design and write the training function for the model. Finally, together with the group I contributed to fine-tuning our models, analyzing our results, and iterating on improvements.

1. Code Libraries

The following external Python libraries were essential to the development and implementation of this project:

- NumPy: Used for efficient numerical operations and array manipulation throughout the project, particularly when handling image masks, pixel data, and preprocessing pipelines.
- Matplotlib: Used for visualizing training progress, loss curves, and before-and-after comparisons of image inpainting results. It plays a key role in debugging and showcasing the model's performance.
- PyTorch: Foundational deep learning framework for this project. It enabled the construction and training of the GAN architecture through its neural network modules and optimization tools.

```
# Standard libraries for math and plotting
import numpy as np
import matplotlib.pyplot as plt

# Standard PyTorch for network training
import torch
import torch.nn as nn
from torch import optim
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms
from torchvision.transforms import v2
import torchvision.utils as vutils

# Other utilities and misc
import os
from copy import deepcopy
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
```

```
device(type='cuda')
```

2. Importing Dataset

Our dataset comes from Hugging Face Dataset cats_vs_dogs: https://huggingface.co/datasets/microsoft/cats_vs_dogs. From the dataset, we filter for images of cats and split up training, validation, and testing sets. We chose this dataset because of the quality and large number of samples within the dataset with a diverse range of features and expressions of cats in images. This will provide a robust dataset for our GANs to train over.

We install the standard `datasets` library to use the dataset in our training.

```
# uncomment cli login if need to login
# !huggingface-cli login
!pip install datasets
```

```
Collecting dill<0.3.9,>=0.3.0 (from datasets)
  Downloading dill-0.3.8-py3-none-any.whl.metadata (10 kB)
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packa
Requirement already satisfied: requests>=2.32.2 in /usr/local/lib/python3.11/
Requirement already satisfied: tqdm>=4.66.3 in /usr/local/lib/python3.11/dist-
Collecting xxhash (from datasets)
  Downloading xxhash-3.5.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x8
Collecting multiprocess<0.70.17 (from datasets)
  Downloading multiprocess-0.70.16-py311-none-any.whl.metadata (7.2 kB)
Collecting fsspec<=2024.12.0,>=2023.1.0 (from fsspec[http]<=2024.12.0,>=2023.
  Downloading fsspec-2024.12.0-py3-none-any.whl.metadata (11 kB)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.11/dist-packa
Requirement already satisfied: huggingface-hub>=0.24.0 in /usr/local/lib/pytho
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-pa
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/pytho
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.11/
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.11/dis
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.11,
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.
Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.11/
Requirement already satisfied: yarl<2.0,>=1.17.0 in /usr/local/lib/python3.11,
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/py
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/pyth
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.1
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.1
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/pytho
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/di
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-pac
Downloading datasets-3.5.0-py3-none-any.whl (491 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 491.2/491.2 kB 9.3 MB/s eta 0:00:
Downloading dill-0.3.8-py3-none-any.whl (116 kB)
```

```
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 116.3/116.3 kB 10.5 MB/s eta 0:00
Downloading fsspec-2024.12.0-py3-none-any.whl (183 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 183.9/183.9 kB 17.0 MB/s eta 0:00
Downloading multiprocess-0.70.16-py311-none-any.whl (143 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 143.5/143.5 kB 13.7 MB/s eta 0:00
Downloading xxhash-3.5.0-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 194.8/194.8 kB 20.3 MB/s eta 0:00
Installing collected packages: xxhash, fsspec, dill, multiprocess, datasets
  Attempting uninstall: fsspec
    Found existing installation: fsspec 2025.3.2
    Uninstalling fsspec-2025.3.2:
      Successfully uninstalled fsspec-2025.3.2
ERROR: pip's dependency resolver does not currently take into account all the
torch 2.6.0+cu124 requires nvidia-cublas-cu12==12.4.5.8; platform_system == "
torch 2.6.0+cu124 requires nvidia-cuda-cupti-cu12==12.4.127; platform_system =
torch 2.6.0+cu124 requires nvidia-cuda-nvrtc-cu12==12.4.127; platform_system =
torch 2.6.0+cu124 requires nvidia-cuda-runtime-cu12==12.4.127; platform_syster
torch 2.6.0+cu124 requires nvidia-cudnn-cu12==9.1.0.70; platform_system == "L
torch 2.6.0+cu124 requires nvidia-cufft-cu12==11.2.1.3; platform_system == "L
torch 2.6.0+cu124 requires nvidia-curand-cu12==10.3.5.147; platform_system ==
torch 2.6.0+cu124 requires nvidia-cusolver-cu12==11.6.1.9; platform_system ==
torch 2.6.0+cu124 requires nvidia-cusparse-cu12==12.3.1.170; platform_system =
torch 2.6.0+cu124 requires nvidia-nvjitlink-cu12==12.4.127; platform_system ==
gcsfs 2025.3.2 requires fsspec==2025.3.2, but you have fsspec 2024.12.0 which
Successfully installed datasets-3.5.0 dill-0.3.8 fsspec-2024.12.0 multiproces
```

2.1 Importing Cats vs. Dogs Dataset

```
from datasets import load_dataset

ds = load_dataset("microsoft/cats_vs_dogs")

print(ds['train'])
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: Use
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tak
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access pu
  warnings.warn(
```

README.md: 100% 8.16k/8.16k [00:00<00:00, 797kB/s]

train-00000-of-00002.parquet: 100% 330M/330M [00:01<00:00, 257MB/s]

train-00001-of-00002.parquet: 100% 391M/391M [00:01<00:00, 261MB/s]

Generating train split: 100% 23410/23410 [00:01<00:00, 18826.98 examples/s]

2.2 Filtering Dataset by Cats and Dogs

Since this dataset contains pictures of both cats and dogs, we must filter our dataset so our final dataset includes only cat photos.

```
# filtering data by cats and dogs
data = ds['train']
dogs = data.filter(lambda isDog: isDog['labels'] == 1)
cats = data.filter(lambda isDog: isDog['labels'] == 0)
print("Dataset size: " + str(len(data)))

print("Total dataset size:", len(data))
print("Dogs:", len(dogs))
print("Cats:", len(cats))
```

```
Filter: 100%    23410/23410 [00:26<00:00, 866.24 examples/s]
Filter: 100%    23410/23410 [00:26<00:00, 864.69 examples/s]
Dataset size: 23410
```

2.3 Splitting each class into training, validation and test sets

We split our dataset into training, validation, and test sets to ensure robust and unbiased model development. The training set is used to teach the GAN how to perform image imprinting and object removal on cat photos. The validation set helps us tune hyperparameters and monitor performance during training, allowing us to prevent overfitting. Finally, the test set provides an unbiased evaluation of how well the trained GAN generalizes to new, unseen data. In the code, the dataset is split into 80% training and 20% test, with the test portion further divided equally to create separate validation and test sets. We print the size of each dataset to show the size of the splits.

```
# Split the dataset
train_test_split = cats.train_test_split(test_size=0.2, seed=42)
test_val_split = train_test_split['test'].train_test_split(test_size=0.5, seed=42
train_dataset = train_test_split['train']['image']
val_dataset = test_val_split['train']['image']
test_dataset = test_val_split['test']['image']

print(f"Training dataset size: {len(train_dataset)}")
print(f"Validation dataset size: {len(val_dataset)}")
print(f"Test dataset size: {len(test_dataset)}")
train_dataset[0]
```

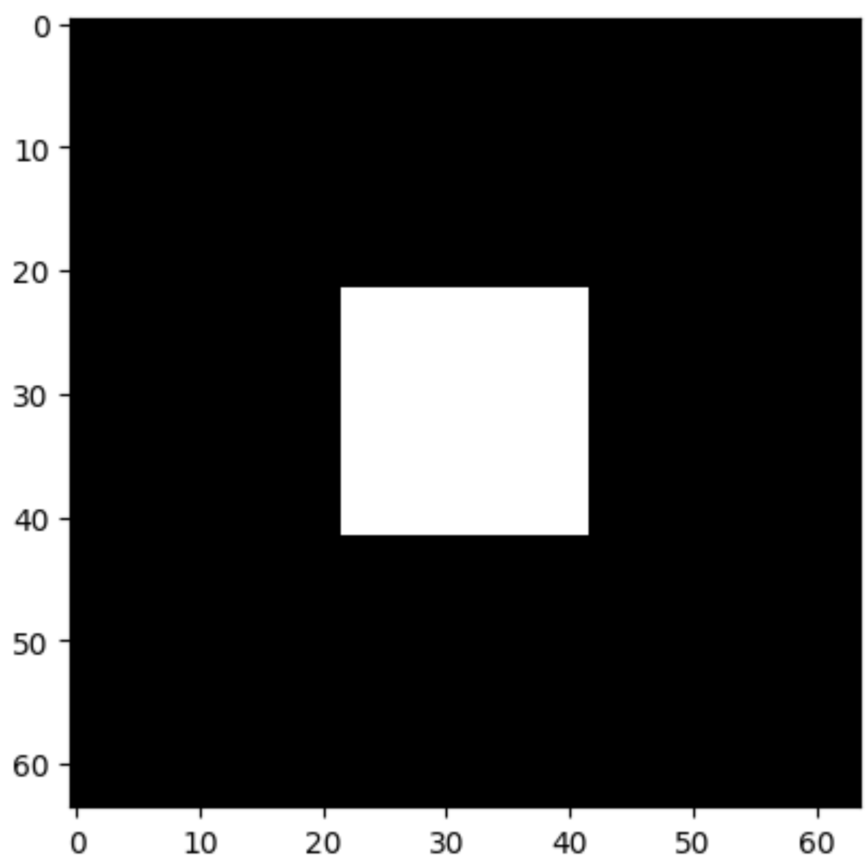
```
Training dataset size: 9392
Validation dataset size: 1174
Test dataset size: 1175
```

3. Creating Data Loaders

3.1 Masks

In order to create training samples for our model, we create the function `create_masks` that cuts out the center of an image of size `hole_size`. We acknowledge that designing a network model for inpainting has a number of challenges. To simplify the project, we assume that the holes are always square, and that they are always in the center and have the same size. The central region is surrounded by image context on all four sides—top, bottom, left, and right. This gives the model rich, balanced context to infer what the missing content should look like. During training, consistently masking the center allows the model to focus its capacity on learning to inpaint a specific region. This makes convergence faster and often leads to better initial performance compared to random or irregular masking. Future improvements to the model would be to allow for custom created masks.

```python
def create_masks(num_masks):
    hole_size = 20
    im_size = 64

    x = int((im_size - hole_size) / 2.0)
    y = int((im_size - hole_size) / 2.0)
    mask = torch.zeros((1, im_size, im_size))
    mask[0, y : y + hole_size, x : x + hole_size] = 1
    masks = mask.repeat_interleave(num_masks, dim=0)

    return masks.unsqueeze(1)

plt.imshow(create_masks(1)[0][0], cmap='gray')
plt.show()
```



3.2 Creating Cat Dataset Definition

The following class defines our Cat Dataset

```
class CatDataset(Dataset):
    def __init__(self, all_imgs, transforms=None) -> None:
        super().__init__()
        self.transforms = transforms
        self.all_imgs = all_imgs

    def __len__(self):
        return len(self.all_imgs)

    def __getitem__(self, index):
        image = self.all_imgs[index]

        # 1. Transform the image as needed
        transformed_img = self.transforms(image)

        if transformed_img.shape[0] == 1:
            transformed_img = transformed_img.repeat(3, 1, 1)

        # 2. Create copy of image before masking
        ground_truth_image = deepcopy(transformed_img)

        # 3. Create center mask depending of transformed image
        mask = create_masks(1)[0]
        transformed_img = (1 - mask) * transformed_img

        return transformed_img, ground_truth_image
```

3.3 Transformations

While composing our dataset, we experiments with several transformations, including what image resolution to train on, various crops, and flips. We landed on the resolution of 64x64 because it would allow the images to retain just enough detail to make out facial features (ie. eyes, nose, mouth), as opposed to smaller resolutions like 32x32 that would make the images almost unrecognizable as cats.

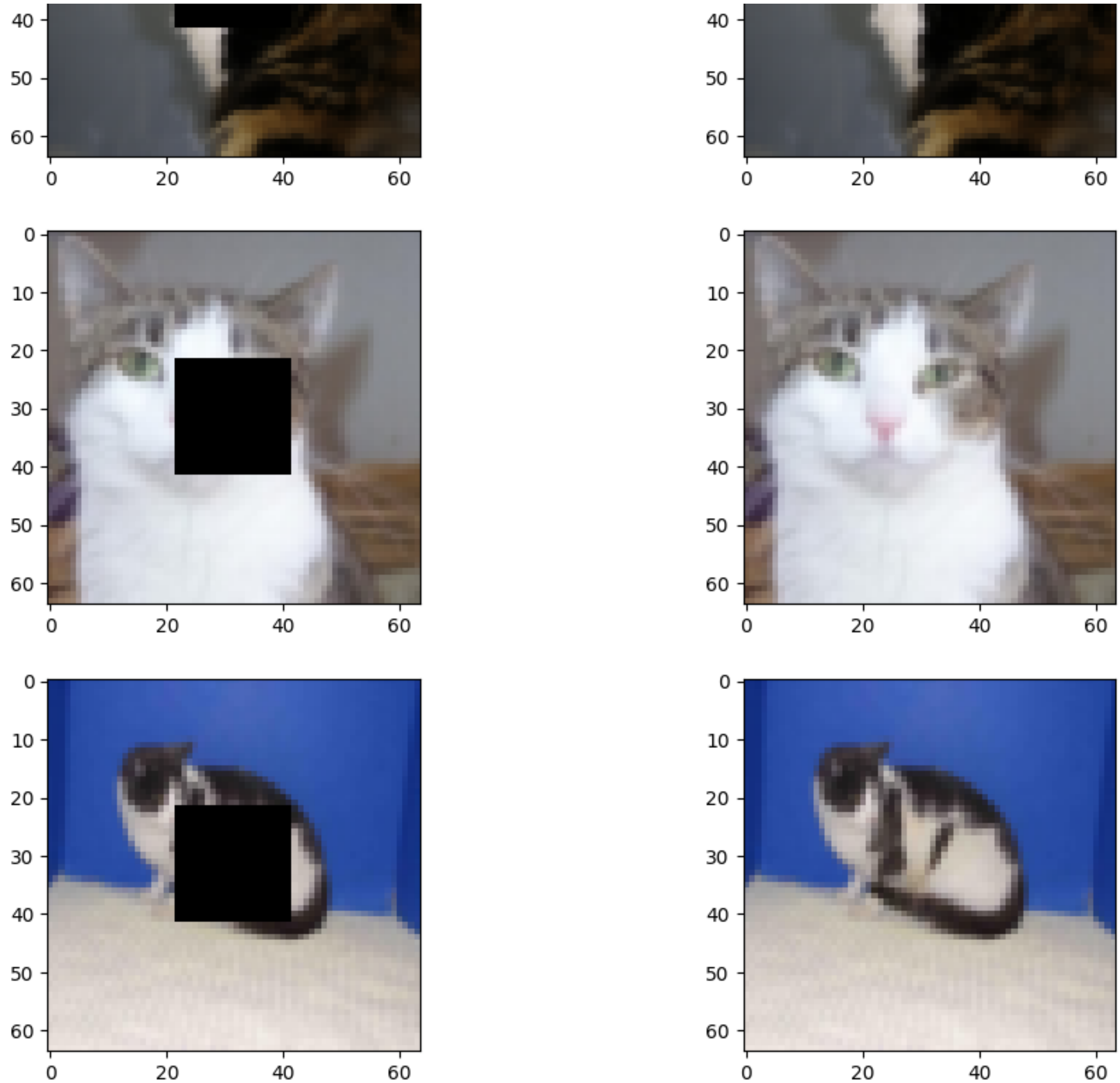
Our original cat dataset consists of images with a wide range of resolutions and aspect ratios, which can introduce inconsistency during training. By applying transformations that resize all images to a uniform resolution of 64x64, we not only standardize the input data for our model but also ensure that each image consistently highlights key facial features—such as the eyes, nose, and mouth—that are essential for effective image imprinting and object removal. This preprocessing step improves model performance and stability by reducing variability unrelated to the task.

```
transforms = v2.Compose([
    v2.Compose([v2.ToImage(), v2.ToDtype(torch.float32, scale=True)]),
    v2.Resize(size=(64, 64))
])

transformed_train_dataset = CatDataset(train_dataset, transforms=transforms)
transformed_val_dataset = CatDataset(val_dataset, transforms=transforms)
transformed_test_dataset = CatDataset(test_dataset, transforms=transforms)
```

3.4 Explore Dataset

To visually confirm that our preprocessing pipeline is working as intended, we display several pairs of training images alongside their corresponding ground truth images. This allows us to directly evaluate whether the applied transformations—such as resizing and cropping—have successfully standardized the input to 64x64 resolution while preserving essential facial features like the eyes, nose, and mouth. By inspecting these visualizations, we can also verify that the generated masks for object removal are accurately positioned, effectively covering key regions of the cat's face necessary for the imprinting task.

```
# Grab pictures of just cats, some other pictures have their owners in the backgr
image0, image1, image2 = transformed_train_dataset[1], transformed_train_dataset[


# Display training images and their ground truth images
f, ax = plt.subplots(3, 2, figsize=(12, 12))
ax[0, 0].imshow(np.transpose(image0[0], (1, 2, 0)))
ax[0, 1].imshow(np.transpose(image0[1], (1, 2, 0)))
ax[1, 0].imshow(np.transpose(image1[0], (1, 2, 0)))
ax[1, 1].imshow(np.transpose(image1[1], (1, 2, 0)))
ax[2, 0].imshow(np.transpose(image2[0], (1, 2, 0)))
ax[2, 1].imshow(np.transpose(image2[1], (1, 2, 0)))

ax[0, 0].set_title("Training Images");
ax[0, 1].set_title("Ground Truth Images");
```



3.5 Setup Dataloader

We set up a data loader to efficiently feed batches of images into our model during training, validation, and testing. The `DataLoader` handles the loading and batching of data in a way that optimizes memory usage and speeds up training, especially when working with large datasets. By setting a `BATCH_SIZE` of 8, we strike a balance between computational efficiency and gradient stability—small enough to fit into memory without overwhelming the GPU, but large enough to provide stable gradient estimates during backpropagation. Shuffling the data ensures that each batch contains a diverse set of examples, which helps prevent the model from learning spurious patterns tied to the order of the dataset.

```
train_dataset, val_dataset, test_dataset = transformed_train_dataset, transformed
```

```
BATCH_SIZE = 8
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=True)
```

4. GAN Model

The special type of GANs we plan to use is an DCGANs, which is a variation of the deep learning models composed of two convolutional neural networks — a generator and a discriminator — trained in opposition. In image processing tasks like inpainting, super-resolution, or style transfer, DCGANs learn to generate realistic images by mimicking the distribution of real images.

The generator aims to produce realistic images that can fool the discriminator. For image processing tasks like inpainting, the generator is conditioned to a masked input image with the center noise.

In the end, the goal is to minimize the loss of our model, which from lec12, we know to be:

$$\min_{w} \max_{\theta} \left(- \sum_{i \in Real} \ln \sigma_w(I_i) - \sum_{i \in Fake} \ln(1 - \sigma_w(I_\theta(z_i))) \right)$$

We experimented various architectures, referencing various academic papers for inspiration. We took inspiration from S. Iizuka, E. Simo-Serra, H. Ishikawa, modifying the architecture to fit our needs. One notable difference we chose to not take from the paper was the use of 2 discriminators, a global one and local one. To simplify our training we instead had one regular discriminator that functions as a simple CNN to classify between authentic and generated images. However, as next steps, we can definitely try experimenting with two discriminators.

The generator uses an encoder-decoder structure with convolutional layers to extract spatial features and deconvolutional layers to reconstruct the full image. Dilated convolutions expand the receptive field, helping the model capture broader context for coherent inpainting. Batch normalization stabilizes training and improves convergence. The generator's input is also 4-channels instead of 3 where the 4th channel is the mask. We included the mask to give the model more contextual information, with the goal of enhancing the learning process. The discriminator, designed as a patch-based classifier, uses convolution and down-sampling to evaluate image realism, focusing on structural and textural consistency. This adversarial setup pushes the generator to create realistic, context-aware content. Together, these components enable effective learning of both local details and global patterns for high-quality image inpainting.

4.1 Generator

```
class Generator(nn.Module):
    """
        This is the generator class. The generator takes in a masked image and it
        missing part of the masked image.

        The architecture of the generator is inspired by the generator in the art
        earlier:
            1) 4x64x64    -> 64x64x64
            2) 64x64x64   -> 128x32x32
            3) 128x32x32  -> 256x16x16
            4) 256x16x16  -> 256x16x16
            5) 256x16x16  -> 256x14x14
            6) 256x14x14  -> 256x8x8
            7) 256x8x8    -> 256x8x8
            8) 256x8x8    -> 128x16x16
            9) 128x16x16  -> 128x16x16
            10) 128x16x16  -> 64x32x32
            11) 64x32x32   -> 64x64x64
            12) 64x64x64   -> 64x64x64
            13) 64x64x64   -> 3x64x64
    """

    def __init__(self, im_channels):
        super().__init__()

        # Encoder
        self.conv1 = nn.Conv2d(im_channels + 1, 64, 5, stride=1, padding=2)
        self.conv2 = nn.Conv2d(64, 128, 3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(128, 256, 3, stride=2, padding=1)
        self.conv4 = nn.Conv2d(256, 256, 3, stride=1, padding=1)

        # Dilation
        self.conv5 = nn.Conv2d(256, 256, 3, stride=1, padding=1, dilation=2)
        self.conv6 = nn.Conv2d(256, 256, 3, stride=1, padding=1, dilation=4)
        self.conv7 = nn.Conv2d(256, 256, 3, stride=1, padding=1)

        # Decoder
        self.deconv8 = nn.ConvTranspose2d(256, 128, 4, stride=2, padding=1)
        self.conv9 = nn.Conv2d(128, 128, 3, stride=1, padding=1)

        self.deconv10 = nn.ConvTranspose2d(128, 64, 4, stride=2, padding=1)
        self.deconv11 = nn.ConvTranspose2d(64, 64, 4, stride=2, padding=1)
        self.conv12 = nn.Conv2d(64, 64, 3, stride=1, padding=1)
        self.conv13 = nn.Conv2d(64, 3, 3, stride=1, padding=1)

        # Batch norms for each layer
        self.bn1 = nn.BatchNorm2d(64)
        self.bn2 = nn.BatchNorm2d(128)
```

```
        self.bn3 = nn.BatchNorm2d(256)
        self.bn4 = nn.BatchNorm2d(256)
        self.bn5 = nn.BatchNorm2d(256)
        self.bn6 = nn.BatchNorm2d(256)
        self.bn7 = nn.BatchNorm2d(256)
        self.bn8 = nn.BatchNorm2d(128)
        self.bn9 = nn.BatchNorm2d(128)
        self.bn10 = nn.BatchNorm2d(64)
        self.bn11 = nn.BatchNorm2d(64)
        self.bn12 = nn.BatchNorm2d(64)

        # Activation Functions
        self.relu = torch.nn.ReLU()
        self.sig = nn.Sigmoid()

    def forward(self, x):
        # Encoder layers
        x1 = self.bn1(self.relu(self.conv1(x)))
        x2 = self.bn2(self.relu(self.conv2(x1)))
        x3 = self.bn3(self.relu(self.conv3(x2)))
        x4 = self.bn4(self.relu(self.conv4(x3)))

        # Dilation Layers
        x5 = self.bn5(self.relu(self.conv5(x4)))
        x6 = self.bn6(self.relu(self.conv6(x5)))
        x7 = self.bn7(self.relu(self.conv7(x6)))

        # Decoder layers
        x8 = self.bn8(self.relu(self.deconv8(x7)))
        x9 = self.bn9(self.relu(self.conv9(x8)))
        x10 = self.bn10(self.relu(self.deconv10(x9)))
        x11 = self.bn11(self.relu(self.deconv11(x10)))
        x12 = self.bn12(self.relu(self.conv12(x11)))

        output = self.sig(self.conv13(x12))
        return output
```

4.2 Discriminator

Our discriminator is implemented as a straightforward CNN-based binary classifier that takes in a full image and predicts whether it is real or generated. The architecture progresses through a series of convolutional layers with increasing depth, downsampling the input from 3×64×64 to a 512-dimensional feature map. These are followed by fully connected layers that gradually reduce dimensionality, ending in a single sigmoid output. Batch normalization is applied after each convolutional layer to stabilize learning, and ReLU activations are used throughout to introduce non-linearity. We also include global average pooling to compress spatial information before classification, ensuring that the discriminator captures high-level texture and structural patterns across the entire image.

In contrast to the dual-discriminator approach of the original paper, which allowed fine-grained evaluation of local patches within masked regions, our discriminator assesses the realism of the entire image at once. While this may reduce sensitivity to small artifacts within the inpainted region, it simplifies the architecture and training dynamics, making it easier to integrate and debug. As a next step, we plan to explore reintroducing a local discriminator to improve the model's ability to detect inconsistencies in smaller, high-detail areas.

```
class Discriminator(nn.Module):
    """
        This is the discriminator class. The discriminator takes in an image and
        or real (non-generated).

        The architecture of the network is the same as a simple CNN classifier
        with the following architecture:
            1) 3x64x64   -> 64x32x32
            2) 64x32x32  -> 128x16x16
            3) 128x16X16 -> 256x8x8
            4) 256x8x8   -> 512x4x4
            5) 512x4x4   -> 512x1x1
            6) 512 -> 512
            7) 512 -> 512
            8) 512 -> 256
            9) 256 -> 10
            10) 10 -> 1
    """
    def __init__(self):
        super().__init__()

        # Convolutional layers
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1)
```

```
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1)

        # Batch norm layers
        self.bn1 = nn.BatchNorm2d(64)
        self.bn2 = nn.BatchNorm2d(128)
        self.bn3 = nn.BatchNorm2d(256)
        self.bn4 = nn.BatchNorm2d(512)

        # Linear layers
        self.fc1 = nn.Linear(512, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, 256)
        self.fc4 = nn.Linear(256, 10)
        self.fc5 = nn.Linear(10, 1)

        # Activation functions and pooling
        self.down_sample = nn.AvgPool2d(2)
        self.global_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # Forwad pass through the layers
        x = self.down_sample(self.conv1(x))
        x = self.relu(x)
        x = self.bn1(x)

        x = self.down_sample(self.conv2(x))
        x = self.relu(x)
        x = self.bn2(x)

        x = self.down_sample(self.conv3(x))
        x = self.relu(x)
        x = self.bn3(x)

        x = self.down_sample(self.conv4(x))
        x = self.relu(x)
        x = self.bn4(x)

        x = self.global_pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
```

```
        x = self.relu(x)
        x = self.fc4(x)
        x = self.relu(x)
        x = self.fc5(x)
        x = self.sigmoid(x)

        return x
```

4.3 Training

The training loop involves alternating updates between the generator (g) and the discriminator (d) over multiple epochs using a training dataset of masked images and corresponding ground truth images. Each batch includes the masked image (input), the real image (target), and a generated mask indicating the region to inpaint. The discriminator is first trained to distinguish real images from generated (inpainted) ones using binary cross-entropy loss, aiming to assign high scores to real images and low scores to fake ones. After updating the discriminator, the generator is trained to produce inpainted images that can "fool" the discriminator into classifying them as real. The generator also receives the mask as a fourth input channel to guide the inpainting process. Both networks are optimized using their respective optimizers.

Optionally, during training, the model can display validation images to visually track progress. This includes the masked input, the generator's raw output, the inpainted composite, and the ground truth. Losses for both networks are recorded for analysis, and if verbose is enabled, progress is printed after each epoch.

```
def train(g, d, train_loader, val_loader, criterion, optimizer_g, optimizer_d, ep
    """
    Trains model for image inpainting.

    Args:
        g: The generator model.
        d: The discriminator model.
        train_loader: DataLoader for the training dataset.
        val_loader: DataLoader for the validation dataset.
        criterion: The loss function.
        optimizer_g: Optimizer for the generator.
        optimizer_d: Optimizer for the discriminator.
        epochs: The number of training epochs.
        verbose: If True, prints training progress.
        show_images: If True, displays generated images during validation.
    """
    # Total losses for epoch x batches
```

```
g_losses = []
d_losses = []

total_steps = len(train_loader)
val_iter = iter(val_loader)

for epoch in range(epochs):
    g.train()
    d.train()

    # Average loss per batch
    g_avg_loss_per_batch = 0
    d_avg_loss_per_batch = 0

    for i, batch in enumerate(train_loader):
        im_masked, im_truth = batch

        # Move data to GPU
        im_masked = im_masked.to(device)
        im_truth = im_truth.to(device)

        # Add the masks as a separate channel
        N = batch[0].shape[0]
        masks = create_masks(N).to(device)
        im_masked = torch.cat((im_masked, masks), dim=1)

        # --- Train Discriminator ---
        d.zero_grad()

        # Train discriminator with ground truth images
        outputs = d(im_truth).view(-1)
        true_labels = torch.ones(N).to(device)
        d_err = criterion(outputs, true_labels)

        # Train discriminator with fake images
        fake_im = g(im_masked)
        inpainted = fake_im * masks[0] + im_masked[:, :3, :, :]
        output = d(inpainted.detach()).view(-1)
        fake_labels = torch.zeros(N).to(device)
        d_err += criterion(output, fake_labels)

        d_err.backward()
        optimizer_d.step()

        # --- Train Generator ---
        g.zero_grad()

        fake_im = g(im_masked)
```

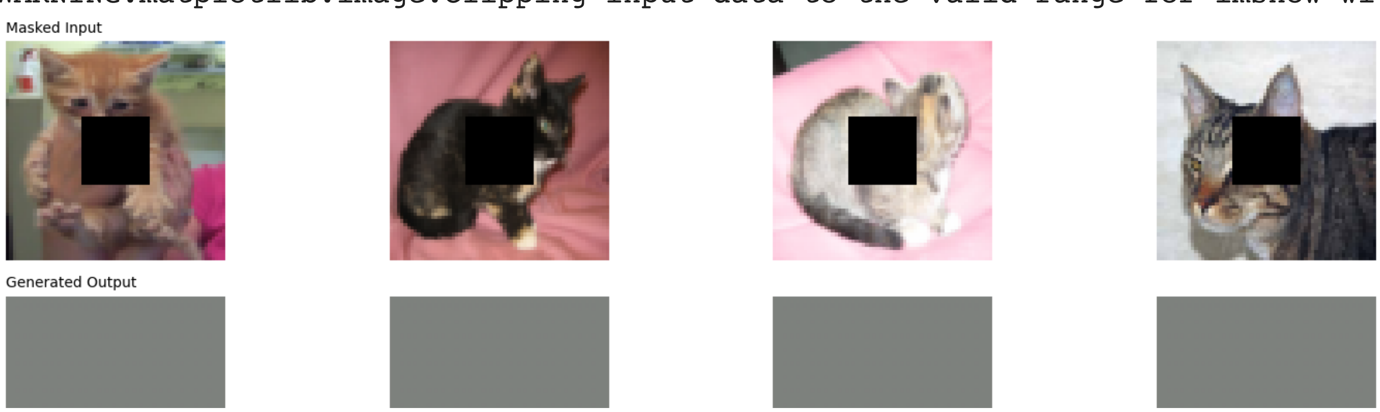
```
            inpainted = fake_im * masks[0] + im_masked[:, :3, :, :]
            outputs = d(inpainted).view(-1)

            g_err = criterion(outputs, true_labels)
            g_err.backward()
            optimizer_g.step()

            # Keep track of loss
            g_avg_loss_per_batch += g_err.item()
            d_avg_loss_per_batch += d_err.item()

            g_losses.append(g_err.item())
            d_losses.append(d_err.item())

            if show_images and i == 0:
                g.eval()
                with torch.no_grad():
                    val_batch = next(val_iter)
                    val_masked, val_truth = val_batch

                    # Move validation data to GPU
                    val_masked = val_masked.to(device)
                    val_truth = val_truth.to(device)

                    N = val_masked.shape[0]
                    val_masks = create_masks(N).to(device)
                    val_input = torch.cat((val_masked, val_masks), dim=1)

                    # Inpainting
                    fake_val = g(val_input)
                    inpainted = val_masked * (1 - val_masks) + fake_val * val_mas

                    # Unpainted
                    unpainted = val_truth * (1 - val_masks)

                    # Get first 4 samples
                    idx = slice(0, 4)
                    sample_masked = val_masked[idx]
                    sample_fake = fake_val[idx]
                    sample_inpainted = inpainted[idx]
                    sample_truth = val_truth[idx]

                    images = [sample_masked, sample_fake, sample_inpainted, sampl
                    labels = ["Masked Input", "Generated Output", "Inpainted", "G

                    # Plot
                    fig, axs = plt.subplots(len(images), 4, figsize=(16, 10))
                    for row in range(len(images)):
```

```
                        for col in range(4):
                            img = images[row][col].cpu()
                            img = img.permute(1, 2, 0)  # CxHxW → HxWxC
                            axs[row, col].imshow(img.numpy(), cmap='gray' if img.
                            axs[row, col].axis('off')
                            if col == 0:
                                axs[row, col].set_title(labels[row], fontsize=10,
                    plt.tight_layout()
                    plt.show()

                g.train()

        if verbose:
            print(f'Epoch [{epoch+1}/{epochs}], Step [{i+1}/{total_steps}], '
                    f'Discriminator Loss: {(d_avg_loss_per_batch/total_steps):.4f
                    f'Generator Loss: {(g_avg_loss_per_batch/total_steps):.4f}')

    return g_losses, d_losses
```

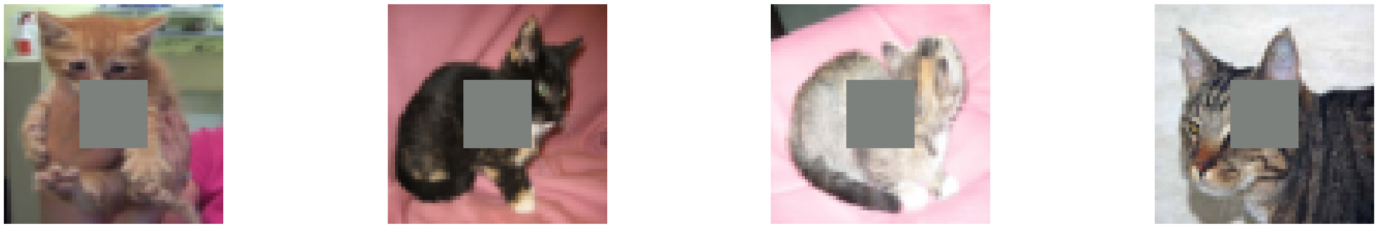
We first do a basic training of our model. We will observe how the generator fills in the masked hole after each epoch and compare it to the ground truth.

```
g = Generator(im_channels=3).to(device)
d = Discriminator().to(device)

optimizer_g = optim.Adam(g.parameters(), lr=0.0001)
optimizer_d = optim.Adam(d.parameters(), lr=0.0001)

criterion = nn.BCELoss()
num_epochs = 50

g_losses, d_losses = train(g, d, train_loader, val_loader, criterion, optimizer_g
```

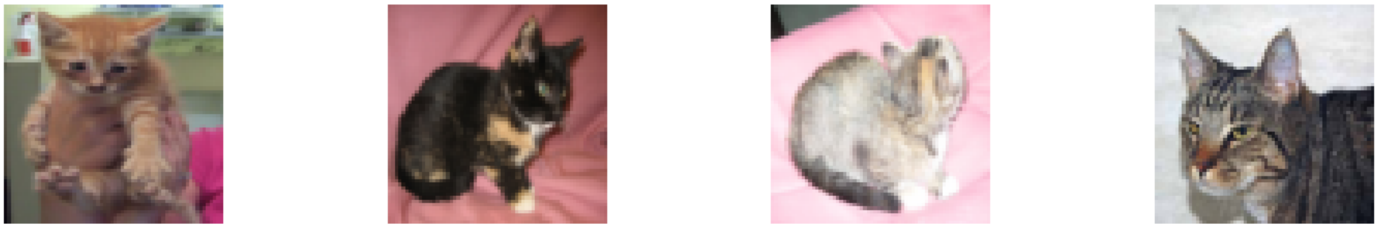
```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
```



Inpainted

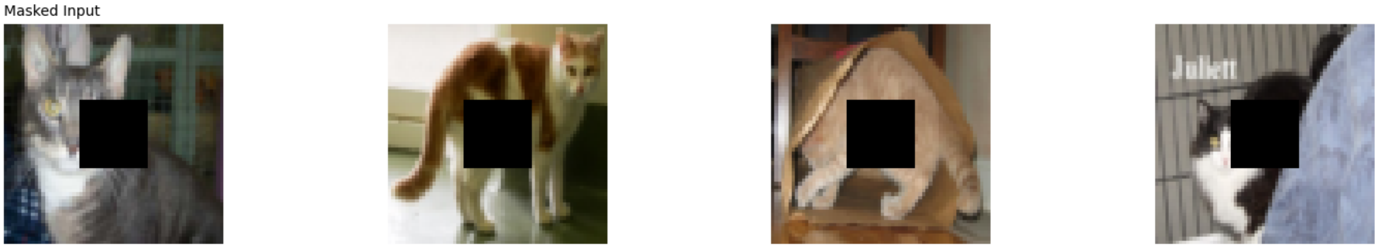


Ground Truth

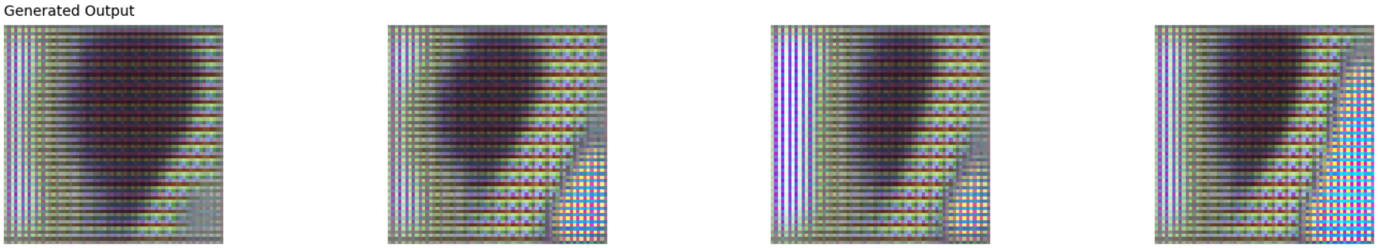


Epoch [1/50], Step [1174/1174], Discriminator Loss: 1.3833, Generator Loss: 0.

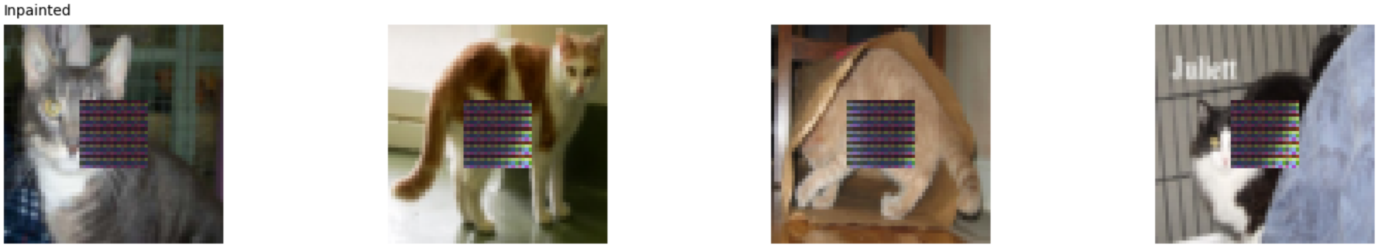
Masked Input



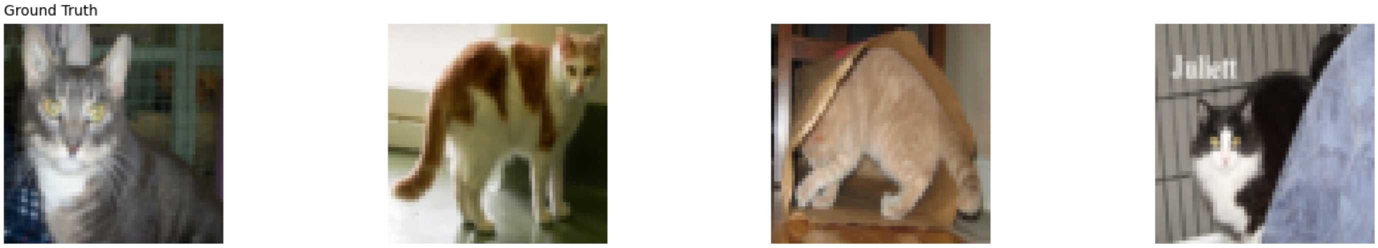
Generated Output



Inpainted

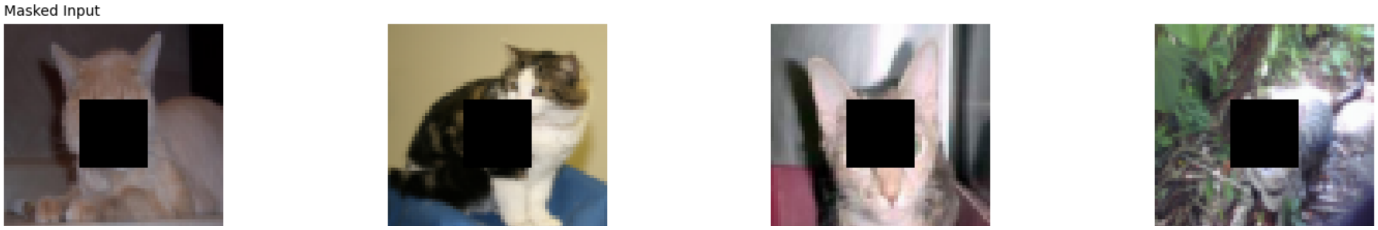


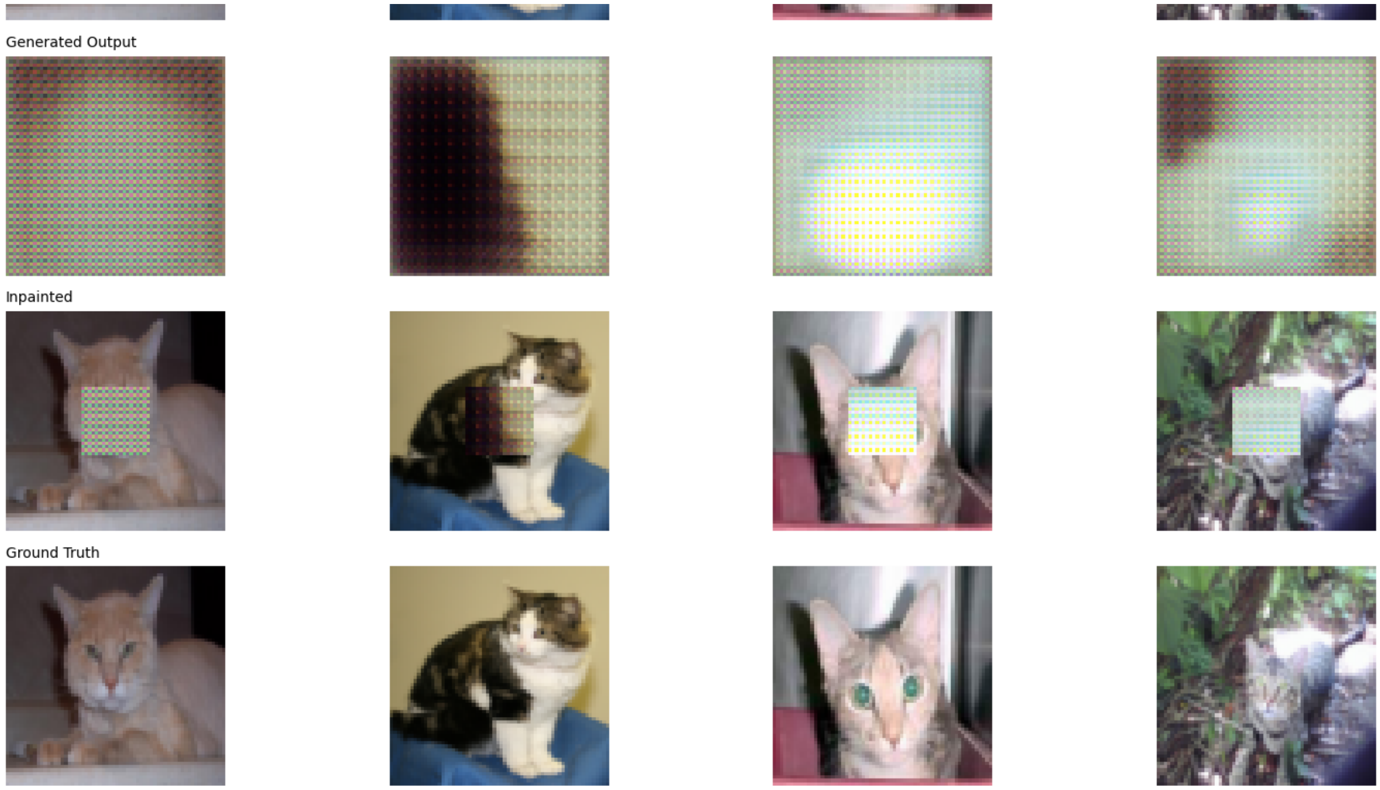
Ground Truth



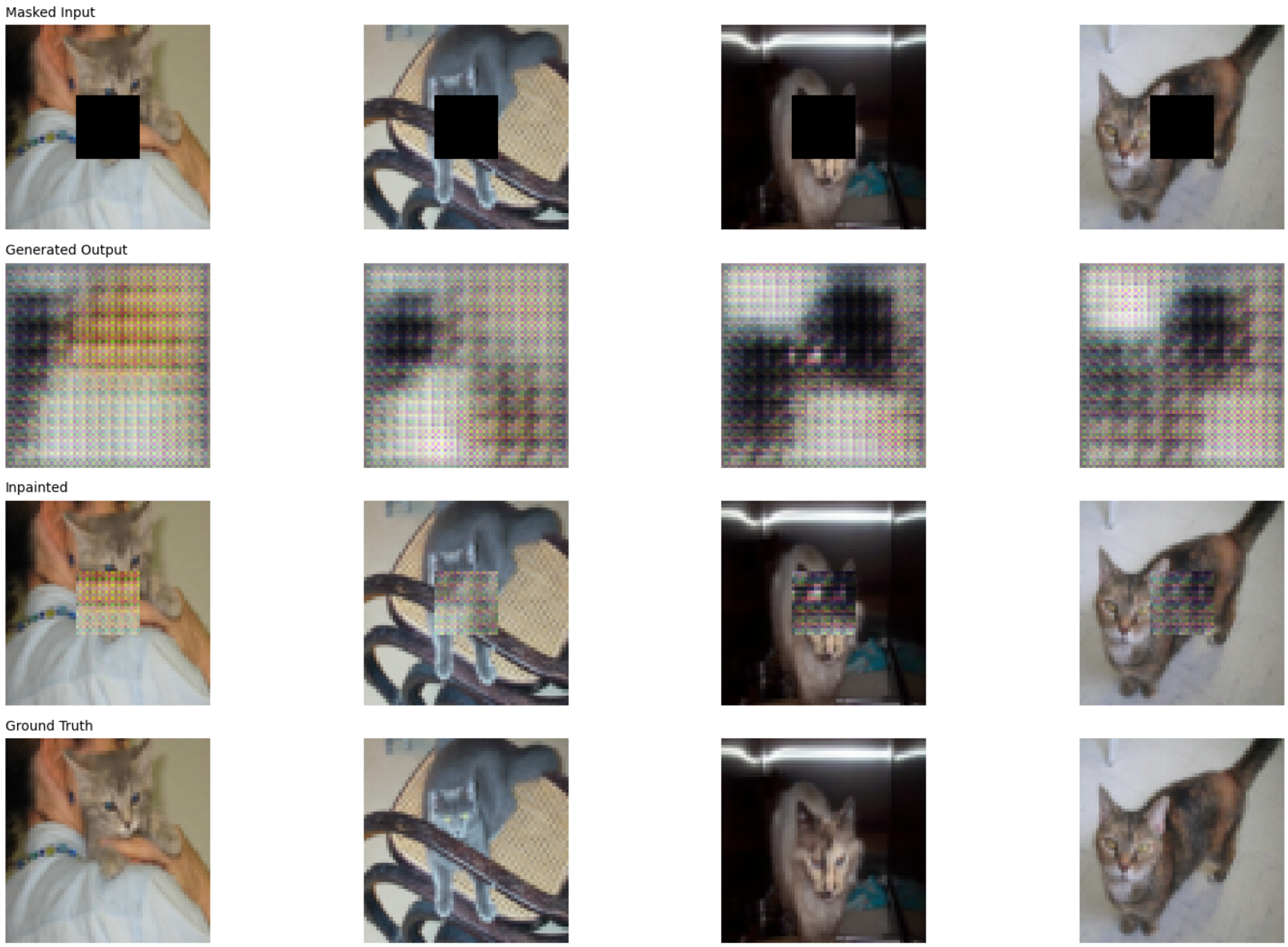
Epoch [2/50], Step [1174/1174], Discriminator Loss: 1.3364, Generator Loss: 0.

Masked Input



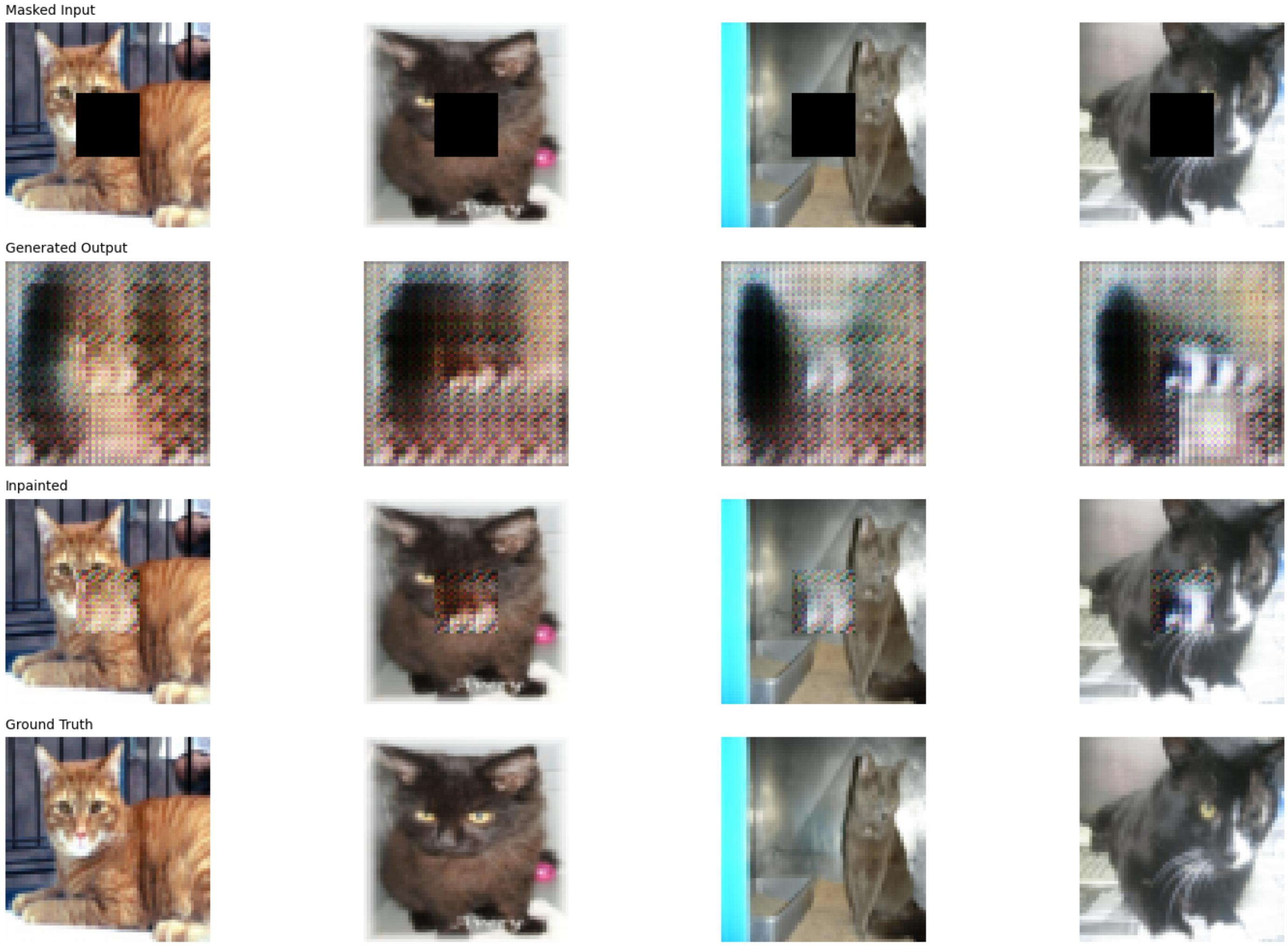



```
Epoch [3/50], Step [1174/1174], Discriminator Loss: 1.3101, Generator Loss: 0.
```

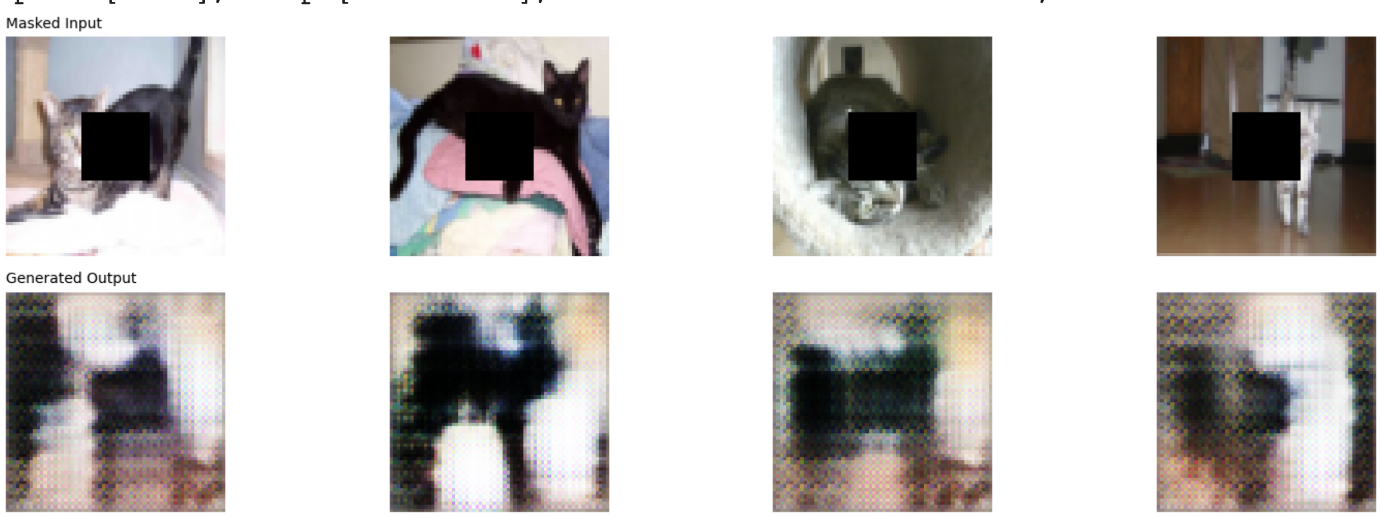


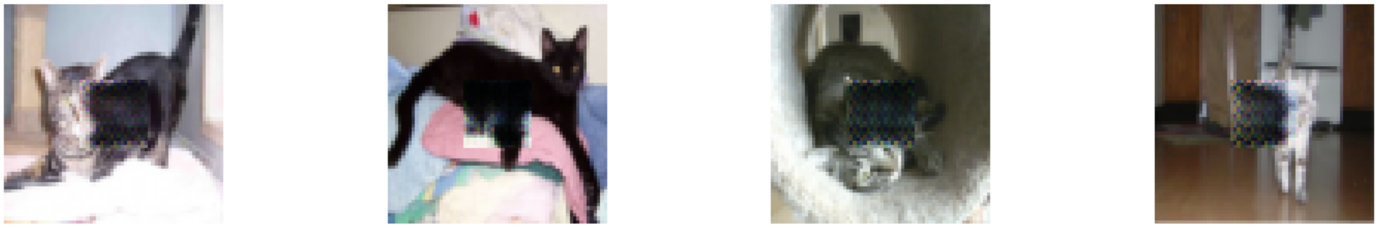

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [4/50], Step [1174/1174], Discriminator Loss: 1.1924, Generator Loss: 1.
```

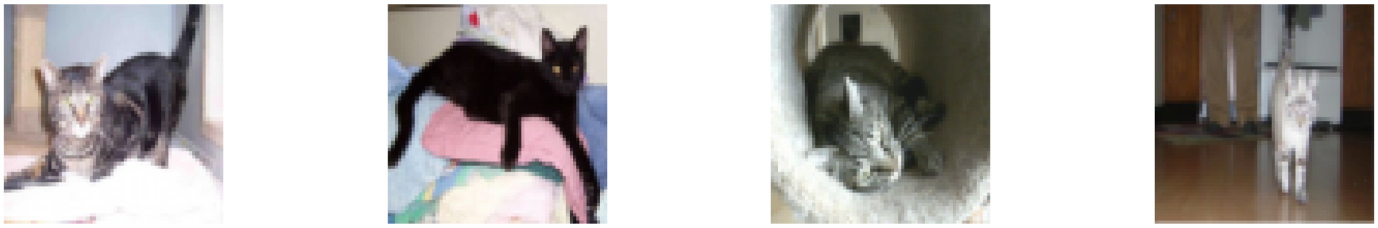



```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [5/50], Step [1174/1174], Discriminator Loss: 0.9779, Generator Loss: 1.
```



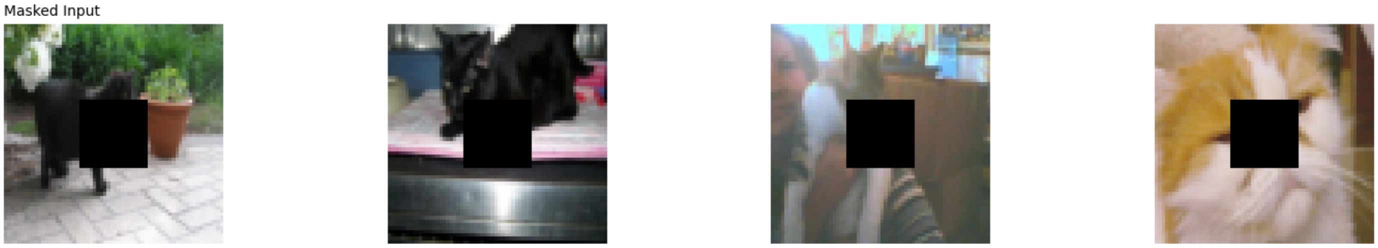


Ground Truth

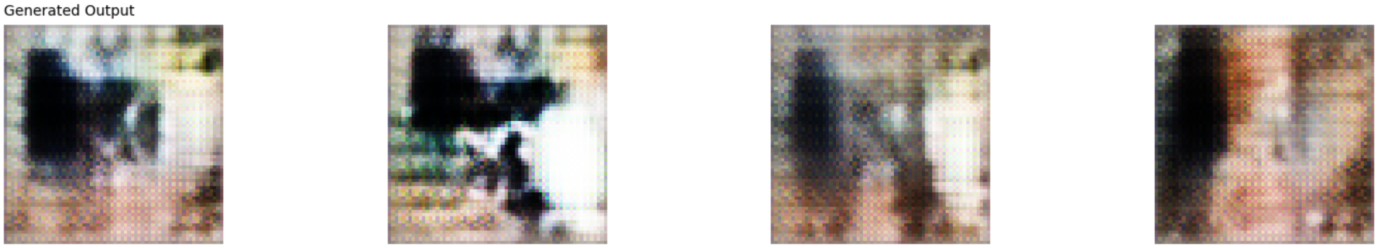


Epoch [6/50], Step [1174/1174], Discriminator Loss: 0.7770, Generator Loss: 2.

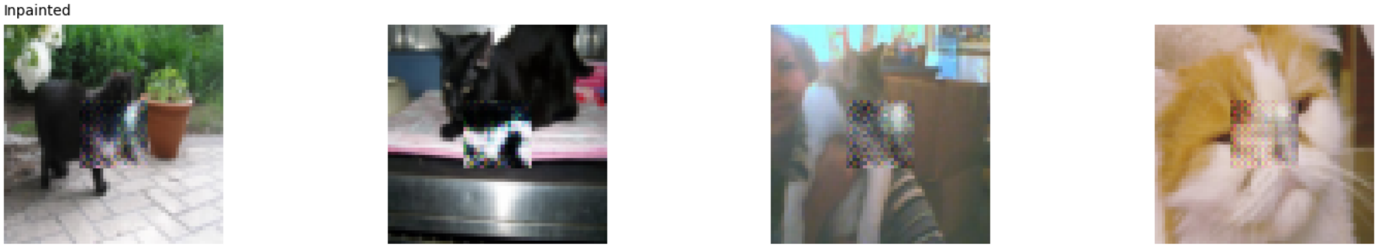
Masked Input



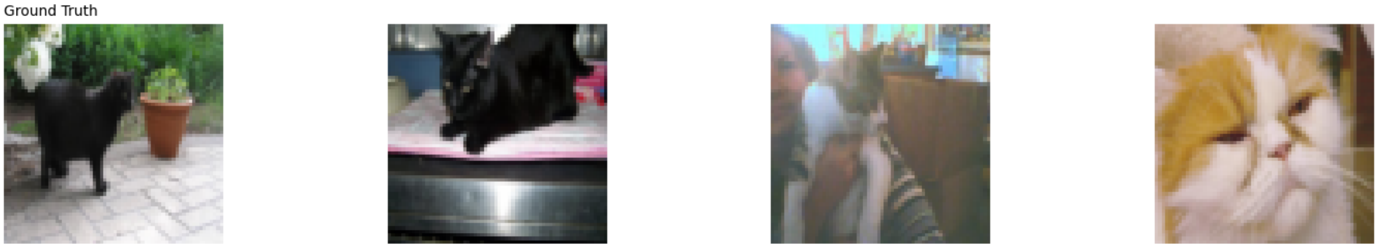
Generated Output



Inpainted

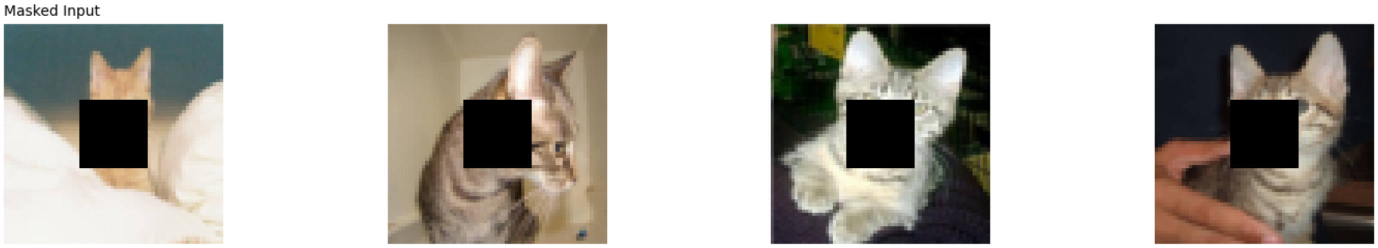


Ground Truth



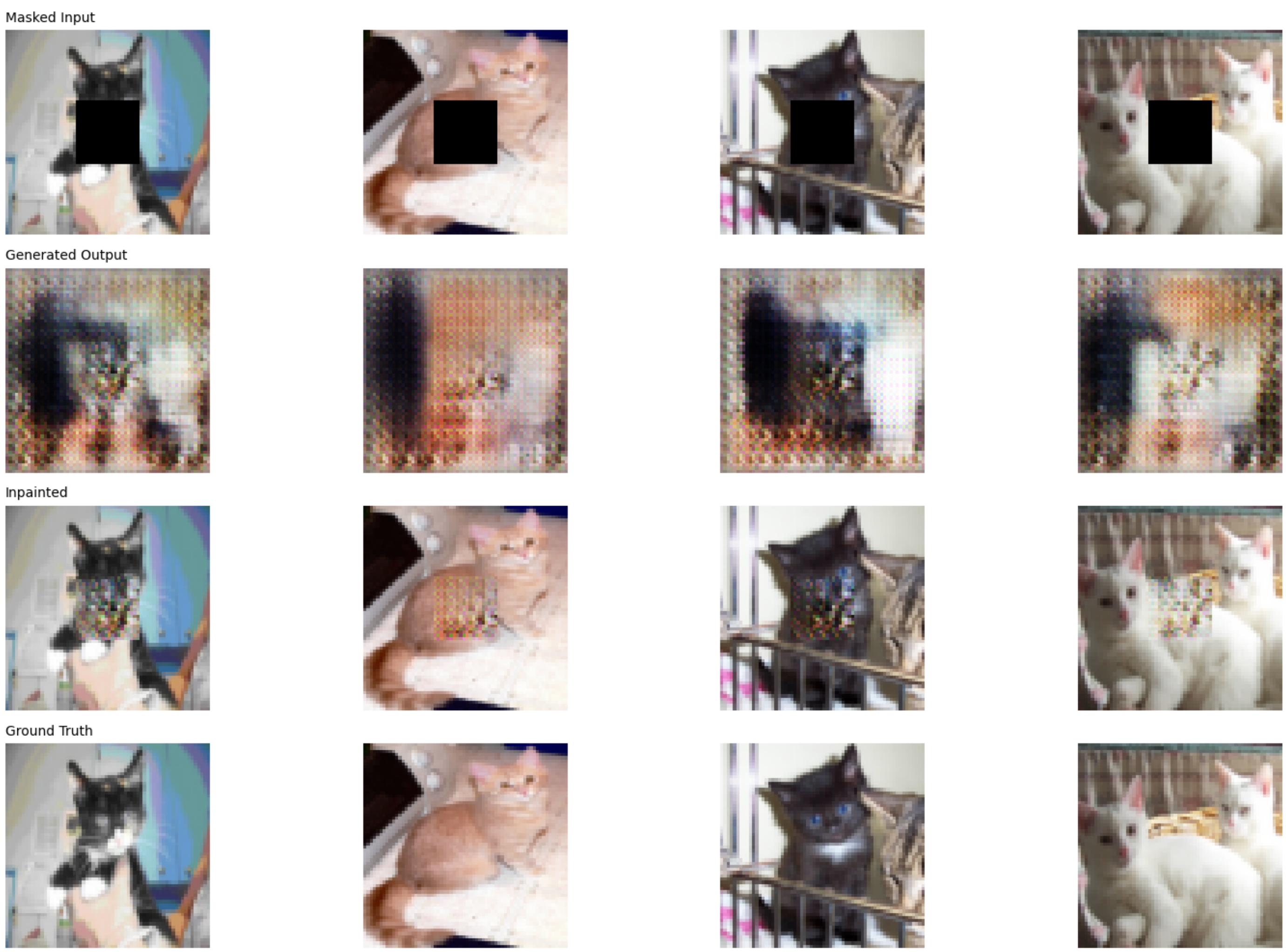
Epoch [7/50], Step [1174/1174], Discriminator Loss: 0.7337, Generator Loss: 2.

Masked Input

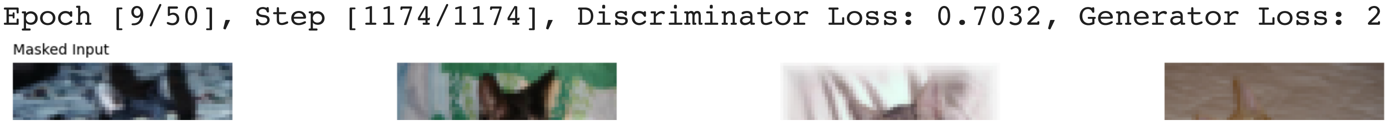


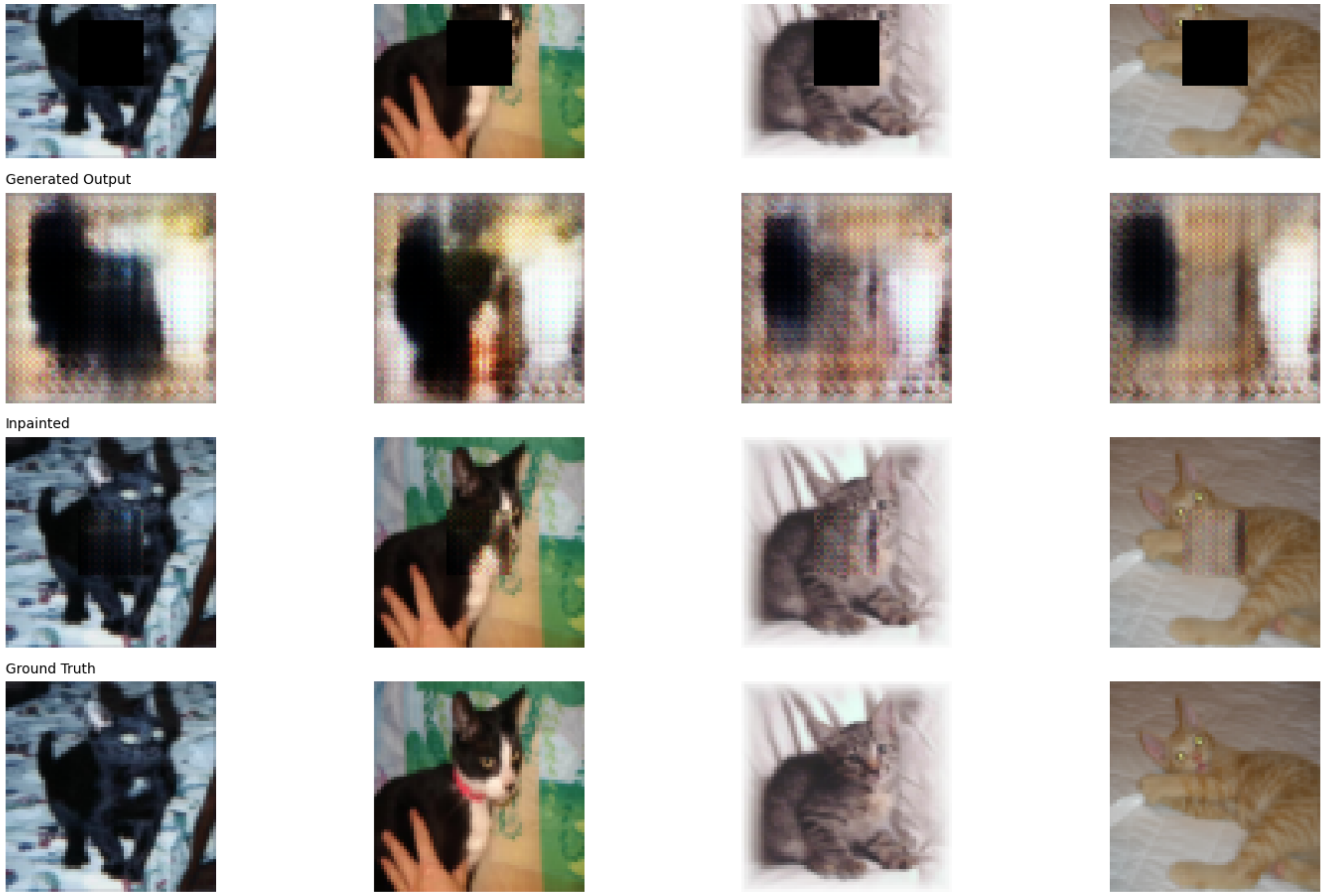
Generated Output


```
Epoch [8/50], Step [1174/1174], Discriminator Loss: 0.6841, Generator Loss: 2.
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
```




```
Epoch [9/50], Step [1174/1174], Discriminator Loss: 0.7032, Generator Loss: 2.
```



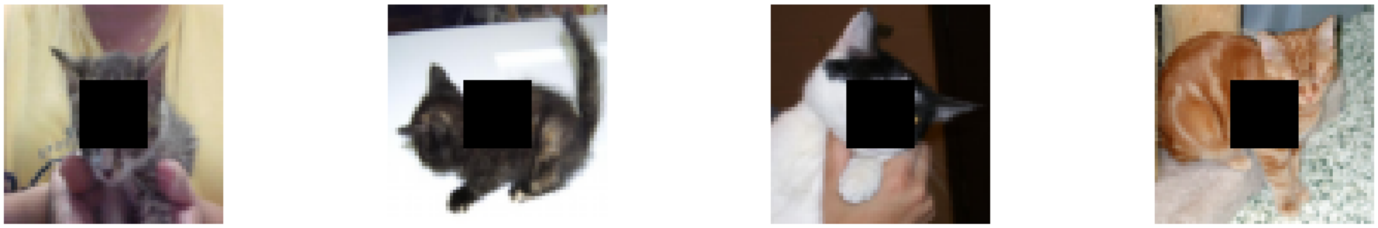



```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [10/50], Step [1174/1174], Discriminator Loss: 0.6772, Generator Loss: 2
```

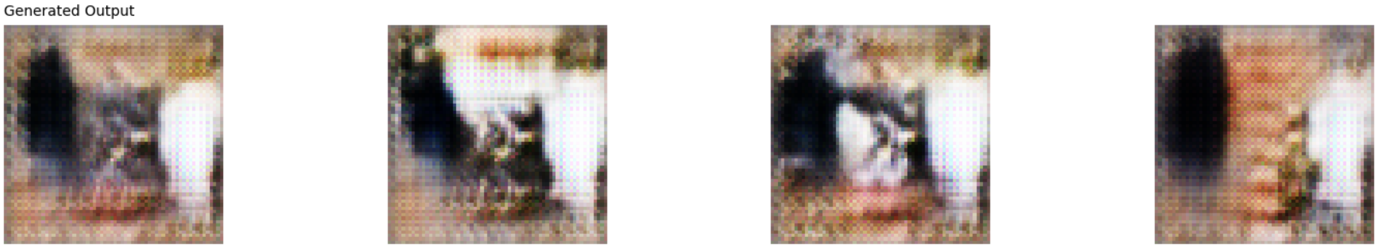
Ground Truth

WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [11/50], Step [1174/1174], Discriminator Loss: 0.7572, Generator Loss: 2

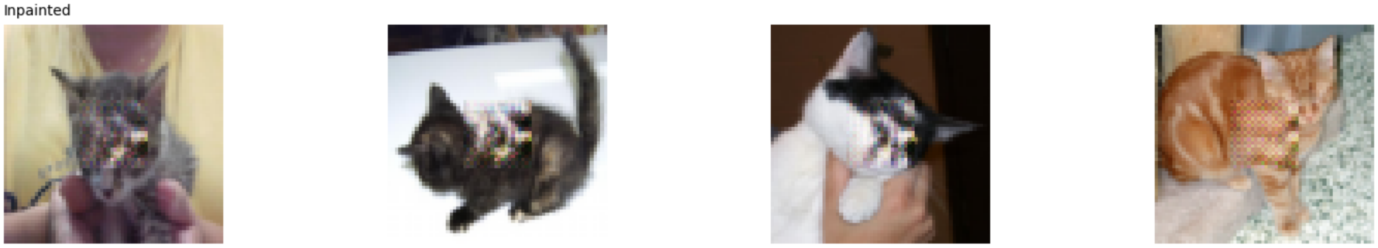
Masked Input



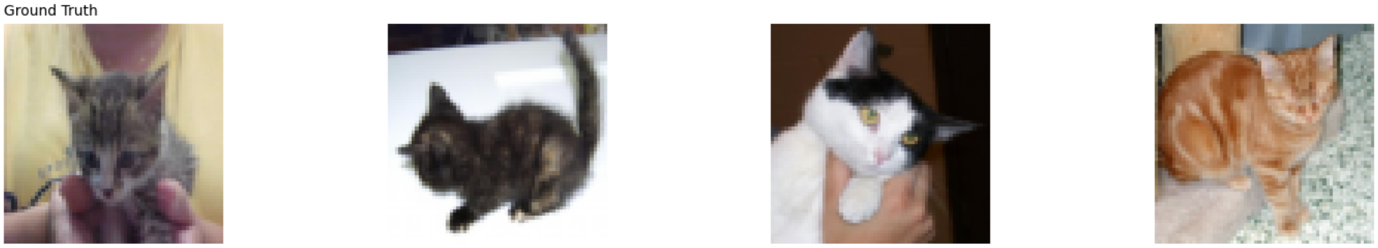
Generated Output



Inpainted



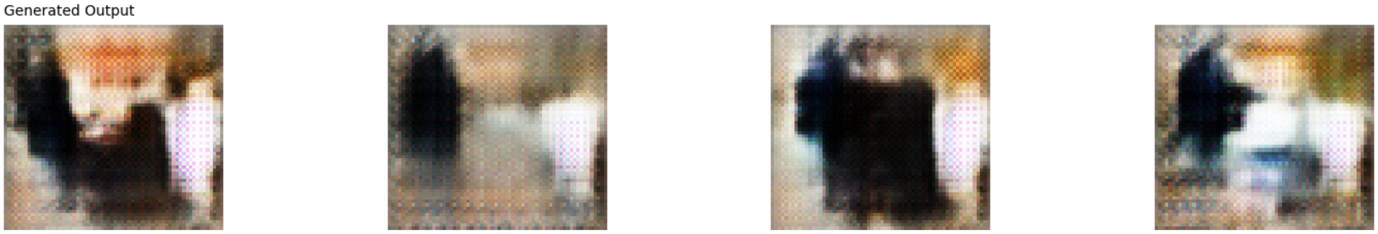
Ground Truth



Epoch [12/50], Step [1174/1174], Discriminator Loss: 0.6982, Generator Loss: 2

Masked Input

Generated Output

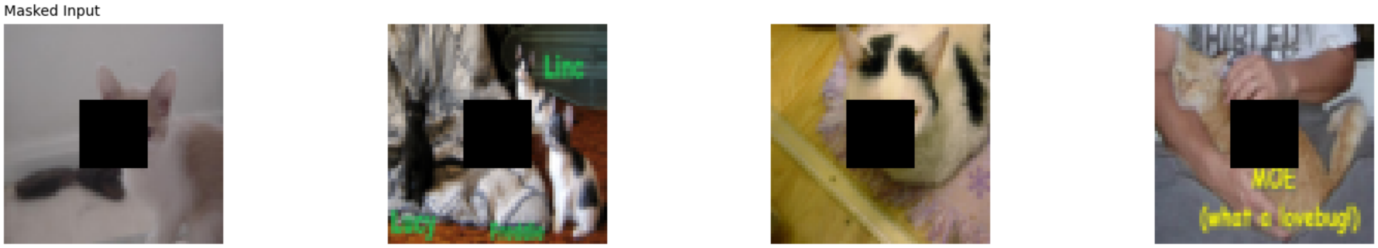


Inpainted

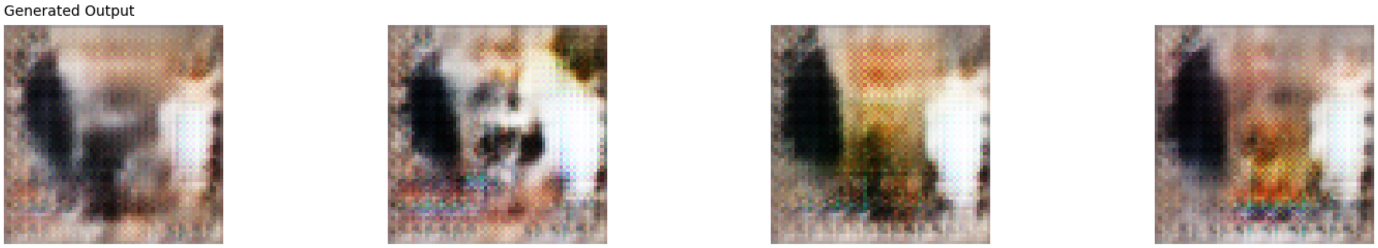
Ground Truth

Epoch [13/50], Step [1174/1174], Discriminator Loss: 0.6908, Generator Loss: 2

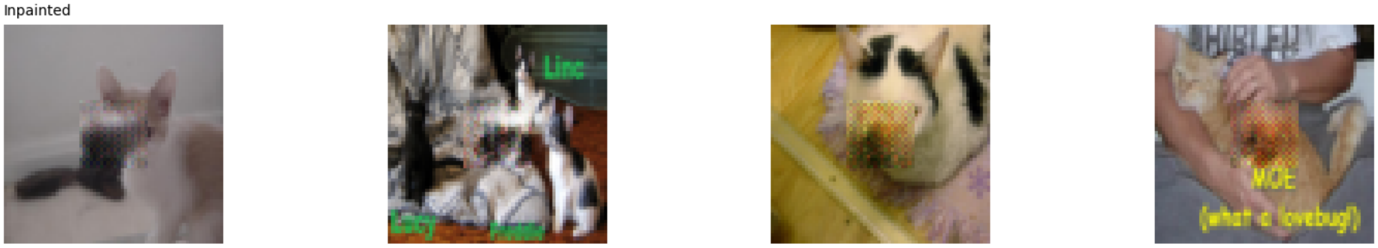
Masked Input



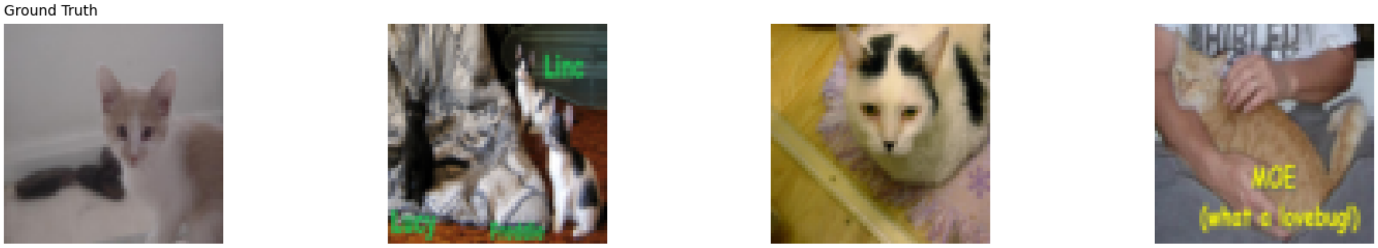
Generated Output



Inpainted

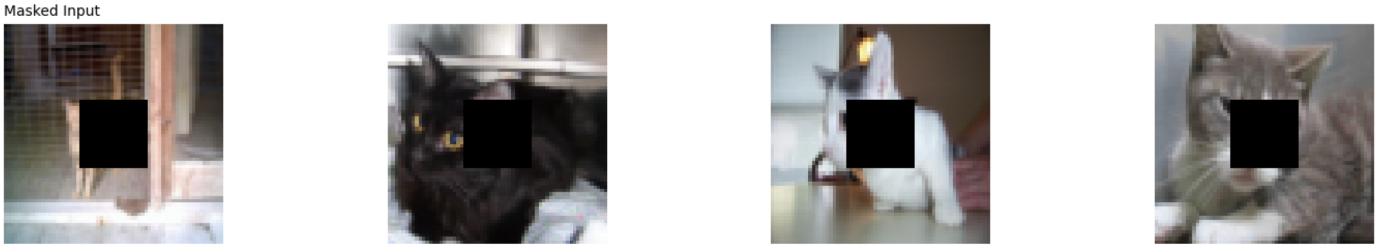


Ground Truth

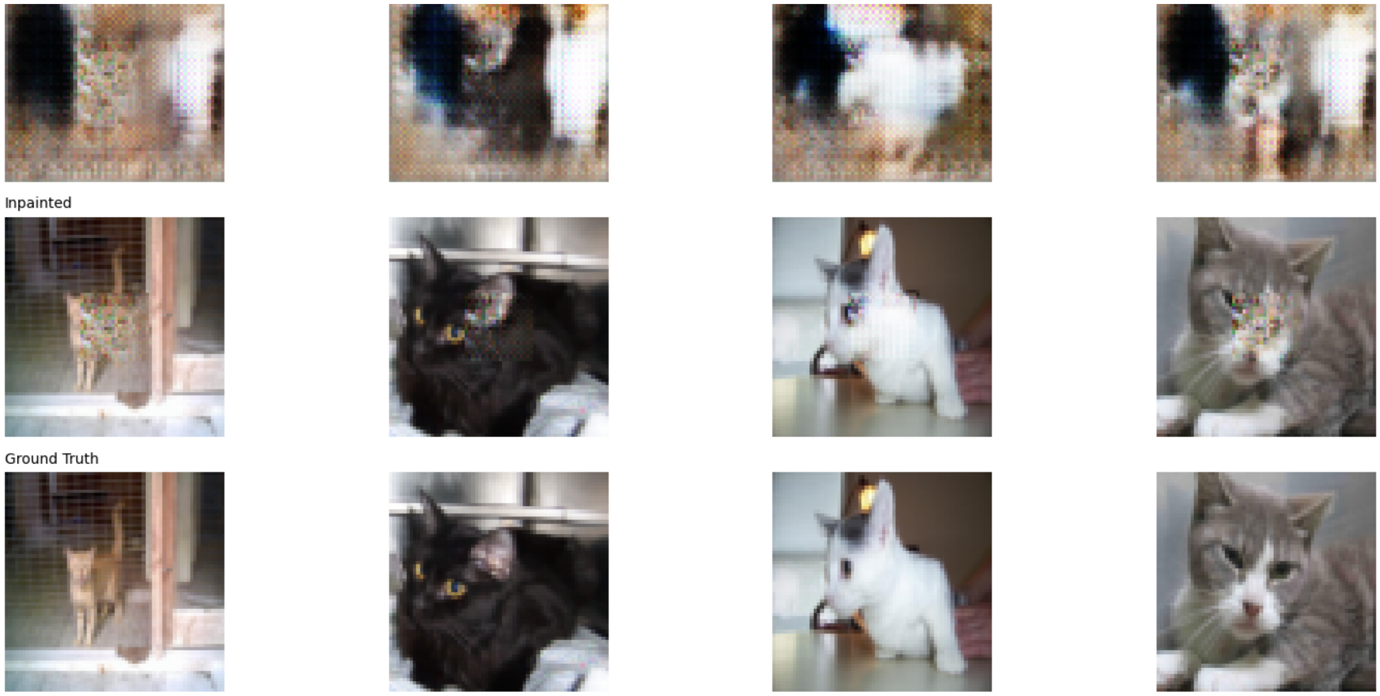


Epoch [14/50], Step [1174/1174], Discriminator Loss: 0.6046, Generator Loss: 2

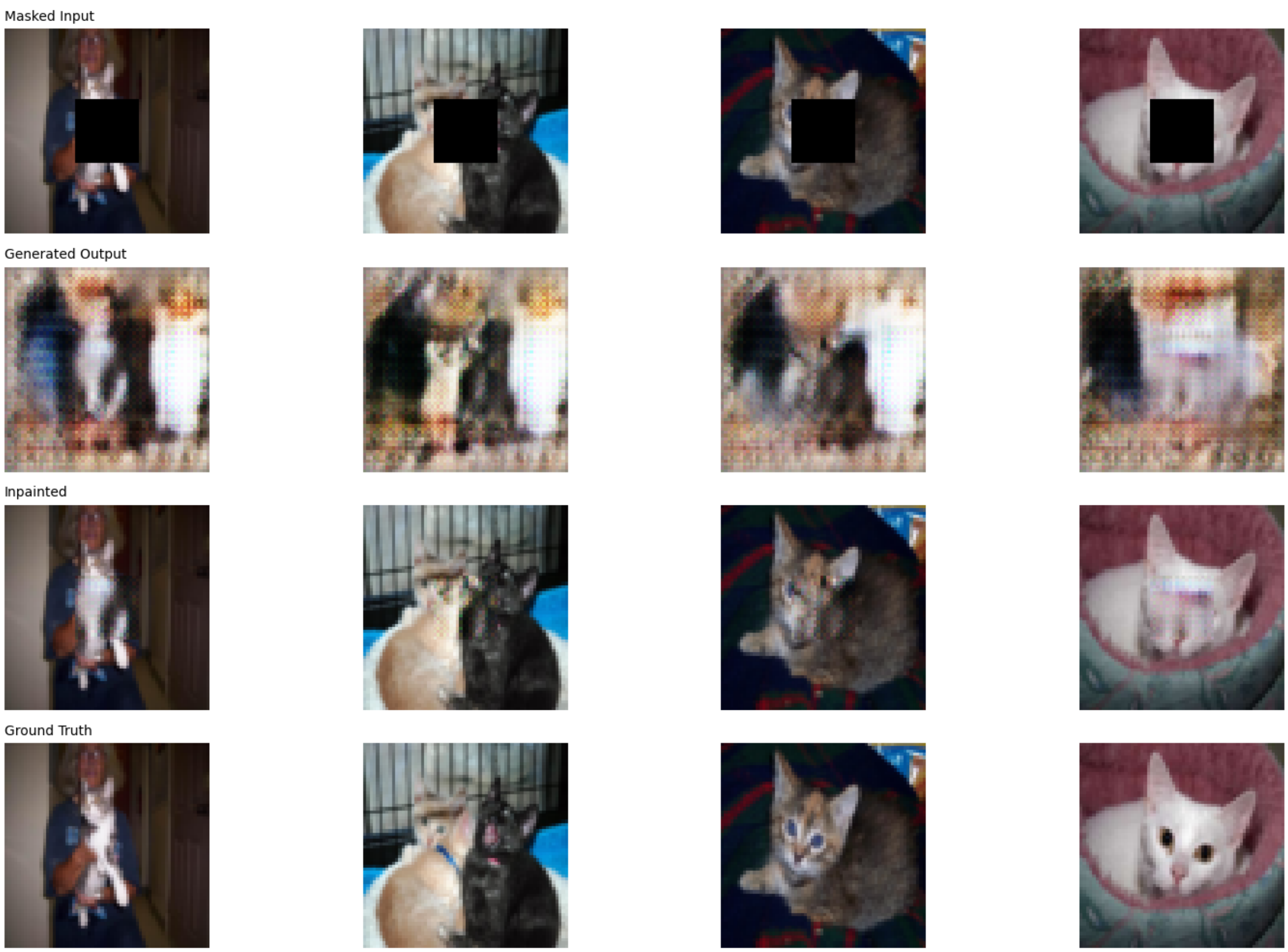
Masked Input



Generated Output

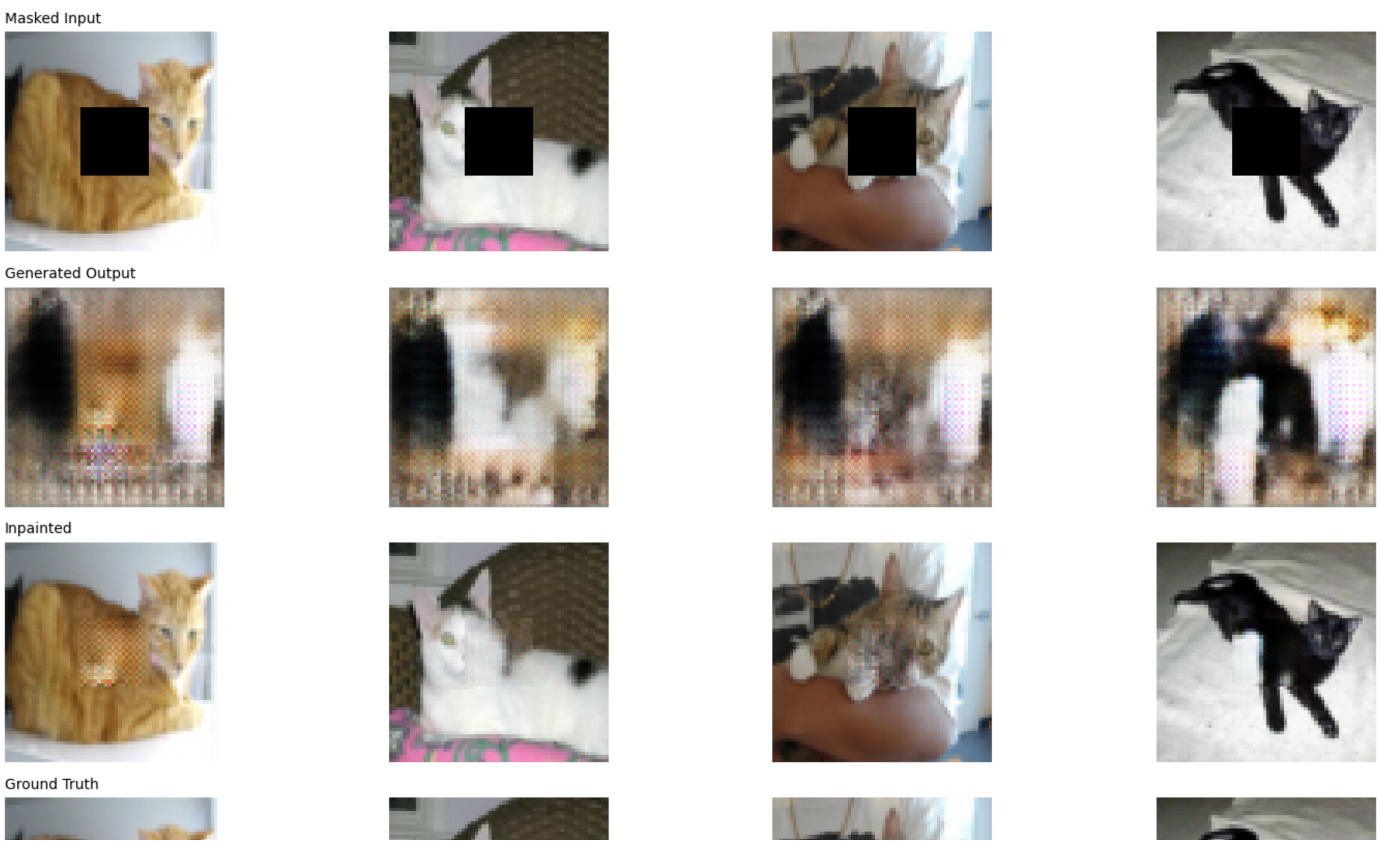


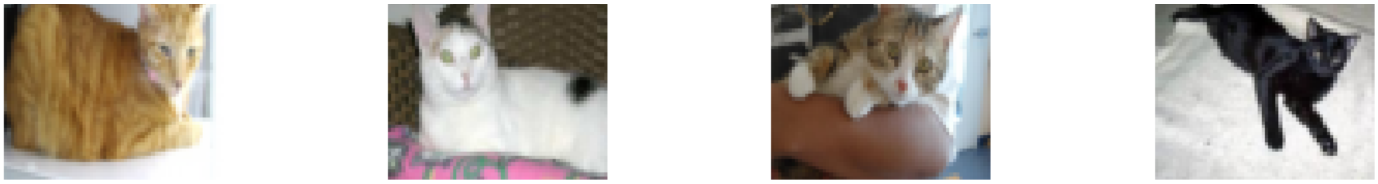
Epoch [15/50], Step [1174/1174], Discriminator Loss: 0.6179, Generator Loss: 2



Epoch [16/50], Step [1174/1174], Discriminator Loss: 0.6072, Generator Loss: 2

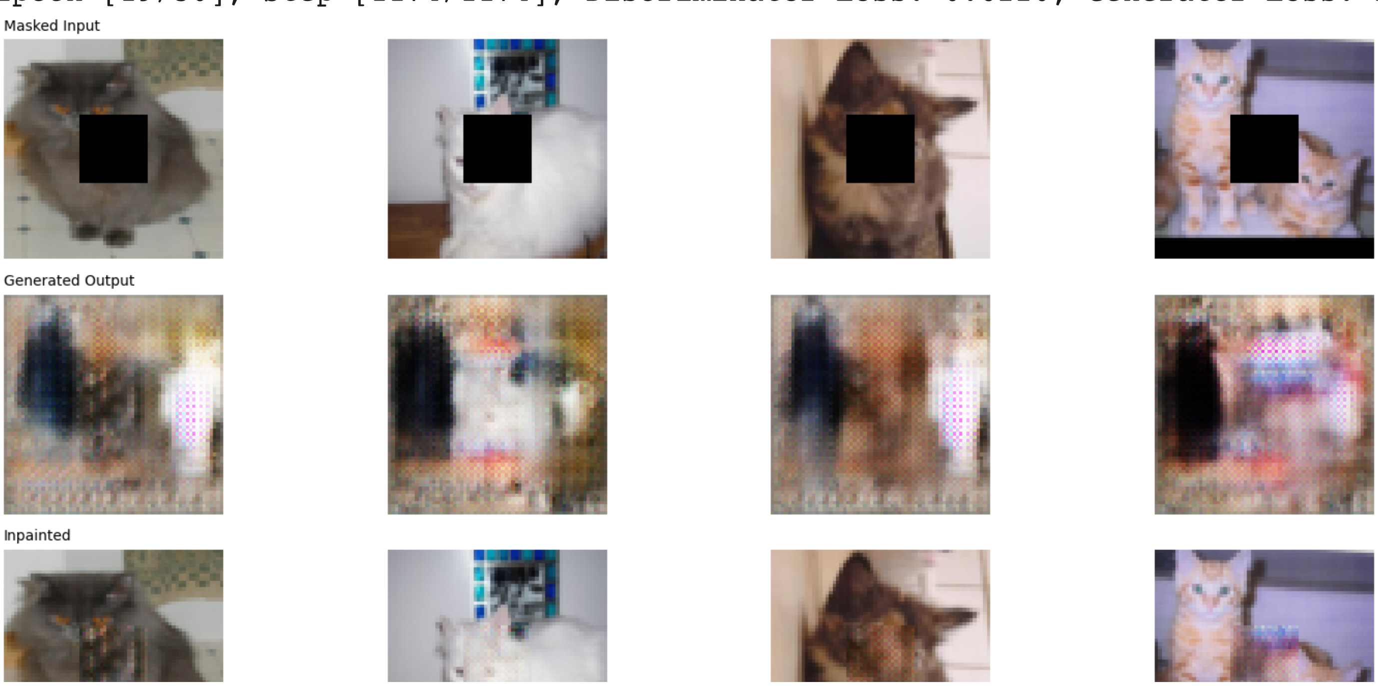

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [17/50], Step [1174/1174], Discriminator Loss: 0.6708, Generator Loss: 2
```



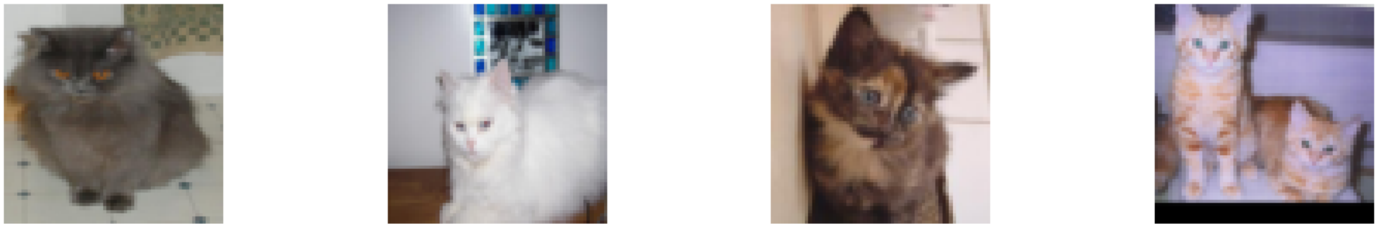


Epoch [18/50], Step [1174/1174], Discriminator Loss: 0.6265, Generator Loss: 2

Epoch [19/50], Step [1174/1174], Discriminator Loss: 0.6220, Generator Loss: 2

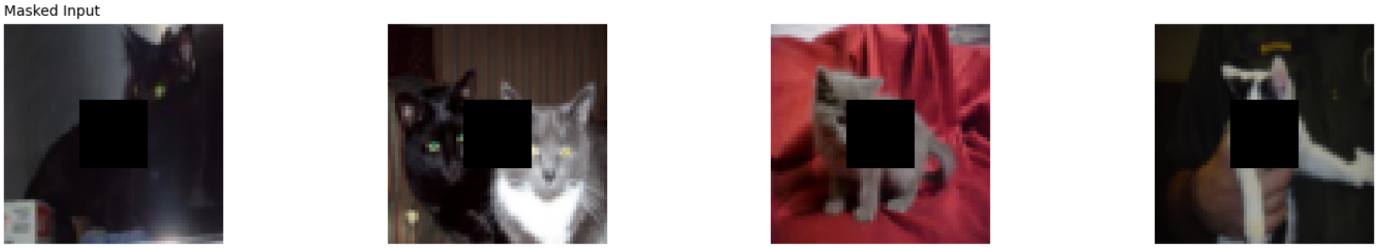


Ground Truth



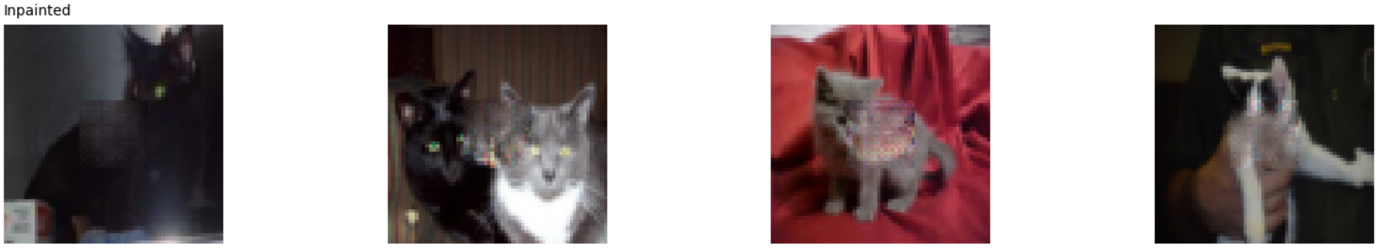
```
Epoch [20/50], Step [1174/1174], Discriminator Loss: 0.5632, Generator Loss: 2
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
```

Masked Input

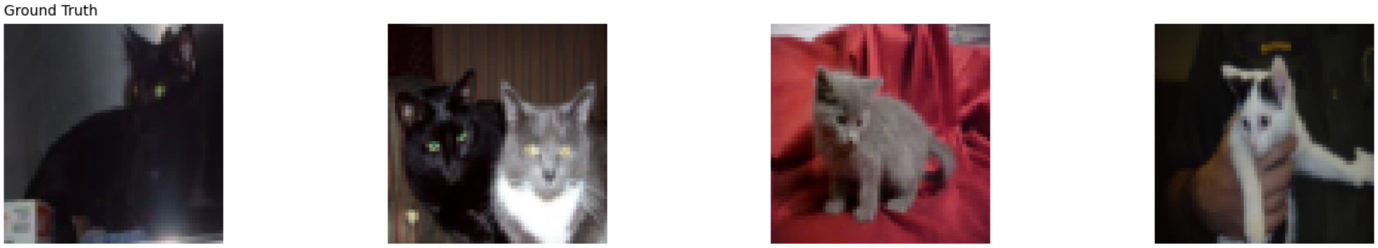


Generated Output

Inpainted

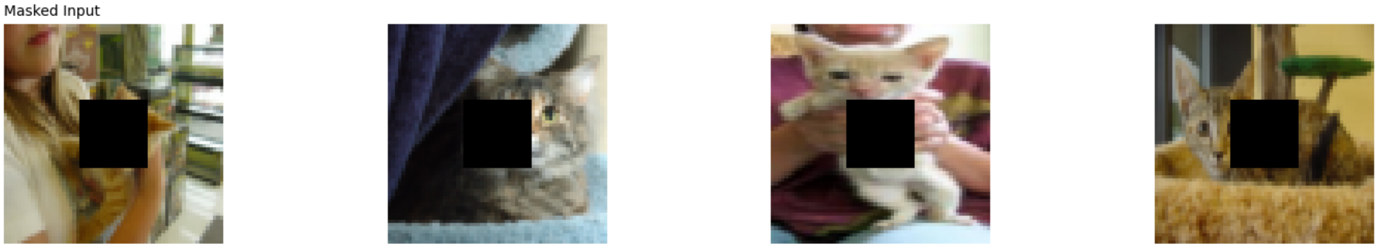


Ground Truth



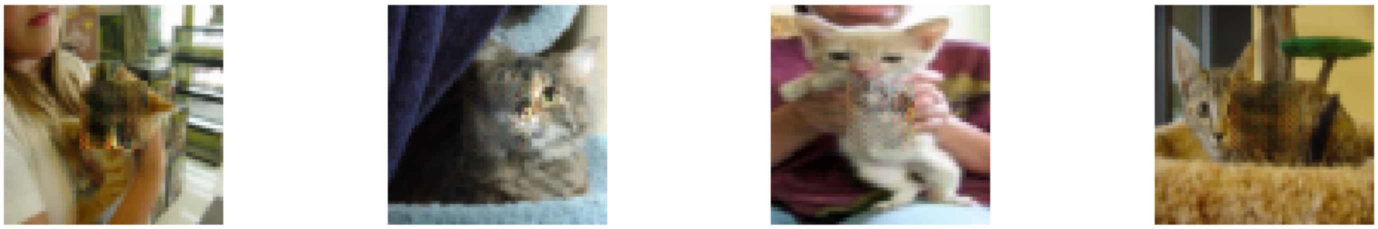
```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [21/50], Step [1174/1174], Discriminator Loss: 0.6216, Generator Loss: 2
```

Masked Input

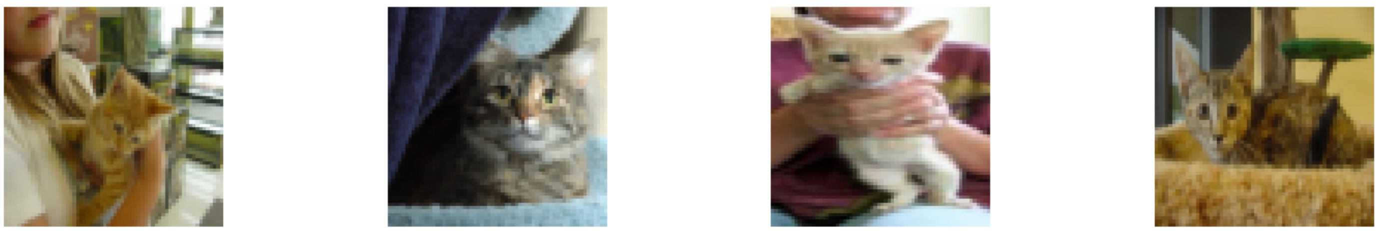


Generated Output

Inpainted

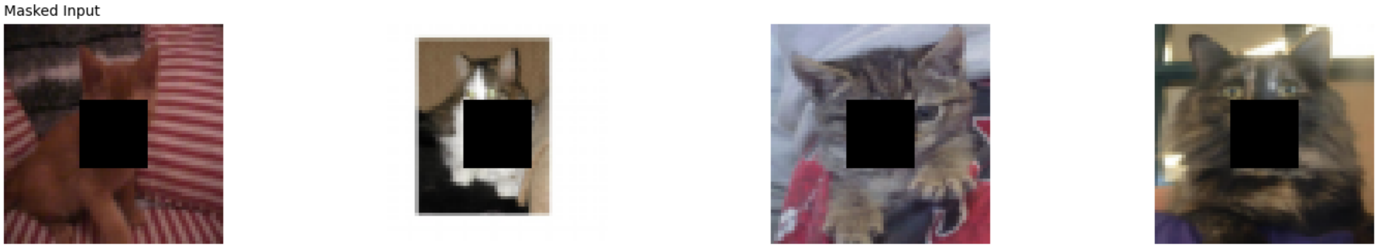


Ground Truth



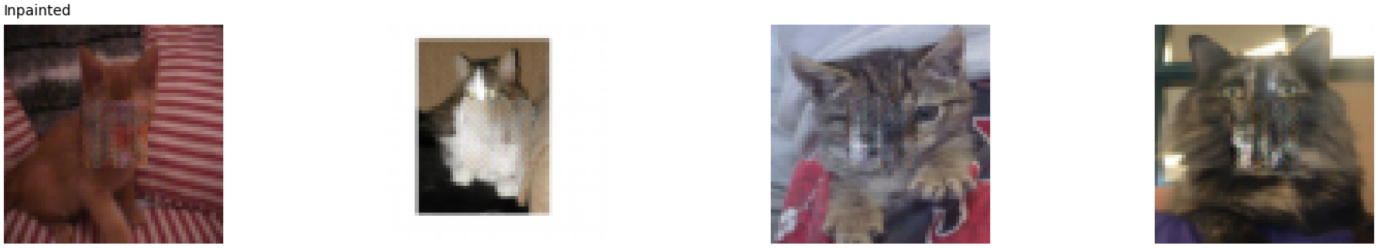
```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [22/50], Step [1174/1174], Discriminator Loss: 0.5694, Generator Loss: 2
```

Masked Input



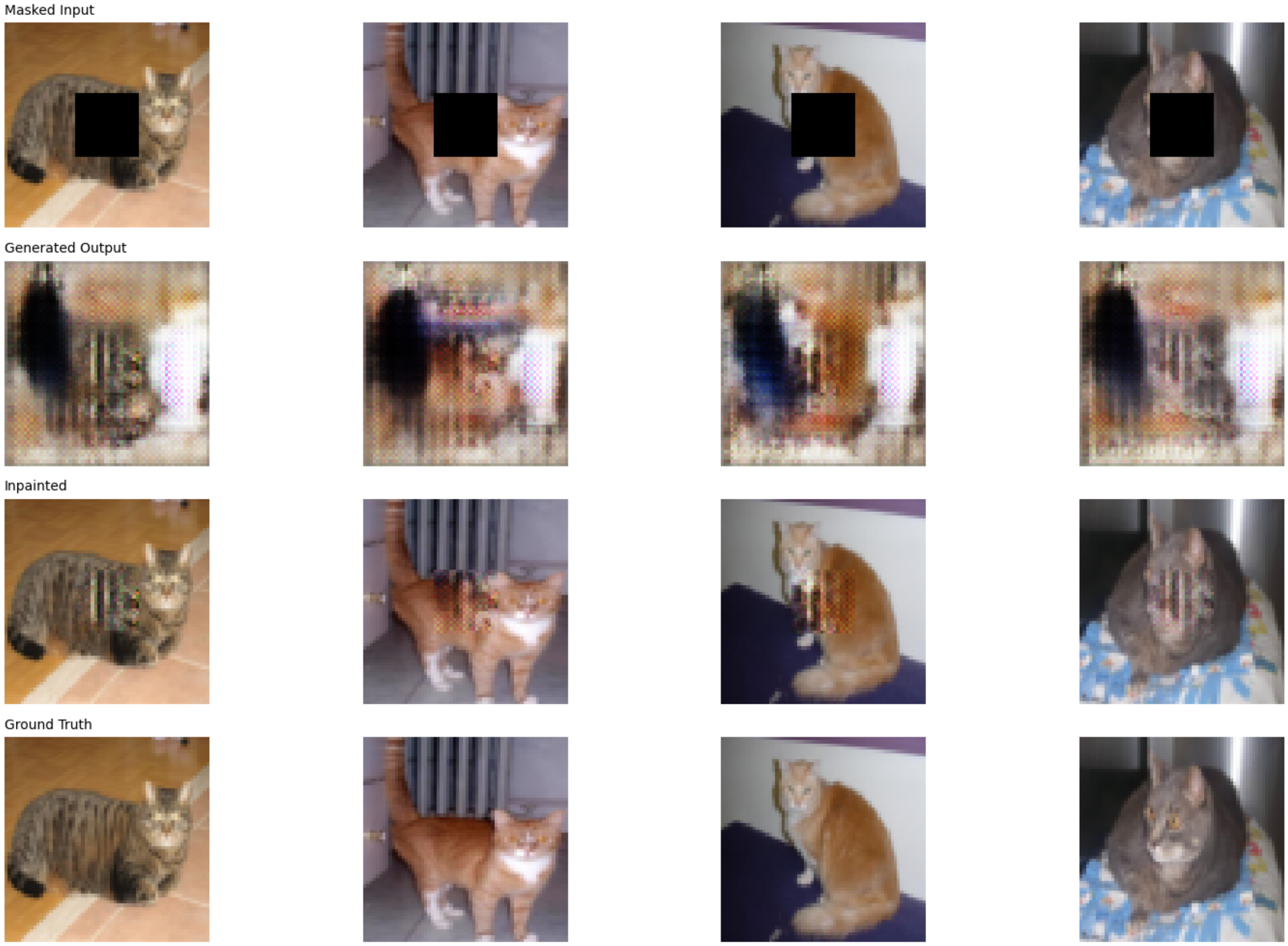
Generated Output

Inpainted



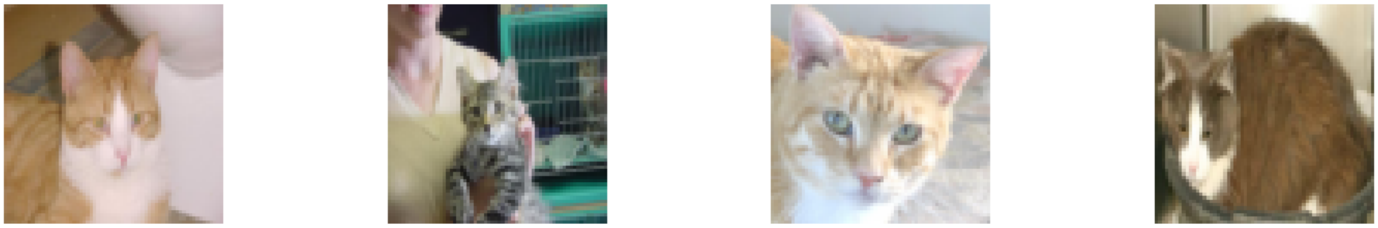
Ground Truth

```
Epoch [23/50], Step [1174/1174], Discriminator Loss: 0.5563, Generator Loss: 3
```



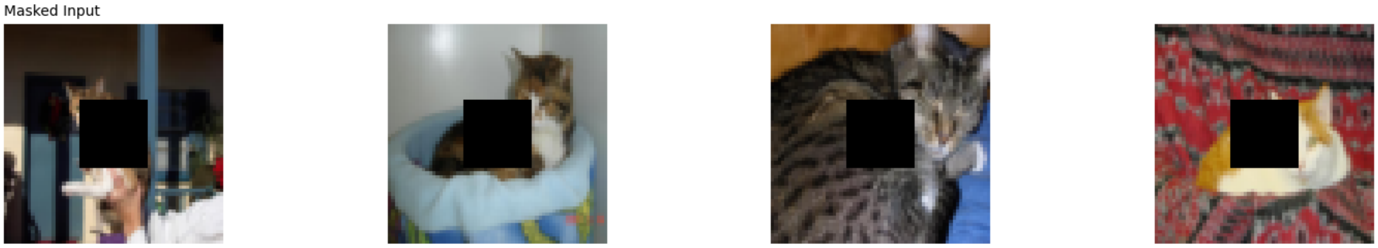

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [24/50], Step [1174/1174], Discriminator Loss: 0.5722, Generator Loss: 2
```

Ground Truth



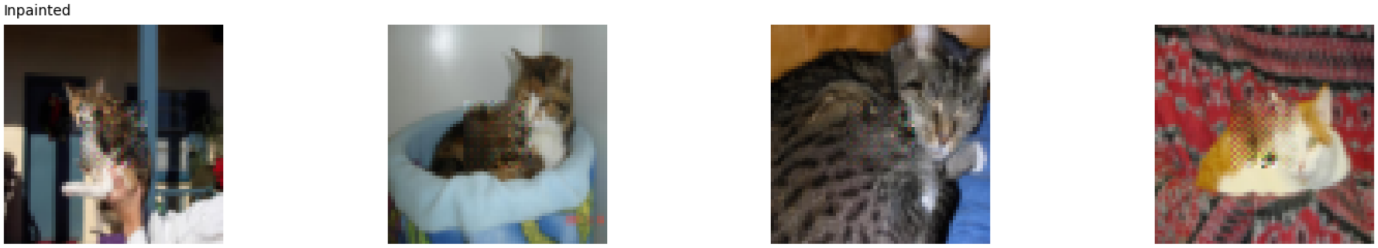
```
Epoch [25/50], Step [1174/1174], Discriminator Loss: 0.5597, Generator Loss: 3
```

Masked Input

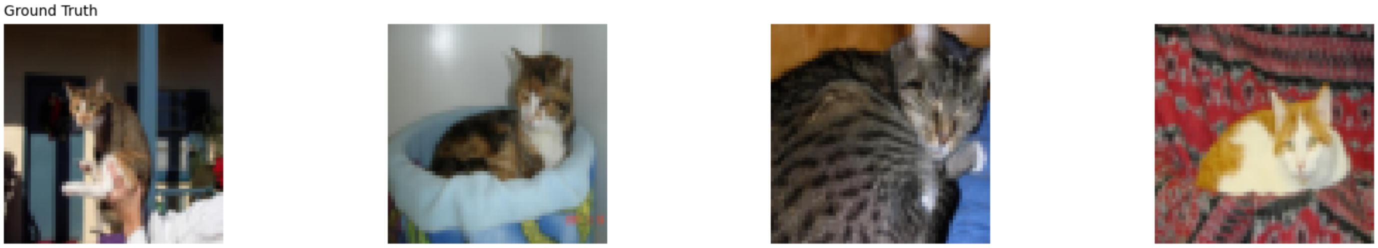


Generated Output

Inpainted

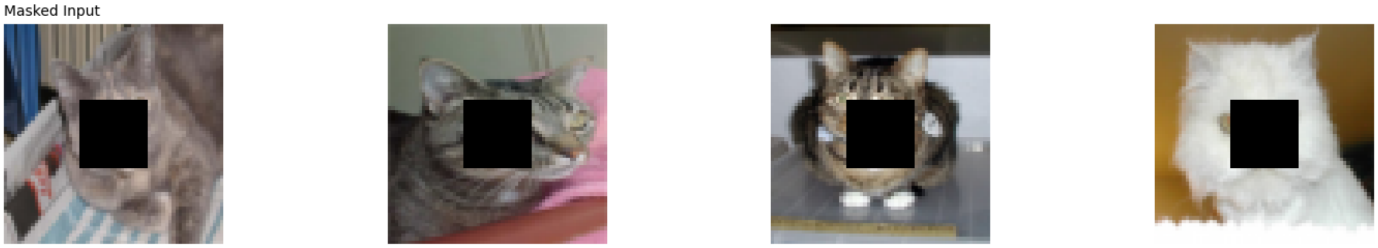


Ground Truth



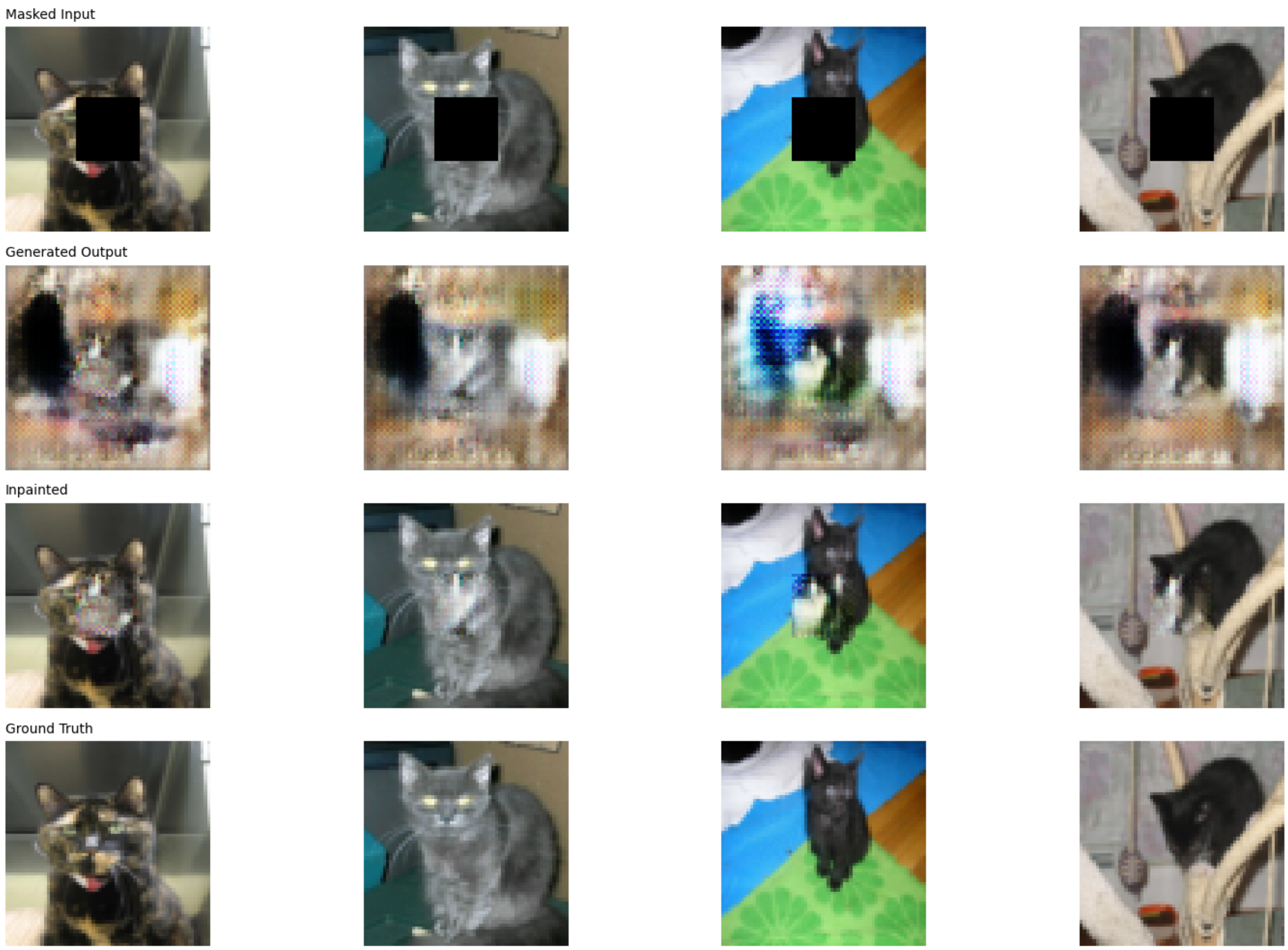
```
Epoch [26/50], Step [1174/1174], Discriminator Loss: 0.5908, Generator Loss: 2
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
```

Masked Input



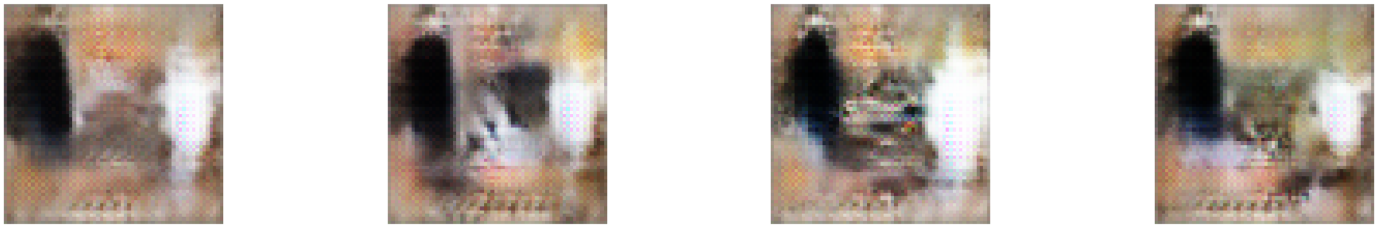
Generated Output

Epoch [27/50], Step [1174/1174], Discriminator Loss: 0.5831, Generator Loss: 3

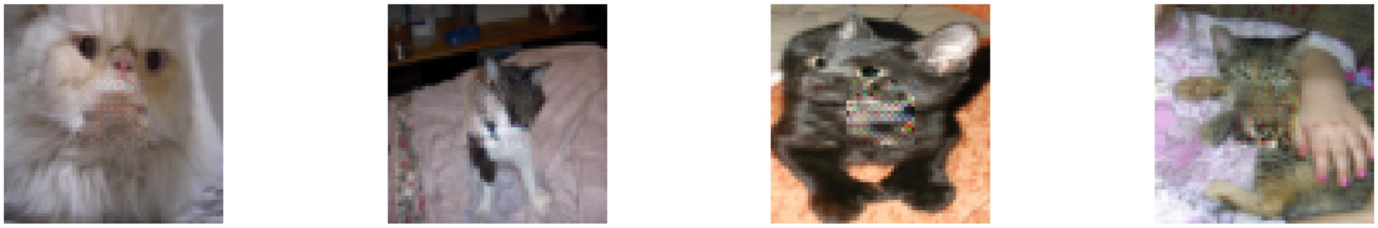


Epoch [28/50], Step [1174/1174], Discriminator Loss: 0.5639, Generator Loss: 3

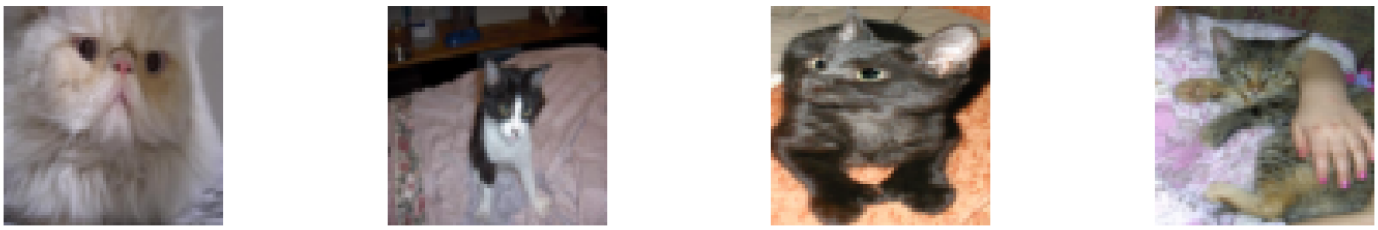
Generated Output



Inpainted

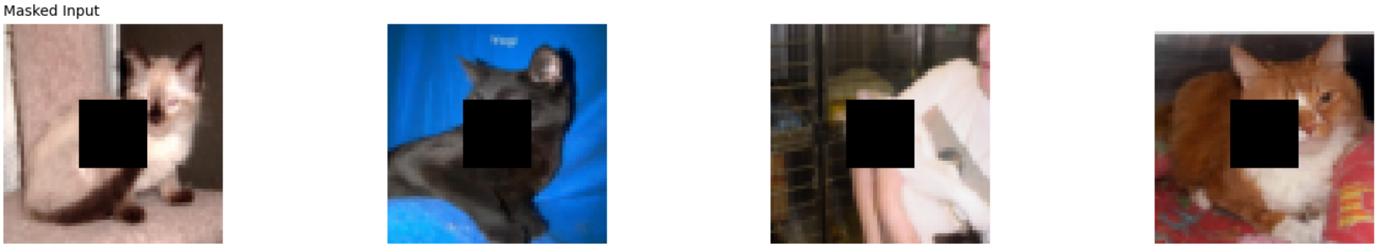


Ground Truth



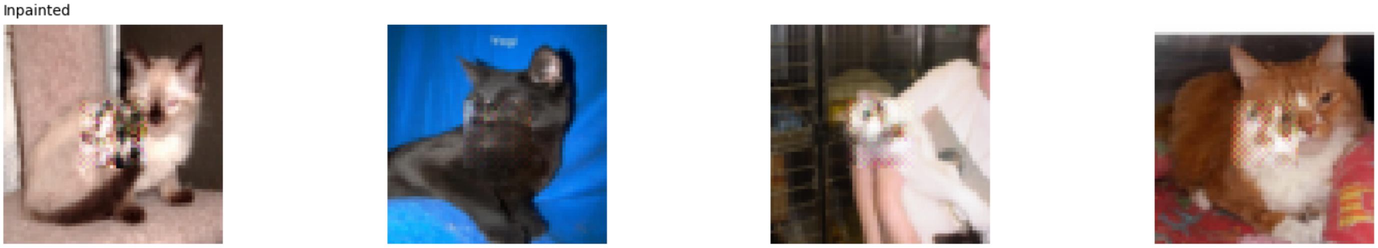
Epoch [29/50], Step [1174/1174], Discriminator Loss: 0.5256, Generator Loss: 3

Masked Input

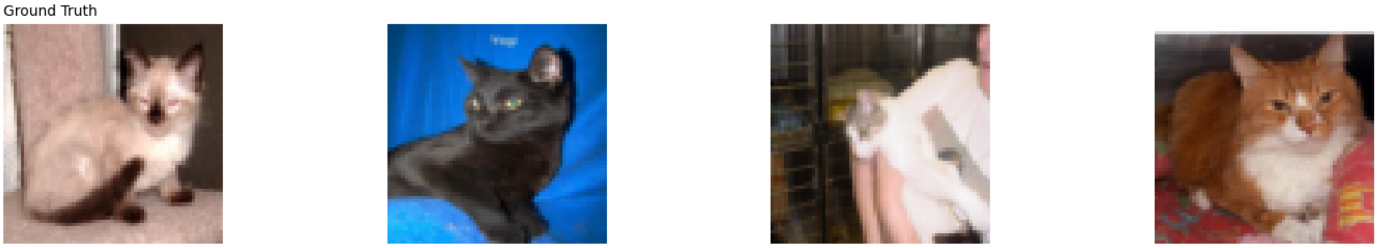


Generated Output

Inpainted



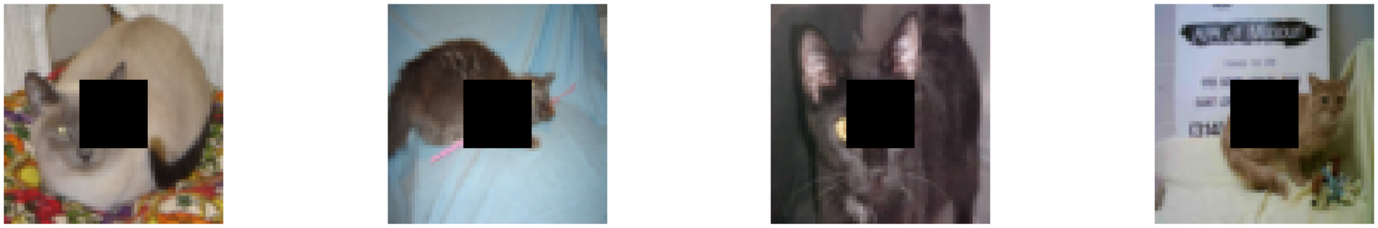
Ground Truth



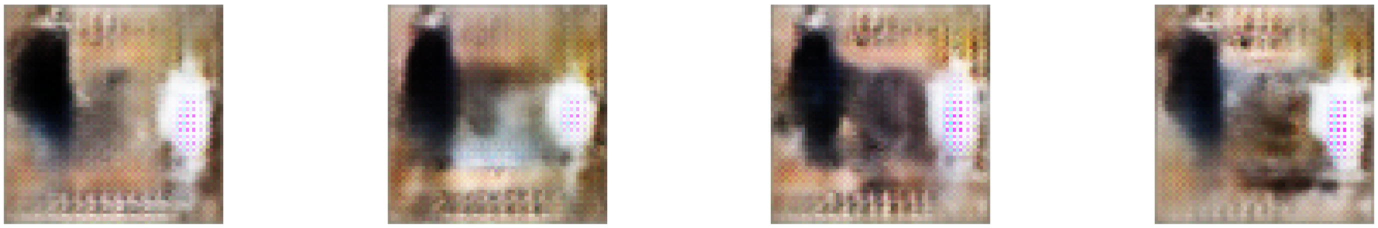
Epoch [30/50], Step [1174/1174], Discriminator Loss: 0.5506, Generator Loss:

Epoch [30/50], Step [1174/1174], Discriminator Loss: 0.5506, Generator Loss: 3

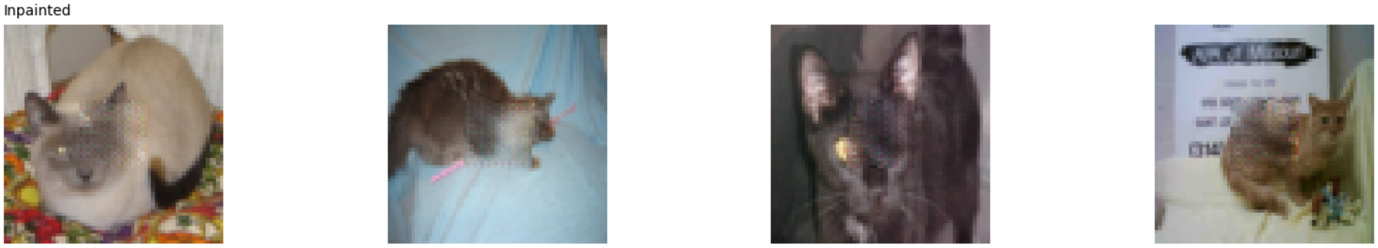
Masked Input



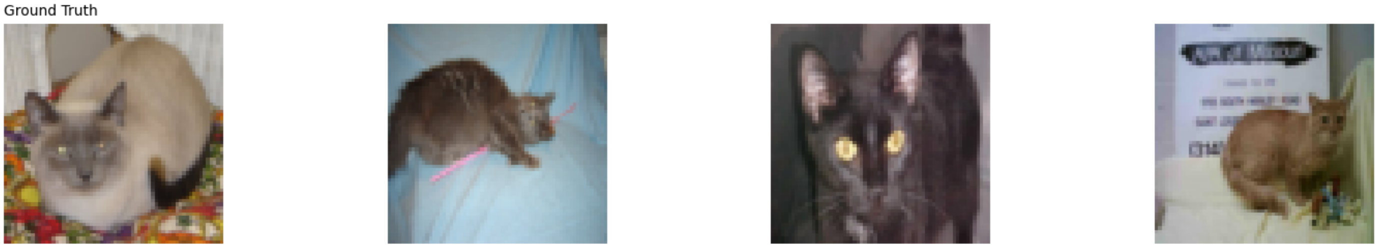
Generated Output



Inpainted



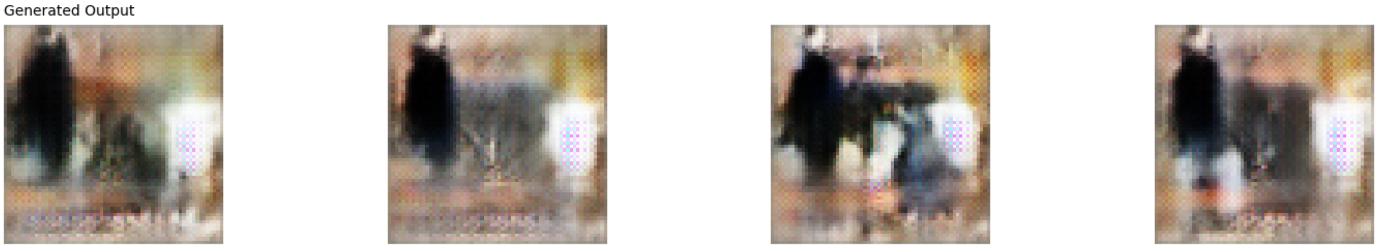
Ground Truth



Epoch [31/50], Step [1174/1174], Discriminator Loss: 0.5170, Generator Loss: 3

Masked Input

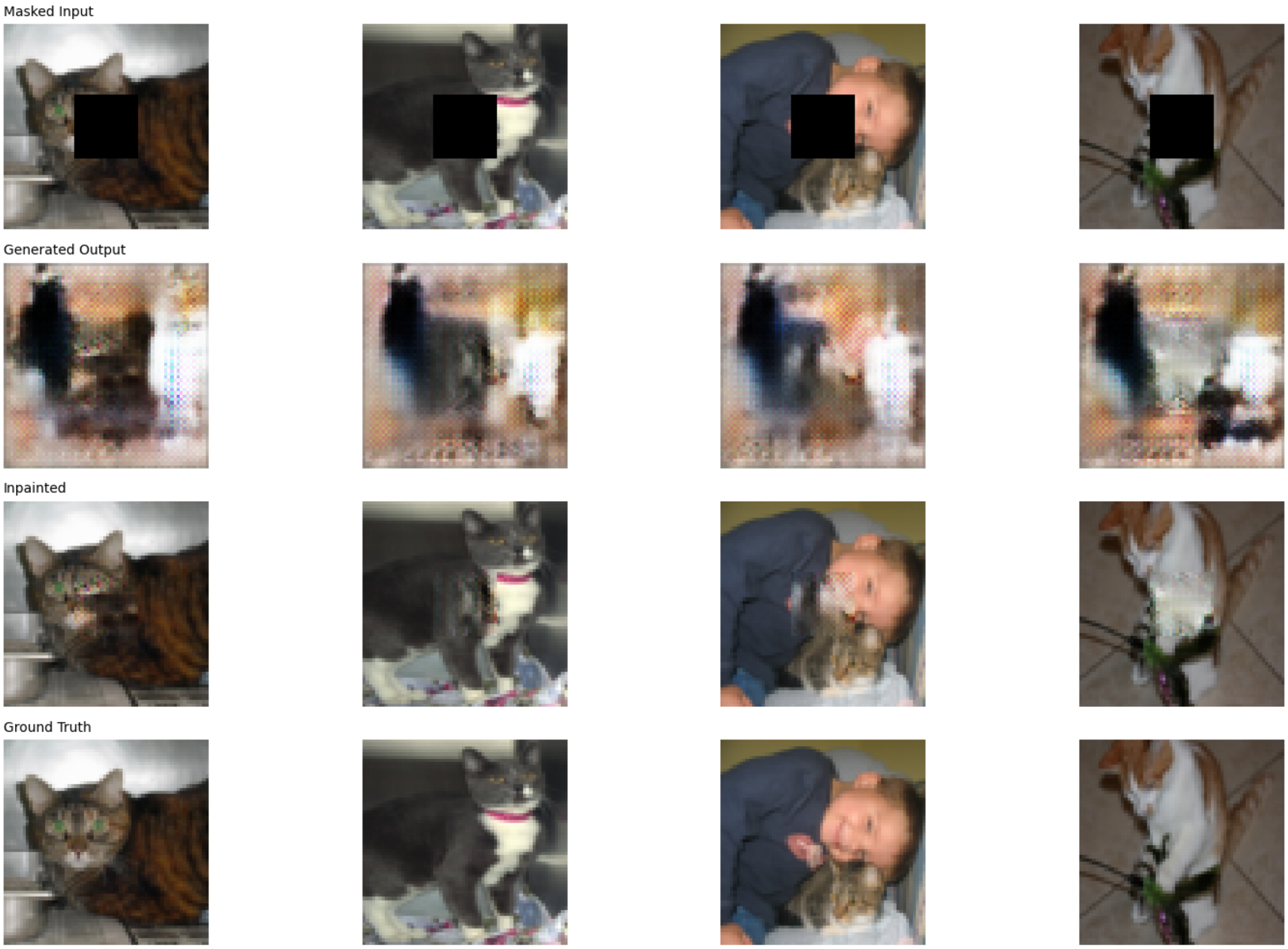
Generated Output



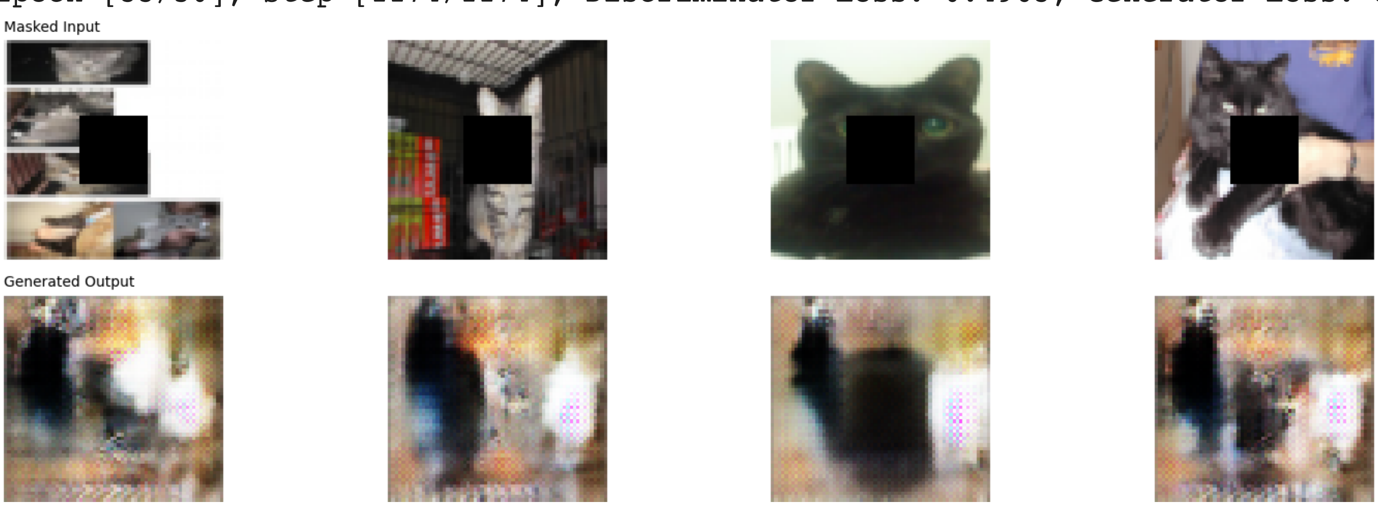
Inpainted

Ground Truth

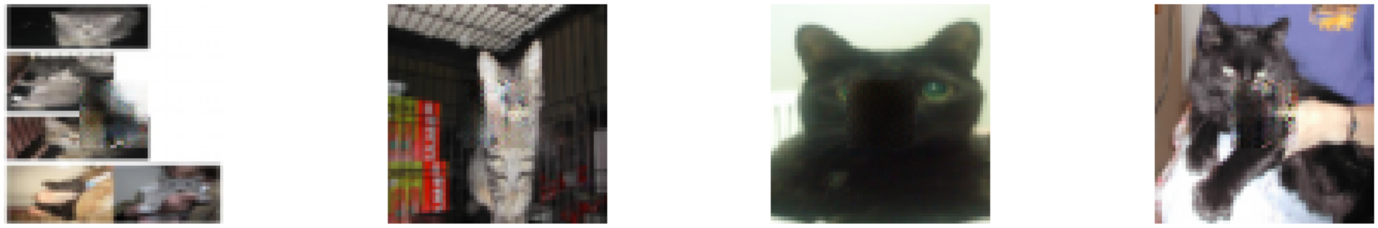
```
Epoch [32/50], Step [1174/1174], Discriminator Loss: 0.5649, Generator Loss: 3
```



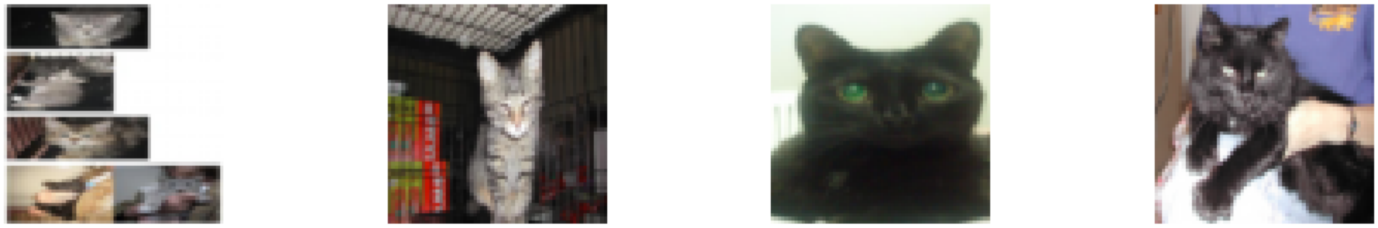

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [33/50], Step [1174/1174], Discriminator Loss: 0.4963, Generator Loss: 3
```



Inpainted

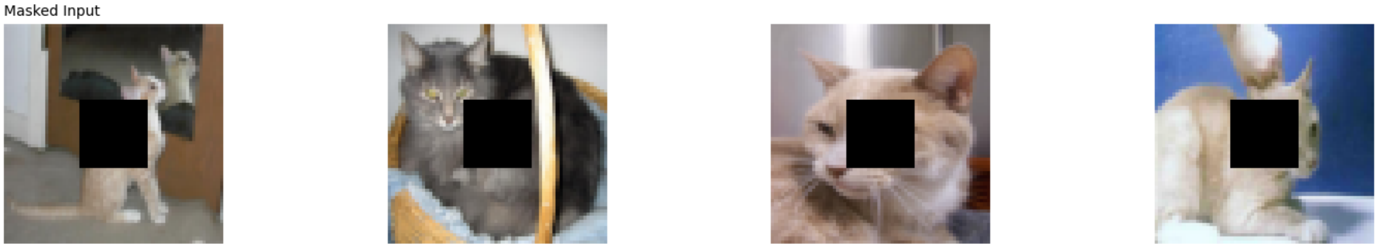


Ground Truth



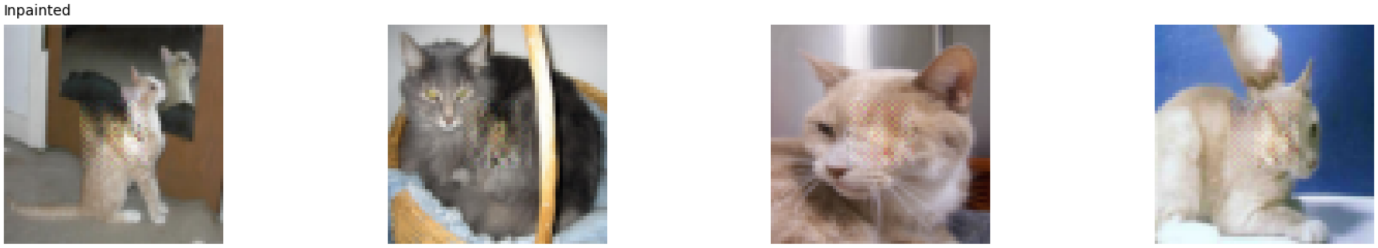
```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [34/50], Step [1174/1174], Discriminator Loss: 0.5523, Generator Loss: 3
```

Masked Input

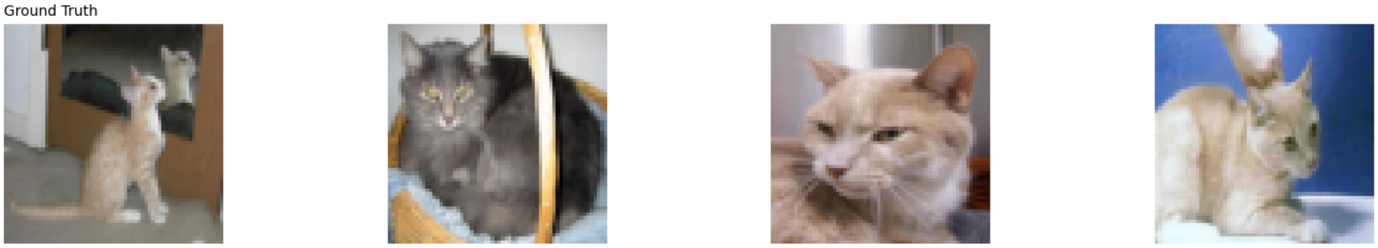


Generated Output

Inpainted

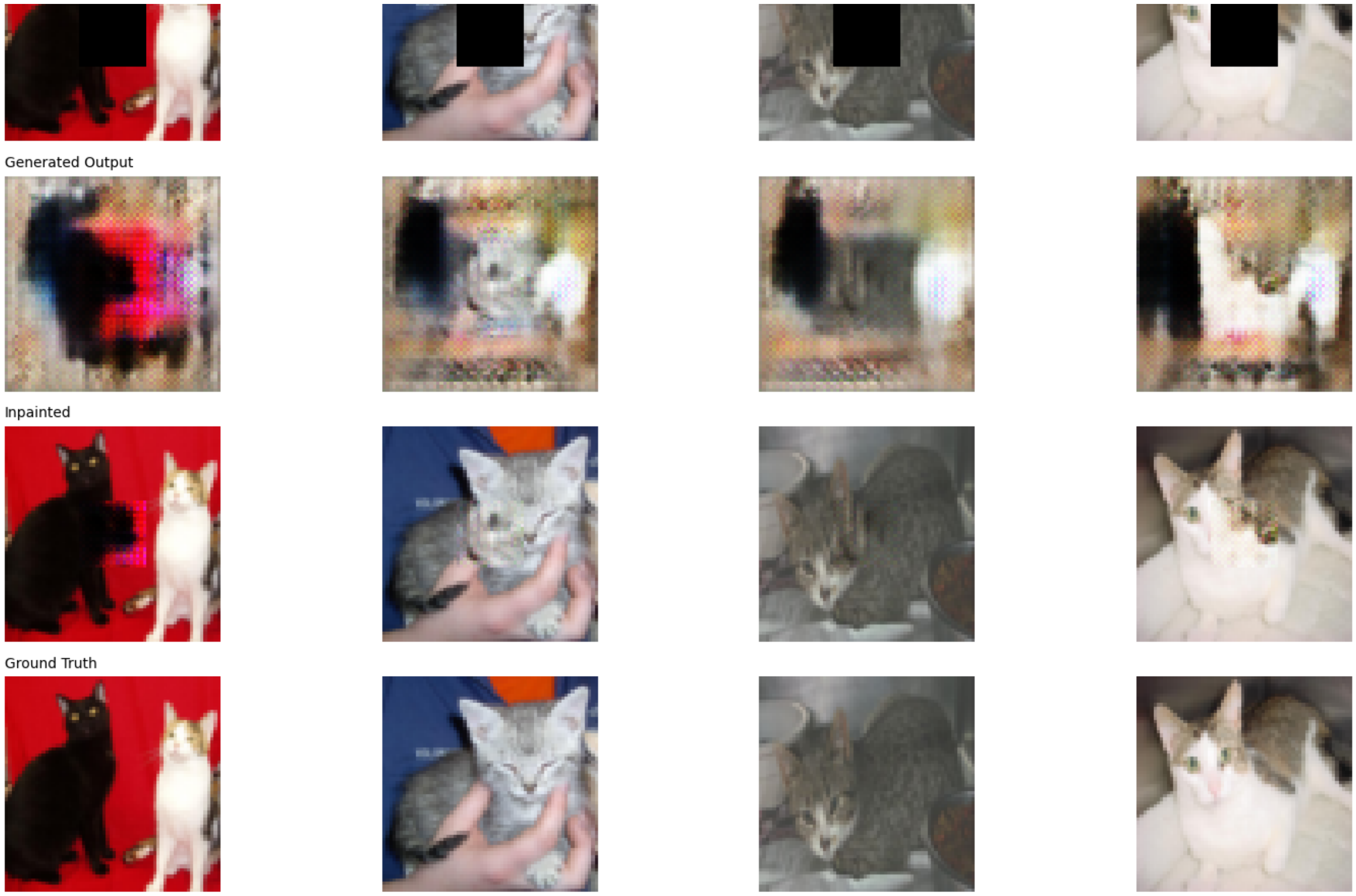


Ground Truth

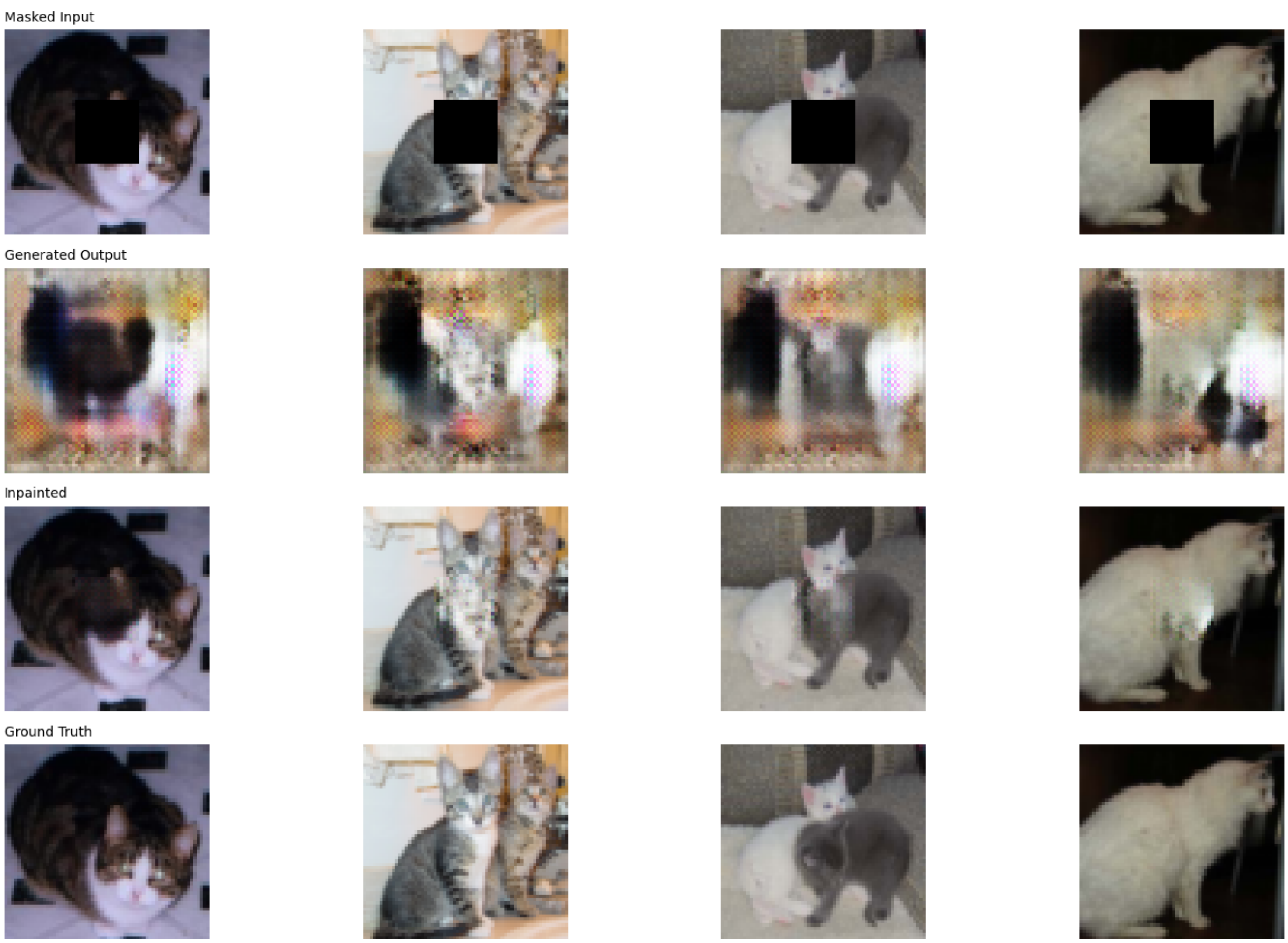


```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [35/50], Step [1174/1174], Discriminator Loss: 0.4810, Generator Loss: 3
```

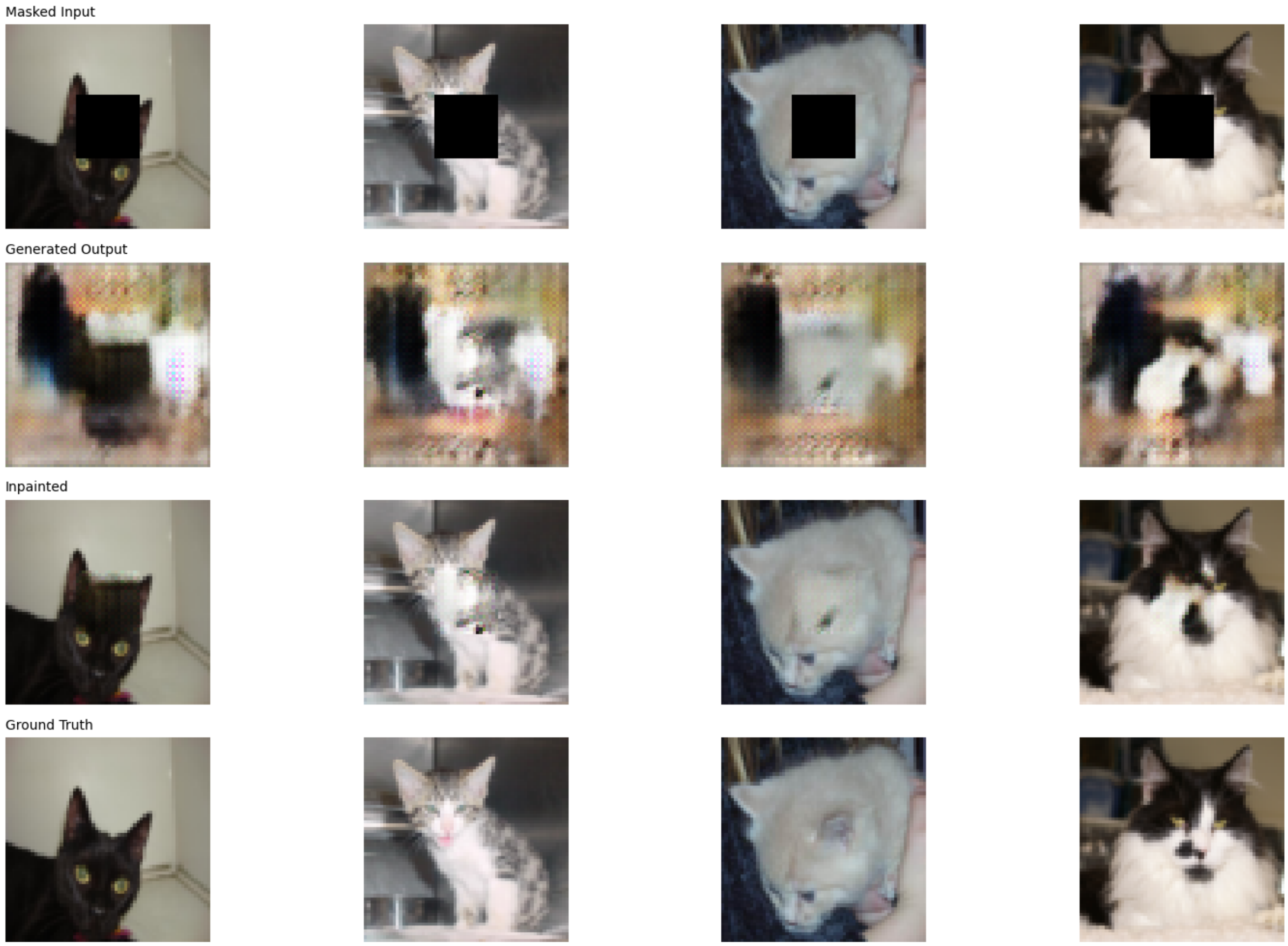
Masked Input



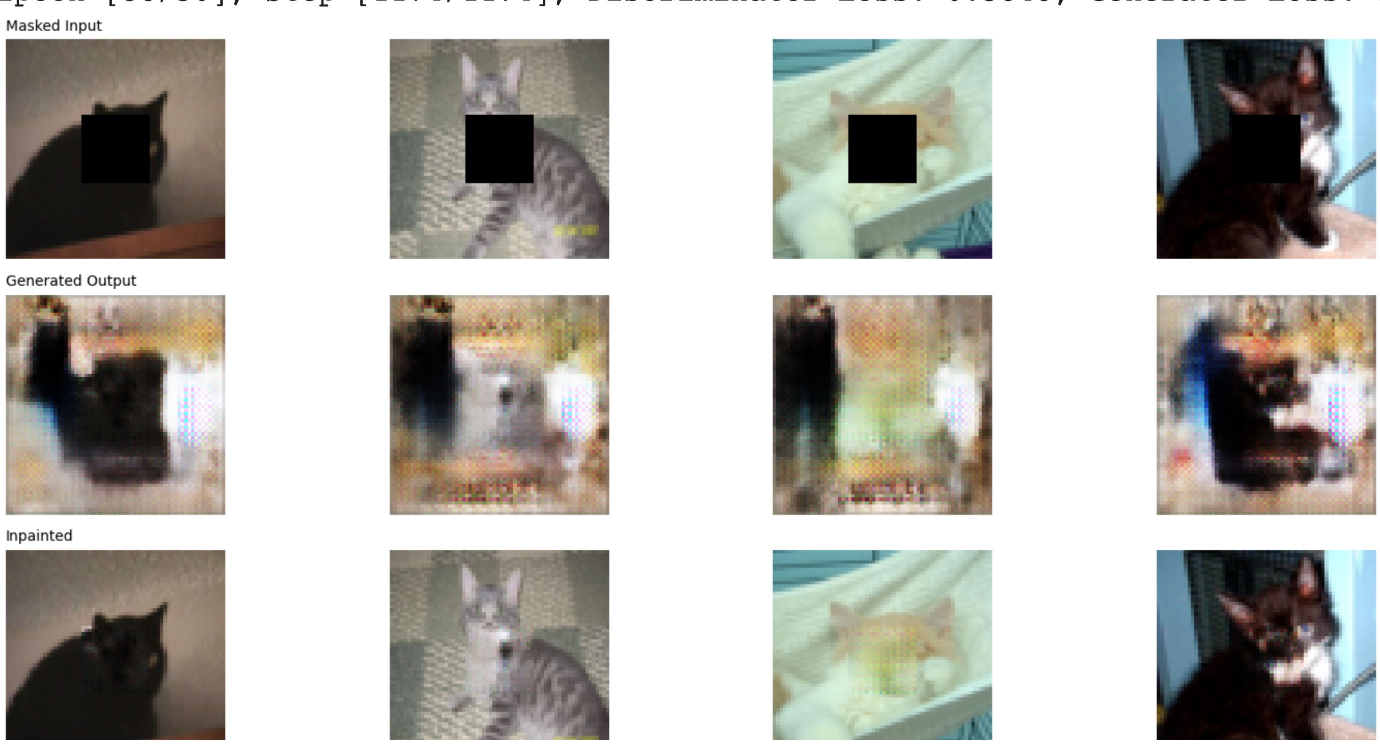
Epoch [36/50], Step [1174/1174], Discriminator Loss: 0.5378, Generator Loss: 3

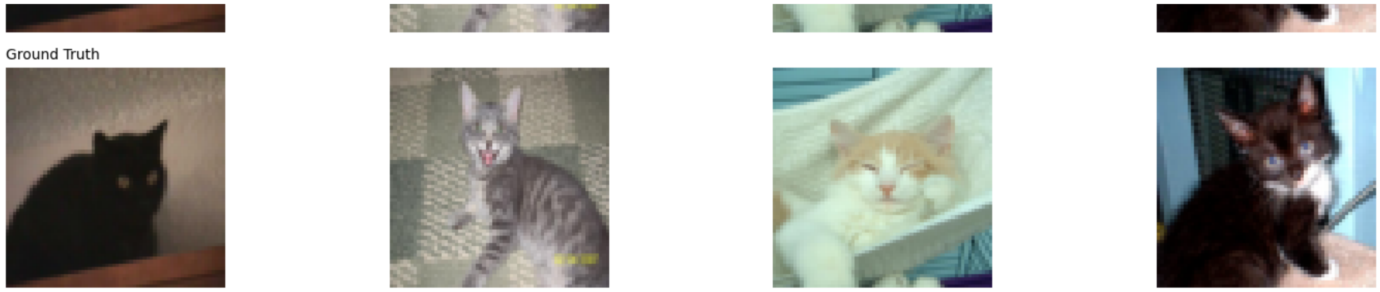


```
Epoch [37/50], Step [1174/1174], Discriminator Loss: 0.5012, Generator Loss: 3
```

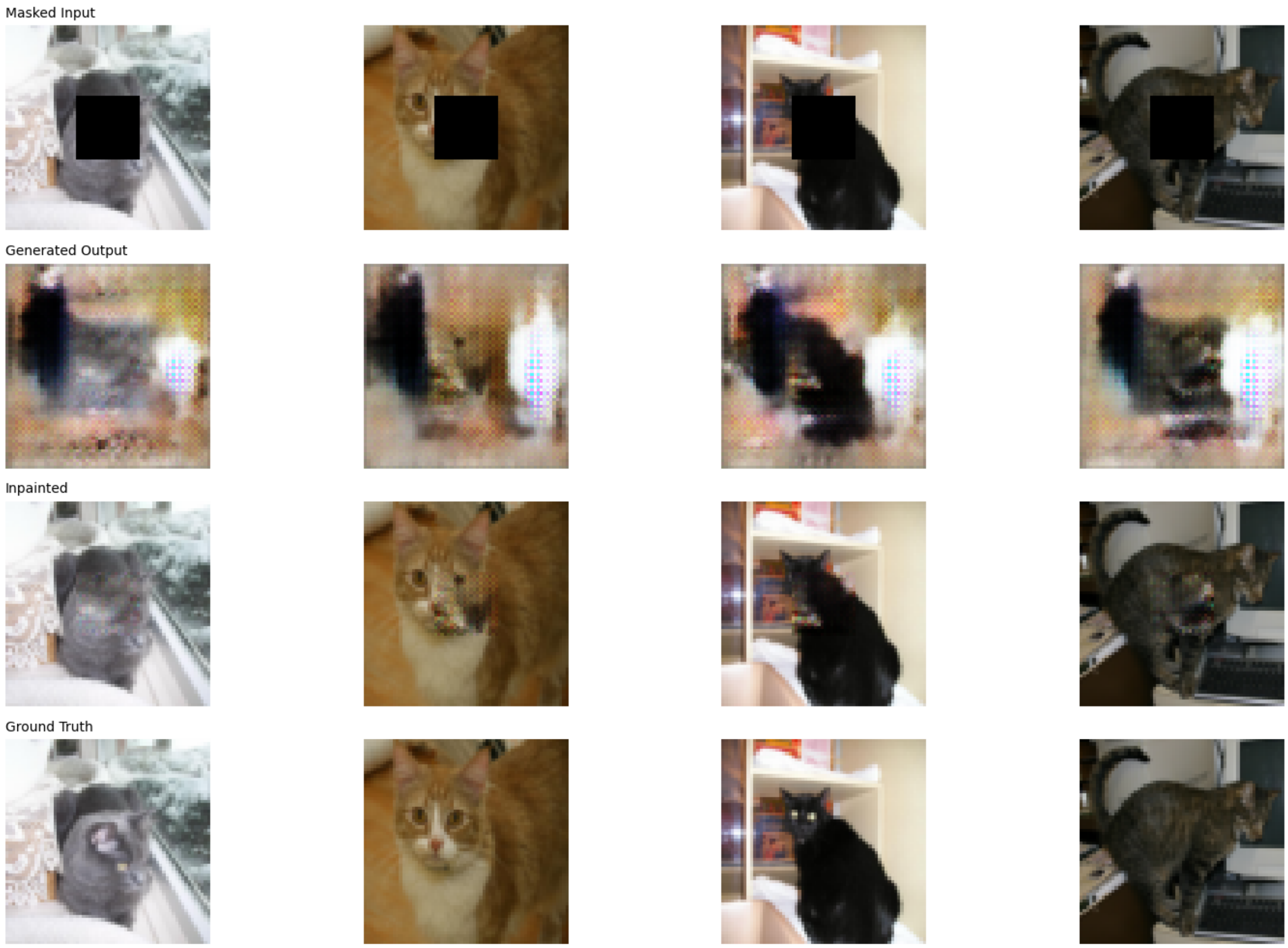



```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [38/50], Step [1174/1174], Discriminator Loss: 0.5848, Generator Loss: 3
```



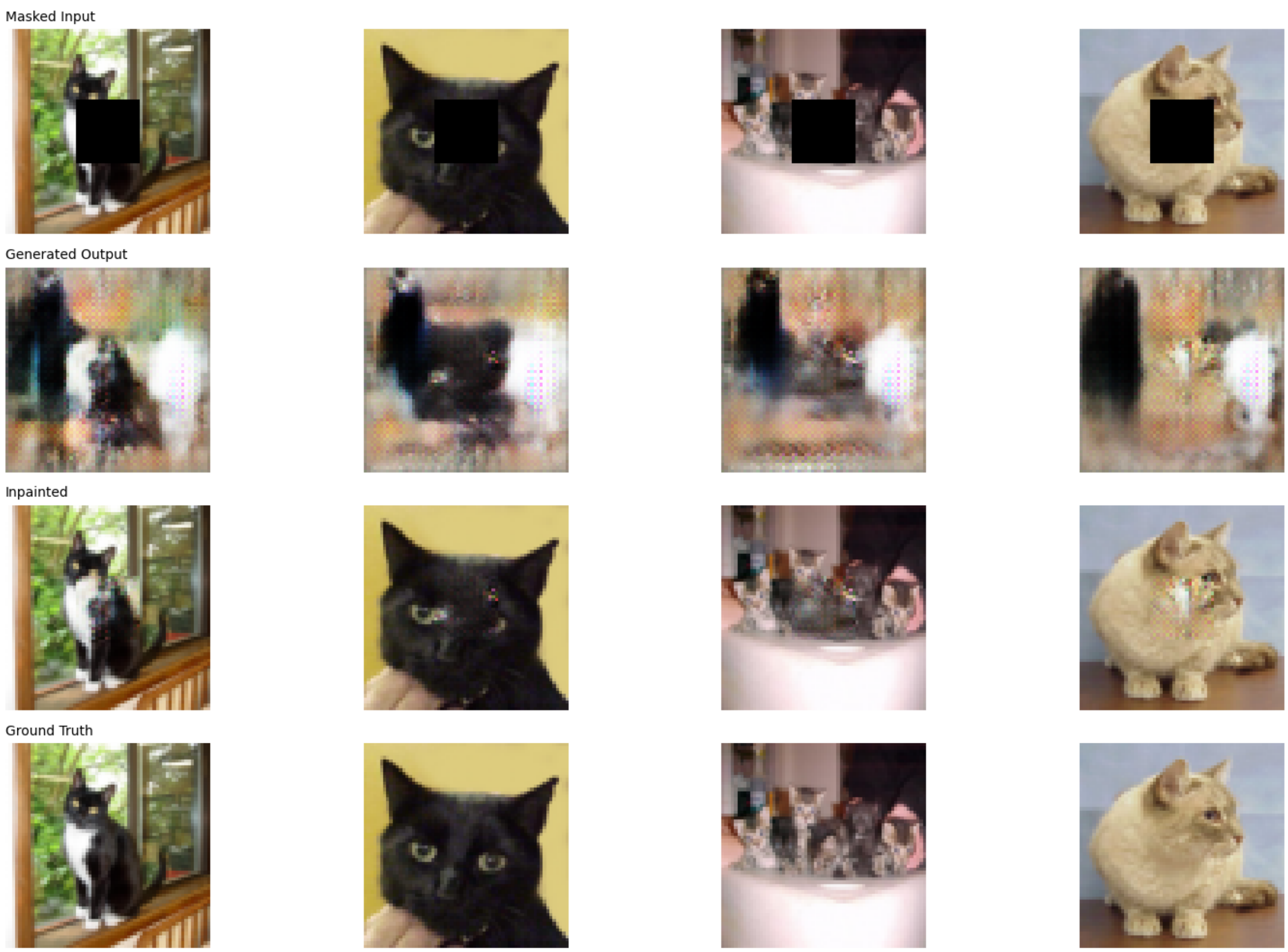



```
Epoch [39/50], Step [1174/1174], Discriminator Loss: 0.5257, Generator Loss: 3
```

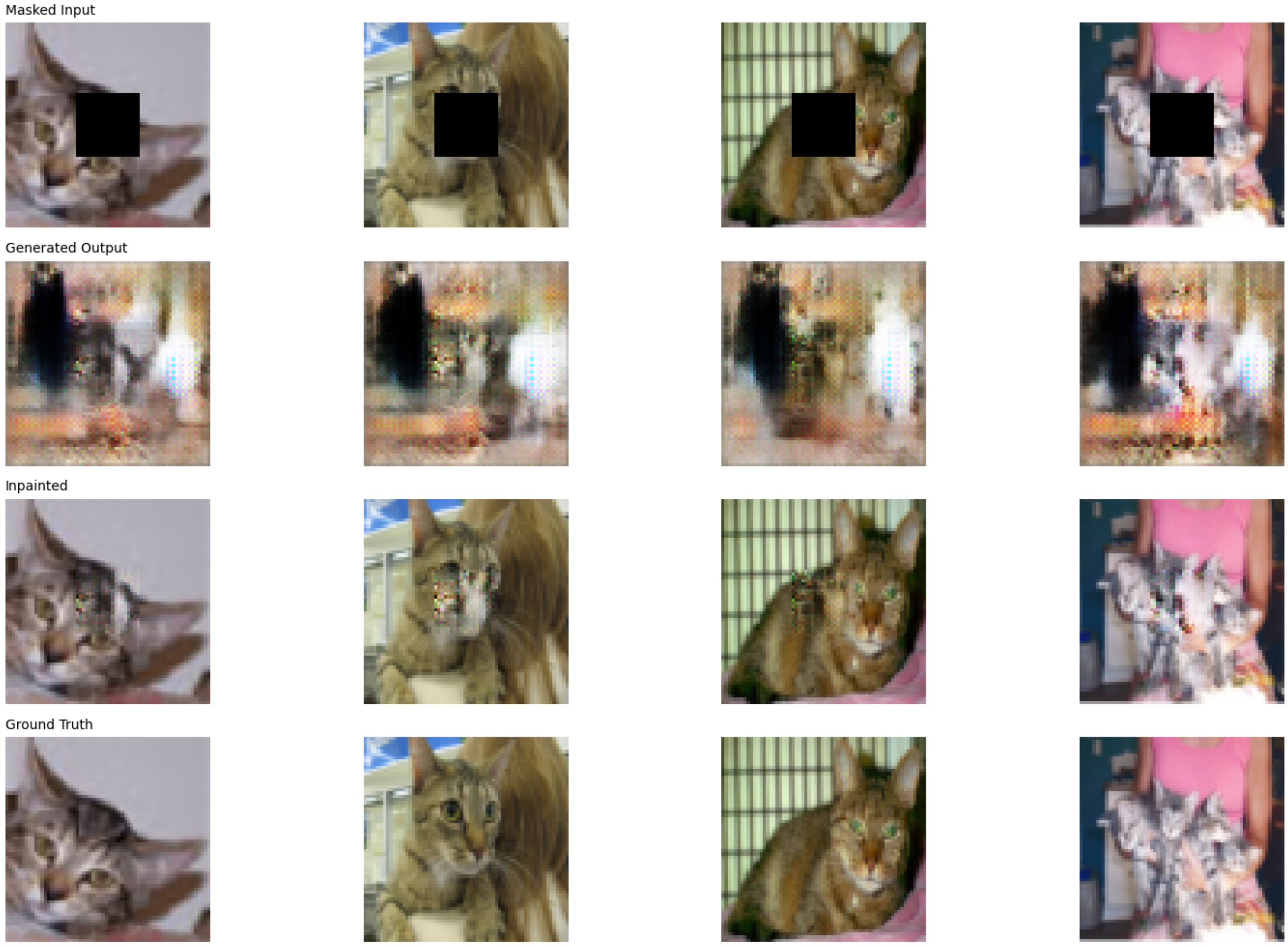



```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [40/50], Step [1174/1174], Discriminator Loss: 0.4931, Generator Loss: 3
```


```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
WARNING:matplotlib.image:Clipping input data to the valid range for imshow wit
Epoch [41/50], Step [1174/1174], Discriminator Loss: 0.4746, Generator Loss: 3
```

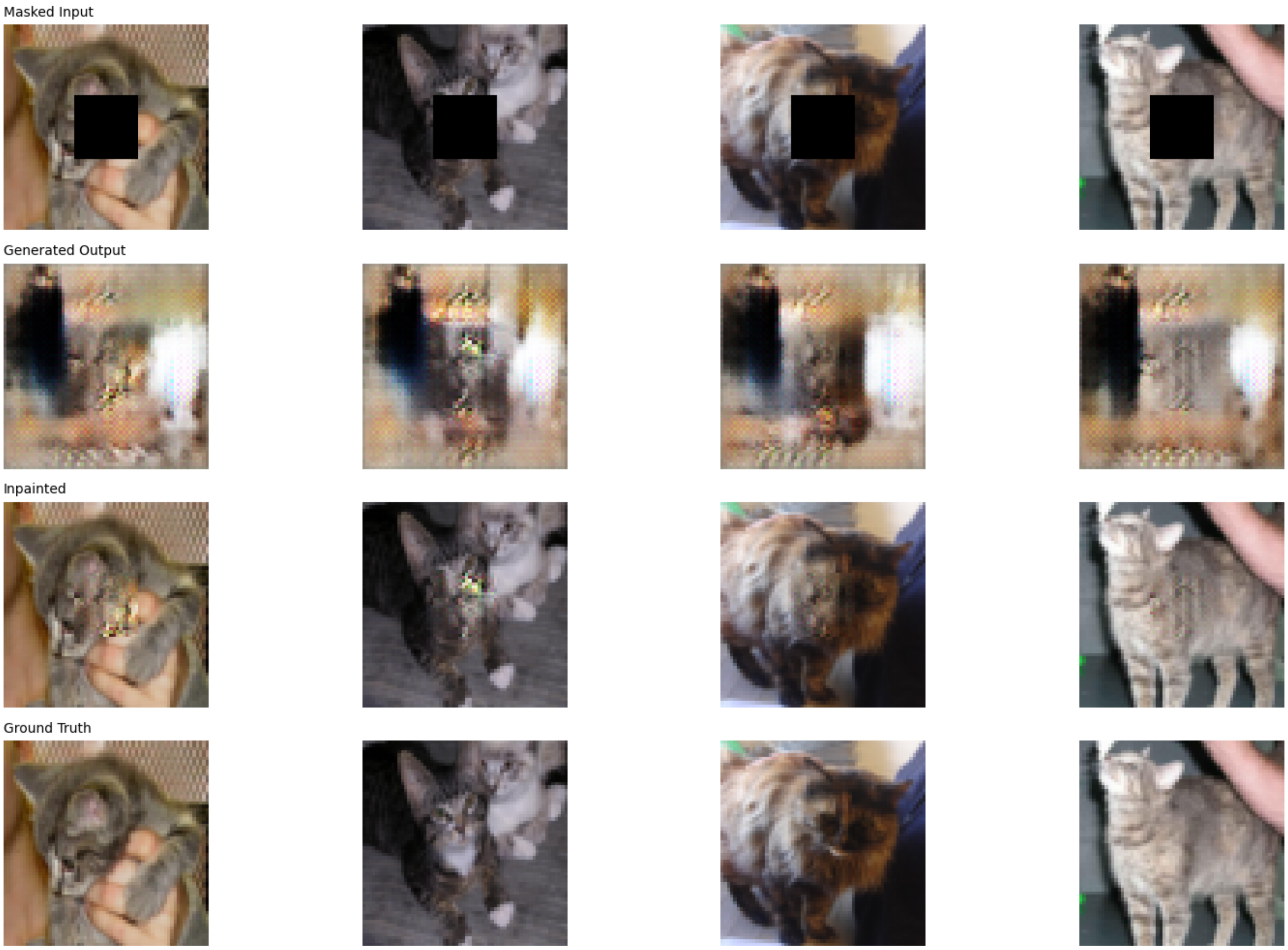


Epoch [42/50], Step [1174/1174], Discriminator Loss: 0.5233, Generator Loss: 3



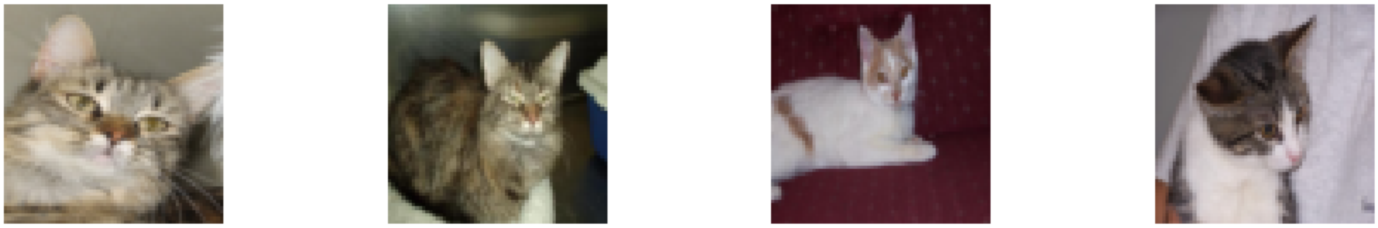
Epoch [43/50], Step [1174/1174], Discriminator Loss: 0.5043, Generator Loss: 3

Epoch [44/50], Step [1174/1174], Discriminator Loss: 0.4936, Generator Loss: 3



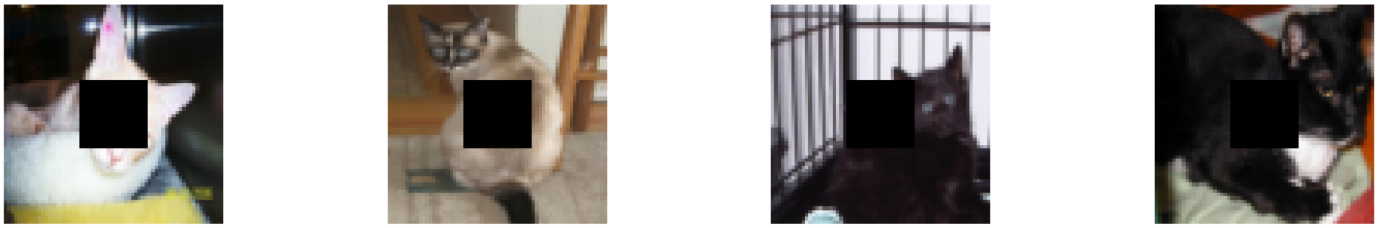
Epoch [45/50], Step [1174/1174], Discriminator Loss: 0.5493, Generator Loss: 3

Ground Truth

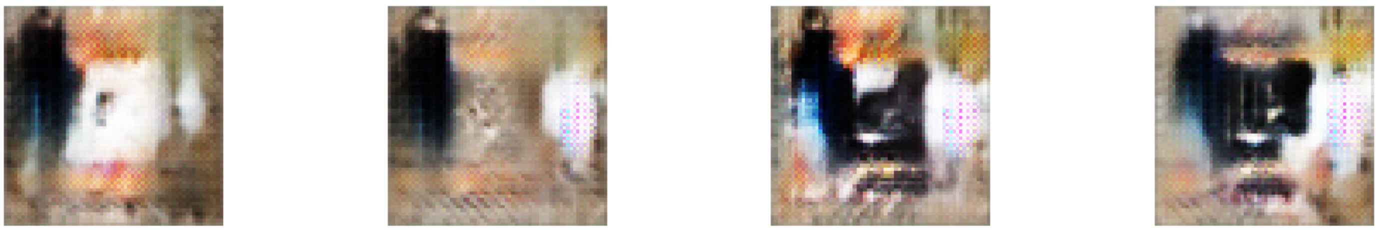


Epoch [46/50], Step [1174/1174], Discriminator Loss: 0.4963, Generator Loss: 3

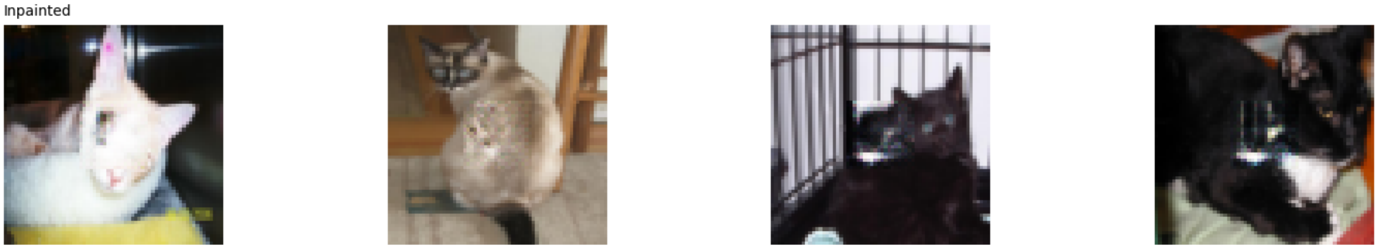
Masked Input



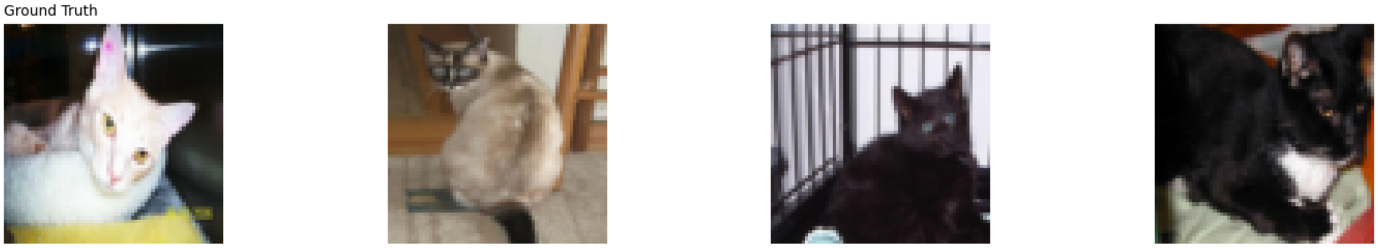
Generated Output



Inpainted

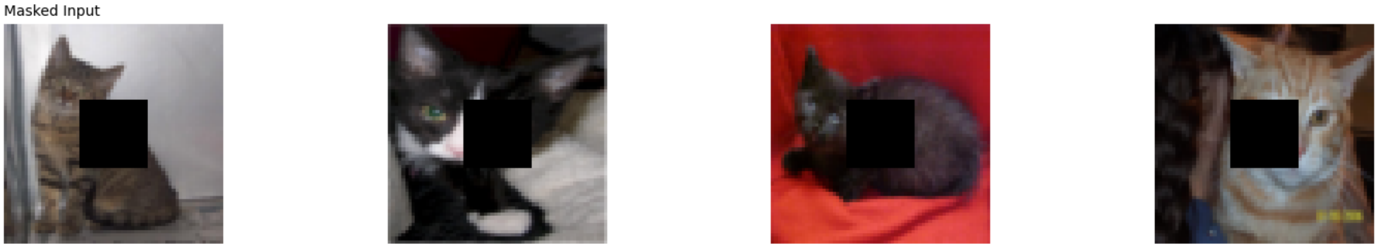


Ground Truth



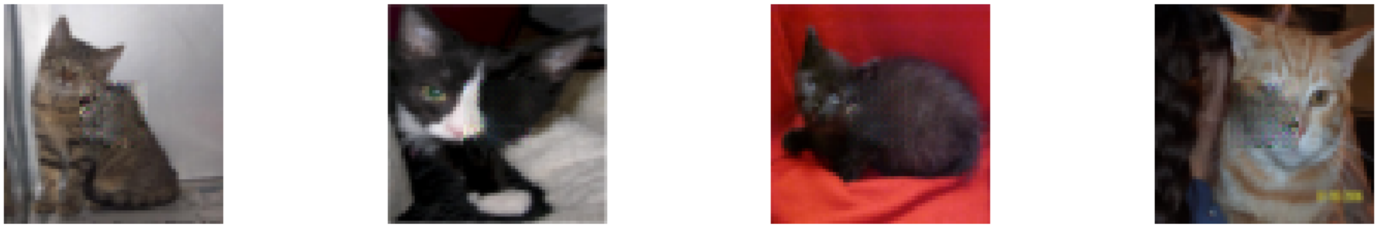
Epoch [47/50], Step [1174/1174], Discriminator Loss: 0.4893, Generator Loss: 3

Masked Input

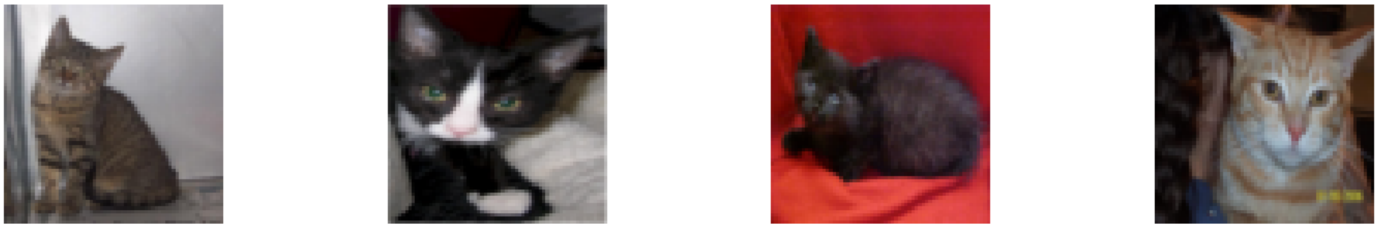


Generated Output

Inpainted

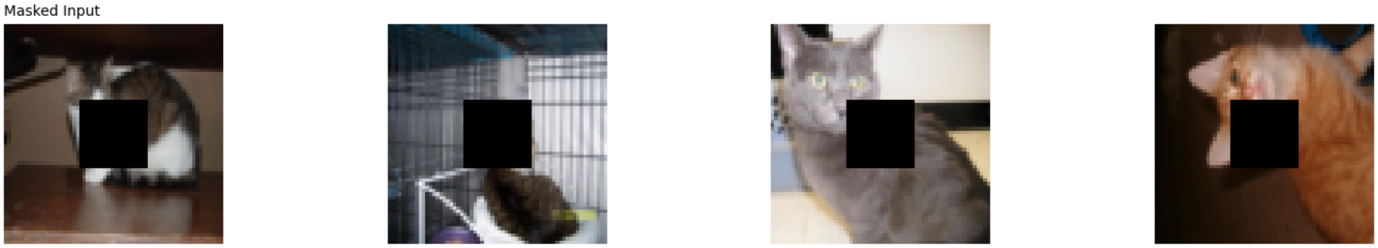


Ground Truth



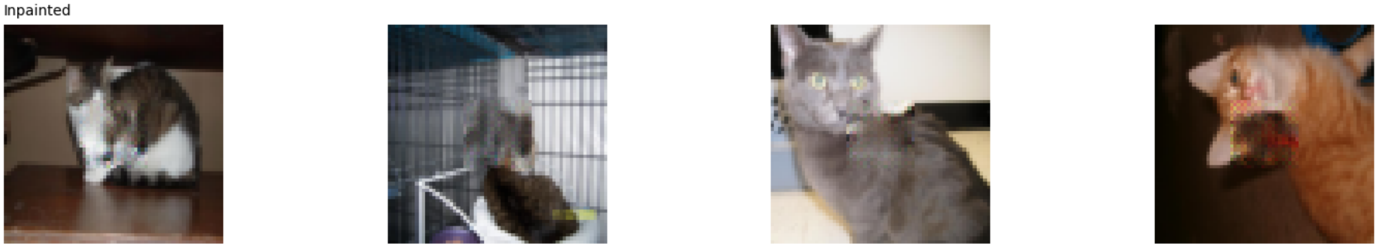
Epoch [48/50], Step [1174/1174], Discriminator Loss: 0.4578, Generator Loss: 3

Masked Input

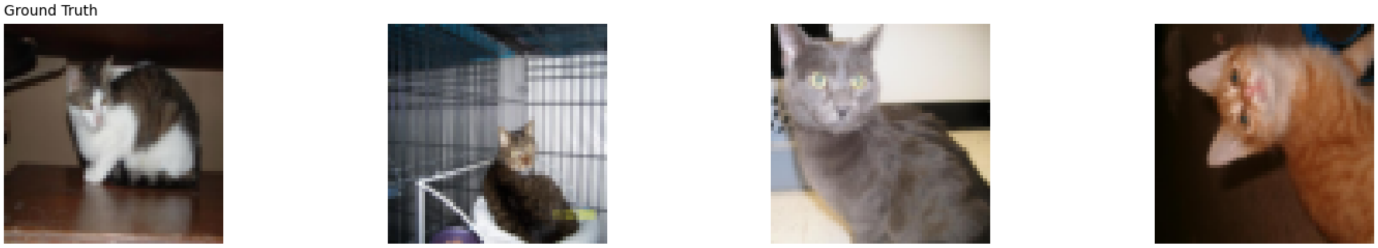


Generated Output

Inpainted



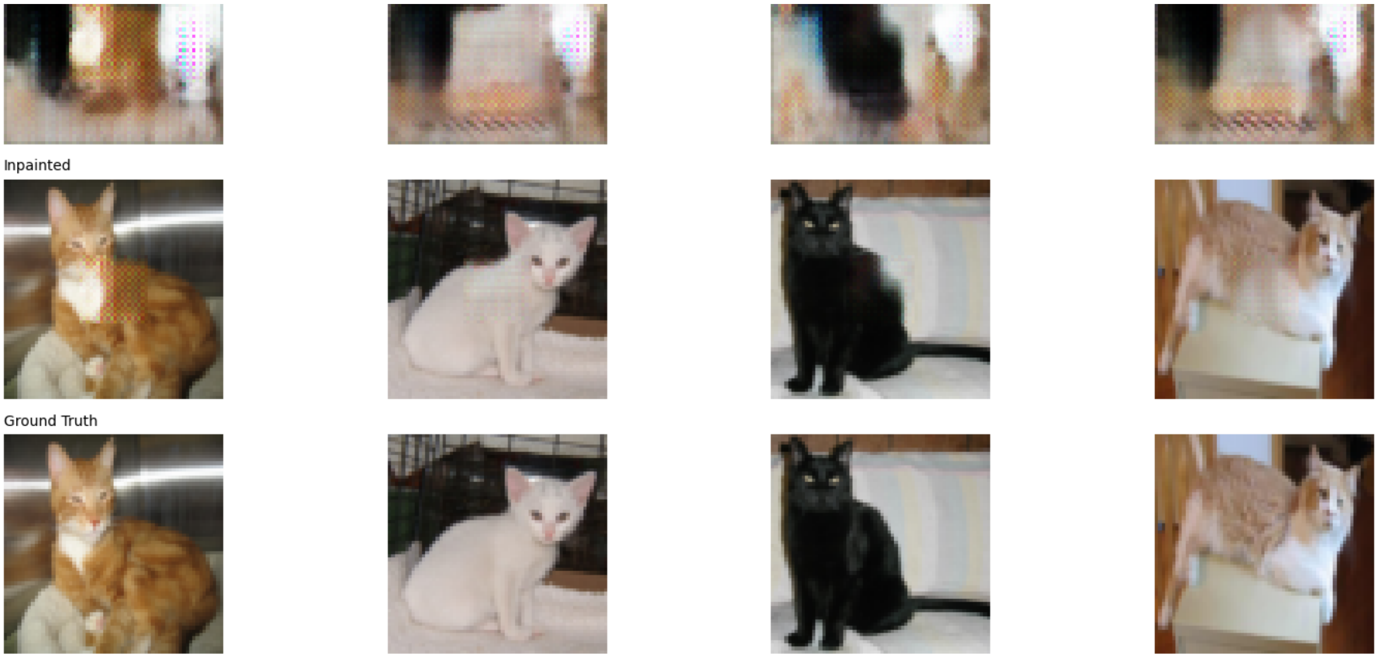
Ground Truth



Epoch [49/50], Step [1174/1174], Discriminator Loss: 0.4979, Generator Loss: 3

Masked Input

Generated Output

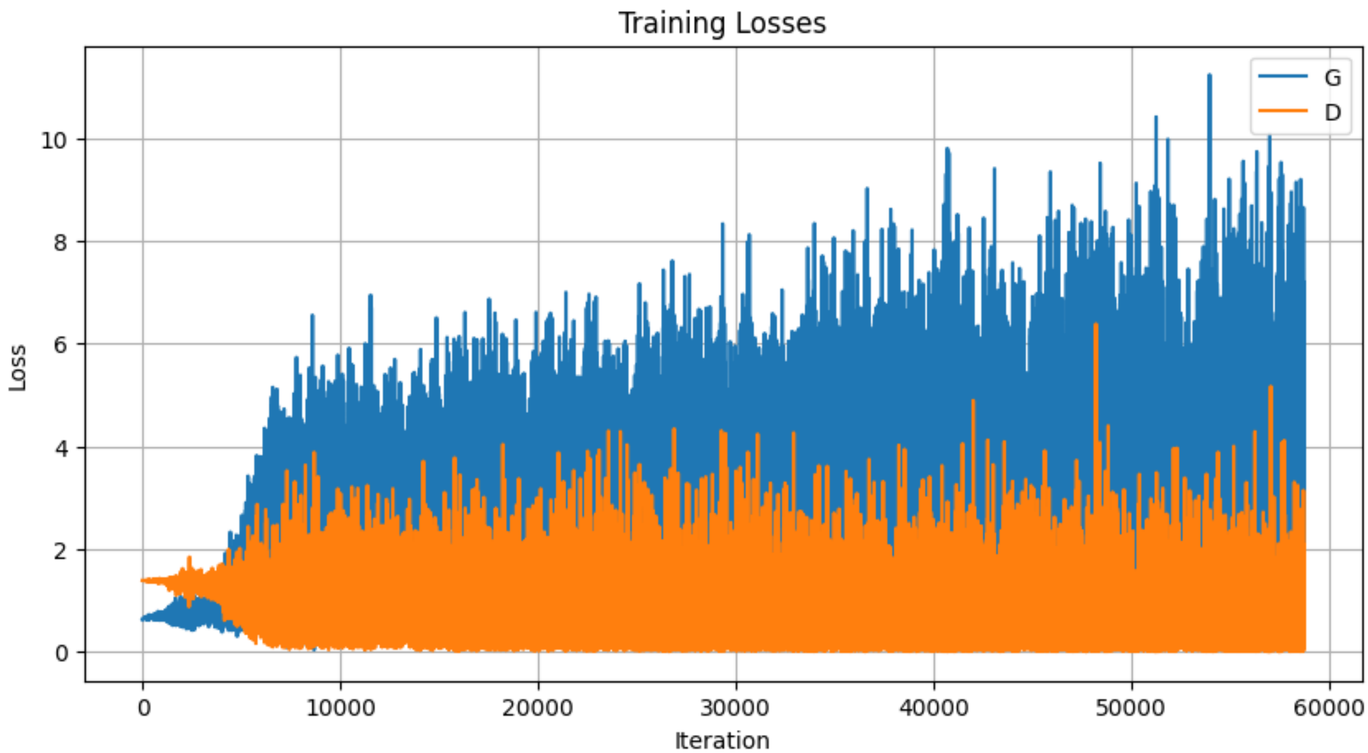


Epoch [50/50], Step [1174/1174], Discriminator Loss: 0.4898, Generator Loss: 3

4.4 Training results

We find that the model's inpainting quality greatly improves with training. Reconstructing background elements is difficult in the beginning, frequently resulting in mismatched colors and shading, but these features improve significantly in later eras. Simpler cat body parts like the belly and legs are generated by the model fairly well, but it constantly fails to produce more complex facial features like the eyes, nose, and mouth—though it can occasionally produce eyes. One significant drawback is its inability to handle human features, like hands, which is still a known problem in generative modeling. All things considered, the picture clearly demonstrates the model's growing comprehension of texture and structure over time.

```
# Plot the loss graph
plt.figure(figsize=(10, 5))
plt.plot(g_losses, label="G")
plt.plot(d_losses, label="D")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Training Losses")
plt.legend()
plt.grid(True)
plt.show()
```



Throughout training, this graph displays the discriminator's and generator's loss values. On average, the discriminator loss shows a distinct downward trend; the lower boundary of its oscillating curve gradually drops, taking the shape of a cone that gets smaller over time. On the other hand, the generator's baseline only slightly drifts downward, and its loss stays largely constant. As training goes on, both loss curves exhibit an intriguing pattern of rising volatility with larger value fluctuations. The source of this instability is unknown, but it doesn't seem to have a detrimental effect on the generated inpainted images' visual quality, indicating that average loss behavior is still trending in a productive direction.

One thing to note is that the discriminator consistently has a lower loss than the generator. This makes sense considering the generator's job is "harder" than the discriminators in the beginning. During training, the discriminator has a set of ground truth images and fake images (generated by the generator) making the training and gradient calculation much more straightforward, while the generator has to generate images from scratch and its gradients depends on the discriminator's, which can make it much more unstable. Furthermore, the generator is very bad at generating "real" images in the beginning, so it is very easy for the discriminator to tell which ones are real and which ones are fake, making the discrimnator's loss significantly lower than the generator's. However, as training goes on, we can expect the two values to converge eventually, but since our epochs are fairly small, we can;t see the convergence in the graph.

```
torch.save(g.state_dict(), f"initial_generator.pt")
torch.save(d.state_dict(), f"initial_discriminator.pt")
```

5. Model Performance on Test Set

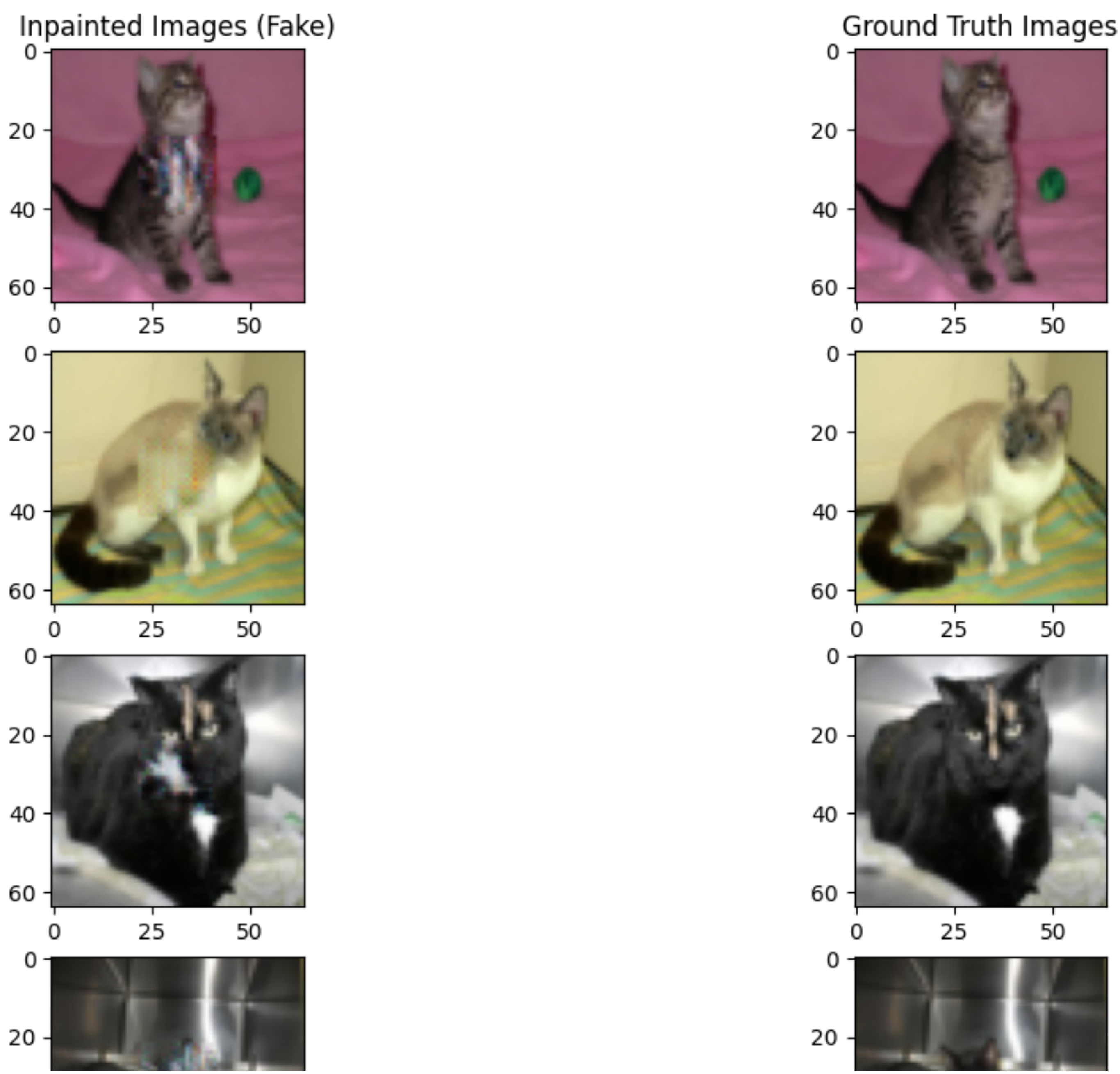
Testing provides a final, unbiased evaluation of our GAN’s performance on completely unseen data. After training and validation, the test set serves as a benchmark to assess how well the generator can inpaint masked images and how effectively the discriminator can distinguish real from generated content. This step confirms whether the model has truly generalized and produces high-quality, realistic outputs beyond the data it was exposed to during training.

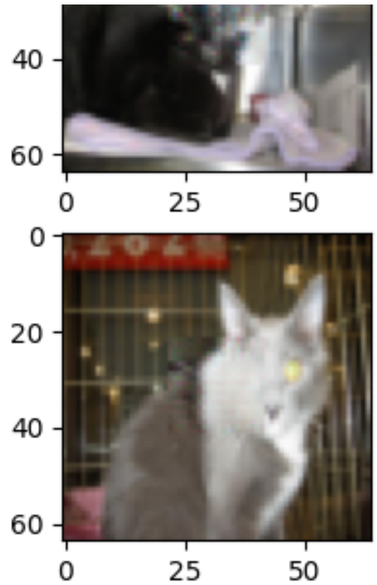
```
# Load pre-trained models
g = Generator(im_channels=3).to(device)
d = Discriminator().to(device)

g.load_state_dict(torch.load("initial_generator.pt", map_location=device))
d.load_state_dict(torch.load("initial_discriminator.pt", map_location=device))
```

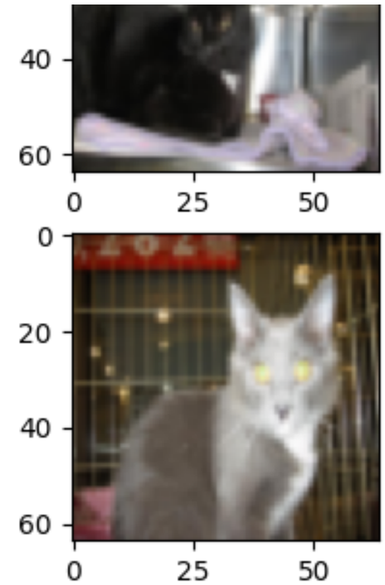
```
<All keys matched successfully>
```

```
# Shows results after training
num_evaluation = 10
num_image_display = 5
val_iter = iter(val_loader)

# Loss
g_tot_loss = 0
d_tot_loss = 0
num_total = 1

# Test GAN
f, ax = plt.subplots(num_image_display, 2, figsize=(12, 12))
ax[0, 0].set_title("Inpainted Images (Fake)")
ax[0, 1].set_title("Ground Truth Images")
for i in range(num_evaluation):
  g.eval()
  d.eval()
  with torch.no_grad():
      val_batch = next(val_iter)
      val_masked, val_truth = val_batch
      N = len(val_masked)

      # Move Test data to GPU
      val_masked = val_masked.to(device)
      val_truth = val_truth.to(device)
      N = len(val_masked)

      # Get labels
      true_labels = torch.ones(N).to(device)
      fake_labels = torch.zeros(N).to(device)

      val_masks = create_masks(N).to(device)
      val_masked = torch.cat((val_masked, val_masks), dim=1)

      # Get the inpainted image using the generator
      fake_im = g(val_masked)
      inpainted = fake_im * val_masks[0] + val_masked[:, :3, :, :]
```

```
      outputs = d(inpainted).view(-1)

      d_tot_loss += criterion(outputs, fake_labels).item()
      g_tot_loss += criterion(outputs, true_labels).item()
      num_total += 1

      if (i < num_image_display):
        # Display first sample
        ax[i, 0].imshow(inpainted[0].cpu().permute(1, 2, 0))
        ax[i, 1].imshow(val_truth[0].cpu().permute(1, 2, 0))

plt.show()
print(f'Number of Test Images: {num_total * N}, '
            f'Average Discriminator Loss: {d_tot_loss / num_total:.4f}, '
            f'Average Generator Loss: {g_tot_loss / num_total:.4f}')
```








```
Number of Validation Images: 88, Average Discriminator Loss: 0.0366, Average G
```

5.1 Results

We evaluated our GAN model on a test set of 88 images, comparing the generator's inpainted outputs to the ground truth images. The average discriminator loss was 0.0366, and the average generator loss was 3.7808.

A low discriminator loss indicates that the discriminator is performing well—it's confidently distinguishing between real and generated images. A higher generator loss suggests that the generator still struggles to consistently fool the discriminator, which could reflect gaps in fine detail, texture, or global coherence. Looking at the image pairs, the inpainted images capture the general structure and content well, with correct object shapes and positions. However, there is still noticeable blurring and artifacts, especially in areas requiring high-frequency details like fur texture or lighting transitions. This is consistent with the relatively high generator loss and shows room for improvement in generating finer, more realistic textures.

Overall, the model generalizes reasonably well to unseen data, producing plausible completions, though further tuning—like loss function adjustments, better masking strategies, or discriminator enhancement—could help push the realism further.

6. Experiments

The dynamic interaction between a generator and a discriminator is captured by the rich mathematical framework that underpins Generative Adversarial Networks (GANs). The mathematical viewpoint presented in this blog, which offers a formal and intuitive understanding of the adversarial training setup, served as our source of inspiration. The objective function formalizes the min-max optimization problem at the core of this framework:

$$V(G, D) = \mathbb{E}_{x \sim p_{\text{data}}}[\log(D(x))] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))]$$

This formulation supports our intuition: the generator aims to learn the true data distribution by producing outputs that the discriminator cannot distinguish from real data. To support this goal, we explored several loss functions beyond the adversarial objective—specifically L1, L2, and Huber loss. L1 loss (mean absolute error) is more robust to outliers and encourages sparsity, while L2 loss (mean squared error) penalizes larger errors more heavily, leading to smoother predictions. Huber loss offers a balance between the two, behaving like L2 for small errors and L1 for large ones. By experimenting with these loss functions, we aimed to better guide the generator during training and enhance the quality of inpainted outputs.

6.1 L1 Loss (Mean Absolute Error)

Measures the average absolute difference between predicted and target pixels.

- **Pros:** More robust to outliers; leads to sharper reconstructions.
- **Cons:** Gradient is constant (non-smooth), which can make optimization noisy.

```
l1_g = Generator(im_channels=3).to(device)
l1_d = Discriminator().to(device)

l1_optimizer_g = optim.Adam(l1_g.parameters(), lr=0.0001)
l1_optimizer_d = optim.Adam(l1_d.parameters(), lr=0.0001)

l1_criterion = nn.L1Loss(reduction="mean")
num_epochs = 10

l1_g_losses, l1_d_losses = train(l1_g, l1_d, train_loader, val_loader, l1_criteri
```

```
Epoch [1/10], Step [1174/1174], Discriminator Loss: 0.9982, Generator Loss: 0.
Epoch [2/10], Step [1174/1174], Discriminator Loss: 0.9945, Generator Loss: 0.
Epoch [3/10], Step [1174/1174], Discriminator Loss: 0.9915, Generator Loss: 0.
Epoch [4/10], Step [1174/1174], Discriminator Loss: 0.9940, Generator Loss: 0.
Epoch [5/10], Step [1174/1174], Discriminator Loss: 0.9947, Generator Loss: 0.
Epoch [6/10], Step [1174/1174], Discriminator Loss: 0.9948, Generator Loss: 0.
Epoch [7/10], Step [1174/1174], Discriminator Loss: 0.9990, Generator Loss: 0.
Epoch [8/10], Step [1174/1174], Discriminator Loss: 0.9969, Generator Loss: 0.
Epoch [9/10], Step [1174/1174], Discriminator Loss: 0.9901, Generator Loss: 0.
Epoch [10/10], Step [1174/1174], Discriminator Loss: 0.9916, Generator Loss: 0
```

6.2 L2 (MSE) Loss

Measures the average squared difference between predicted and actual pixels.

- **Pros:** Smooth gradient helps stabilize training.
- **Cons:** Very sensitive to outliers; can produce blurrier outputs.

```
l2_g = Generator(im_channels=3).to(device)
l2_d = Discriminator().to(device)

l2_optimizer_g = optim.Adam(l2_g.parameters(), lr=0.0001)
l2_optimizer_d = optim.Adam(l2_d.parameters(), lr=0.0001)

l2_criterion = nn.MSELoss(reduction="mean")
num_epochs = 10

l2_g_losses, l2_d_losses = train(l2_g, l2_d, train_loader, val_loader, l2_criteri
```

```
Epoch [1/10], Step [1174/1174], Discriminator Loss: 0.4940, Generator Loss: 0.
Epoch [2/10], Step [1174/1174], Discriminator Loss: 0.4849, Generator Loss: 0.
Epoch [3/10], Step [1174/1174], Discriminator Loss: 0.4894, Generator Loss: 0.
Epoch [4/10], Step [1174/1174], Discriminator Loss: 0.4796, Generator Loss: 0.
Epoch [5/10], Step [1174/1174], Discriminator Loss: 0.4420, Generator Loss: 0.
Epoch [6/10], Step [1174/1174], Discriminator Loss: 0.3774, Generator Loss: 0.
Epoch [7/10], Step [1174/1174], Discriminator Loss: 0.3424, Generator Loss: 0.
Epoch [8/10], Step [1174/1174], Discriminator Loss: 0.3160, Generator Loss: 0.
Epoch [9/10], Step [1174/1174], Discriminator Loss: 0.3145, Generator Loss: 0.
Epoch [10/10], Step [1174/1174], Discriminator Loss: 0.2931, Generator Loss: 0
```

6.3 Huber loss

Combines L1 and L2: uses L2 for small errors and L1 for large ones.

- **Pros:** Balanced — robust like L1, smooth like L2.
- **Cons:** Requires a delta threshold, which can be a tuning parameter.

```
huber_g = Generator(im_channels=3).to(device)
huber_d = Discriminator().to(device)

huber_optimizer_g = optim.Adam(huber_g.parameters(), lr=0.0001)
huber_optimizer_d = optim.Adam(huber_d.parameters(), lr=0.0001)

huber_criterion = nn.HuberLoss(reduction="mean")
num_epochs = 10

huber_g_losses, huber_d_losses = train(huber_g, huber_d, train_loader, val_loader
```

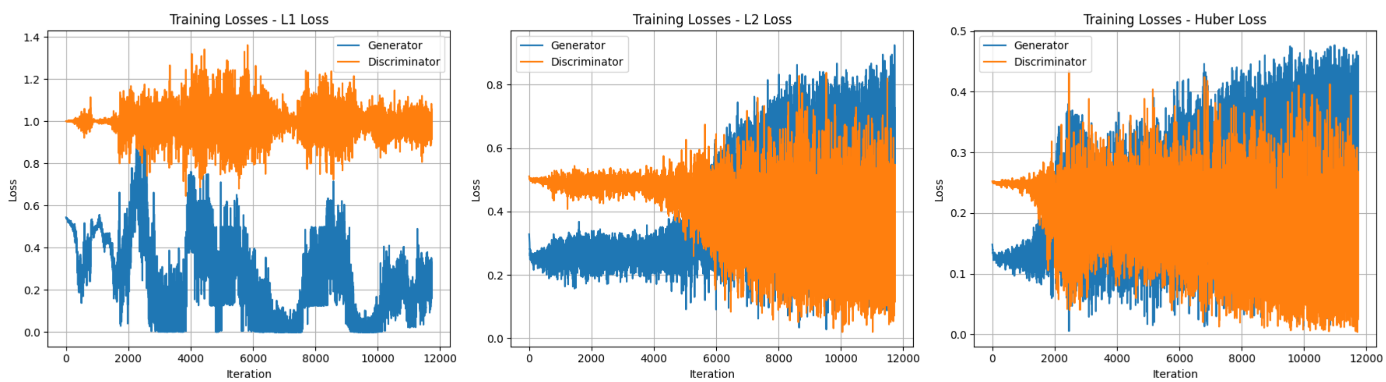
```
Epoch [1/10], Step [1174/1174], Discriminator Loss: 0.2479, Generator Loss: 0.
Epoch [2/10], Step [1174/1174], Discriminator Loss: 0.2132, Generator Loss: 0.
Epoch [3/10], Step [1174/1174], Discriminator Loss: 0.1883, Generator Loss: 0.
Epoch [4/10], Step [1174/1174], Discriminator Loss: 0.2050, Generator Loss: 0.
Epoch [5/10], Step [1174/1174], Discriminator Loss: 0.1929, Generator Loss: 0.
Epoch [6/10], Step [1174/1174], Discriminator Loss: 0.1768, Generator Loss: 0.
Epoch [7/10], Step [1174/1174], Discriminator Loss: 0.1575, Generator Loss: 0.
Epoch [8/10], Step [1174/1174], Discriminator Loss: 0.1488, Generator Loss: 0.
Epoch [9/10], Step [1174/1174], Discriminator Loss: 0.1374, Generator Loss: 0.
Epoch [10/10], Step [1174/1174], Discriminator Loss: 0.1336, Generator Loss: (
```

```
# Save the models
torch.save(l1_g.state_dict(), 'l1_generator.pt')
torch.save(l1_d.state_dict(), 'l1_discriminator.pt')
torch.save(l2_g.state_dict(), 'l2_generator.pt')
torch.save(l2_d.state_dict(), 'l2_discriminator.pt')
torch.save(huber_g.state_dict(), 'huber_generator.pt')
torch.save(huber_d.state_dict(), 'huber_discriminator.pt')
```

6.4 Plot losses

We plot the L1, L2 and Huber losses to compare their performance over 10 epochs

```
loss_sets = [
    ("L1 Loss", l1_g_losses, l1_d_losses),
    ("L2 Loss", l2_g_losses, l2_d_losses),
    ("Huber Loss", huber_g_losses, huber_d_losses)
]

# Create subplots
fig, axs = plt.subplots(1, 3, figsize=(18, 5))

for i, (title, g_losses, d_losses) in enumerate(loss_sets):
    axs[i].plot(g_losses, label="Generator")
    axs[i].plot(d_losses, label="Discriminator")
    axs[i].set_title(f"Training Losses - {title}")
    axs[i].set_xlabel("Iteration")
    axs[i].set_ylabel("Loss")
    axs[i].legend()
    axs[i].grid(True)

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



L1 Loss

- Generator's loss (blue) shows high-frequency oscillations — typical with L1 since the gradient doesn't change magnitude.
- Discriminator's loss (orange) is relatively stable but stays near 1, possibly indicating it's winning more often (i.e., catching fakes).
- **Overall:** training looks noisy, which is common for adversarial setups using L1 alone.

L2 Loss

- Generator loss increases steadily over time, which may suggest it's struggling to fool the discriminator or outputs may be too smooth or blurry (common with L2).
- Discriminator loss drops, implying it's doing a better job separating fake from real.
- **Overall:** the training looks more stable than L1, but maybe at the cost of image quality.

Huber Loss

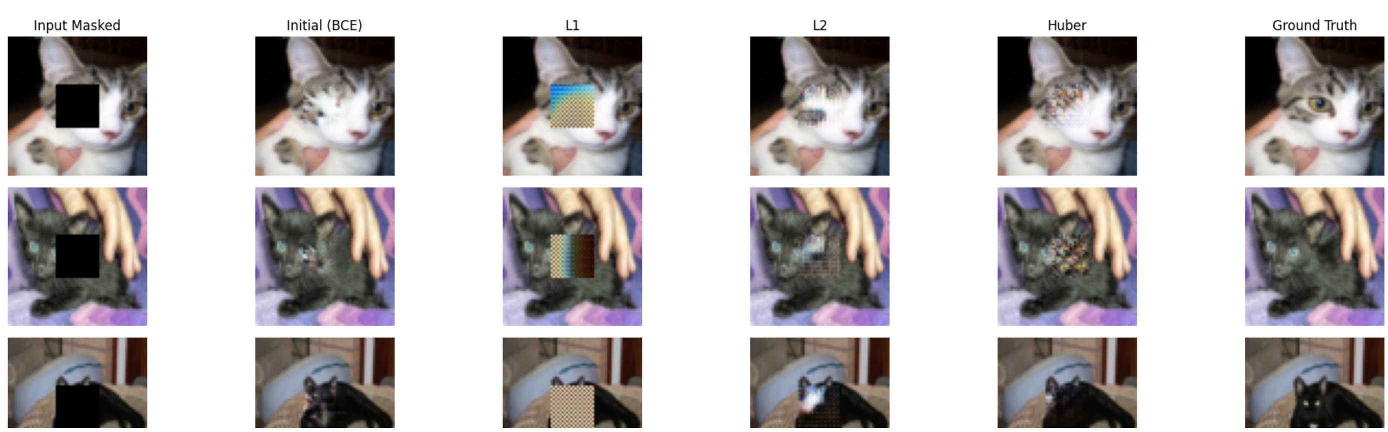
- Generator loss increases but remains smoother than L1 — this suggests a stable training process with good gradient behavior.
- Discriminator loss decreases, but not drastically, meaning it's challenged — a good sign that both networks are learning.
- **Overall:** best balance between realism and stability among the three.

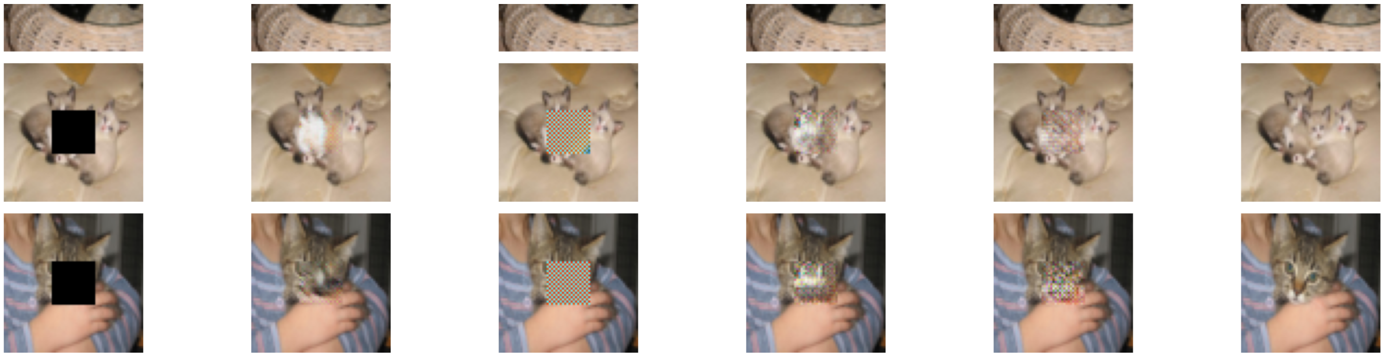
7. Comparison of Models

7.1 Plot Model Results

```
models = {
    "Initial (BCE)": (g, d),
    "L1": (l1_g, l1_d),
    "L2": (l2_g, l2_d),
    "Huber": (huber_g, huber_d)
}

val_iter = iter(val_loader)
num_images_to_display = 5

fig, axes = plt.subplots(num_images_to_display, len(models) + 2, figsize=(20, 10)
```

```
for i in range(num_images_to_display):
    val_batch = next(val_iter)
    val_masked, val_truth = val_batch
    N = len(val_masked)

    # Move data to GPU
    val_masked = val_masked.to(device)
    val_truth = val_truth.to(device)
    val_masks = create_masks(N).to(device)
    val_input = torch.cat((val_masked, val_masks), dim=1)

    #Input Masked Image
    axes[i, 0].imshow(val_masked[0].cpu().permute(1, 2, 0))
    axes[i, 0].axis('off')
    if i == 0:
        axes[i, 0].set_title("Input Masked")

    col = 1
    for model_name, (gen_model, _) in models.items():
        with torch.no_grad():
          gen_model.eval()
          fake_im = gen_model(val_input)
          inpainted = fake_im * val_masks[0] + val_input[:, :3, :, :]
          axes[i, col].imshow(inpainted[0].cpu().permute(1, 2, 0))
          axes[i, col].axis('off')
          if i==0:
              axes[i,col].set_title(model_name)
          col+=1

    axes[i, col].imshow(val_truth[0].cpu().permute(1, 2, 0))
    axes[i, col].axis('off')
    if i == 0:
        axes[i, col].set_title("Ground Truth")

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





7.2 Observations

Initial (BCE):

The BCE-trained generator does a pretty good job to restore structure, but many completions appear slightly blurry or distorted. There's some inconsistency in texture, particularly in fur and facial regions. But overall everything looks quite natural and similar to ground truth image.

L1 Loss:

What we notice, (which is expected for the 10 epochs we trained on), is that the output often has unnatural checkerboard artifacts—a known issue in L1-driven reconstructions, possibly due to over-penalizing absolute pixel differences.

L2 Loss:

L2 tends to smooth out noise and artifacts better, and converges faster to a more natural fitting image than our L1 loss over 10 epochs. Overall, even with the limited epochs, it does a pretty good job at producing a good looking result. However, while with fewer epochs than L1, L2 loss may in the long run result in cleaner but bland and overly-smoothed outputs. Fine details, especially around eyes and fur, are often lost or blurred, reducing realism.

Huber Loss:

Huber strikes a balance between L1 and L2, it converges faster than L1, but still has more of the checkerboard effect than L2. It preserves structure while being robust to outliers, which helps maintain sharpness without severe artifacts.

8. Conclusions

Our DCGAN model demonstrated promising results in the object removal and inpainting task. It was able to generate visually plausible completions of the missing regions, especially for relatively uniform backgrounds like fur. However, the model still has clear room for improvement. Recent advances such as Stable Diffusion and other Transformer-based architectures have shown significant gains in generative tasks. Incorporating these state-of-the-art models, though computationally expensive, could potentially lead to more coherent and higher-quality results.

At the start of our project, we simplified the problem by always masking a fixed square at the center of each image. This made the task more controlled, but less practical for real-world applications. A more flexible approach would involve user-defined masks of arbitrary shape and size. Supporting custom masks would make the model far more robust and widely usable across different tasks. Future work could involve incorporating spatial attention or segmentation cues to guide such inpainting.

Training the model involved experimenting with several loss functions, including L1, L2, and Huber loss, which are commonly used for regression tasks. However, these losses did not yield stable or visually pleasing outputs in our setting. We eventually settled on binary cross-entropy (BCE) loss due to its alignment with the adversarial training objective and improved qualitative results. Still, finding the right balance between adversarial loss and reconstruction loss remains a challenge. Future exploration into hybrid loss strategies or perceptual loss may help improve output fidelity.

One significant challenge we encountered was the inconsistency in our dataset of cat images. Many images were not standardized—cats varied in position, lighting, and scale, and some images even included human hands or other distractions. These inconsistencies made it harder for the model to learn a clean background distribution. Moving forward, it may be beneficial to either cleanse the current dataset or switch to a more standardized and curated one. Starting with an easier dataset may also allow the model to learn more effectively before tackling more complex scenarios.

Another challenge was encountered was a problem with our training method that caused our GAN to not work as intended. Before, we would just pass the generated image straight into the discriminator. This caused our GAN to learn how to generate cat pictures from scratch, rather than filling in the "hole", as seen in the image below.

From this picture, we can see that the generated pictures vaguely resemble cats and don't really fit into the image once we overlay it with the actual image. To fix this issue, we would pass the "inpainted" image into the discriminator. We get this "inpainted" picture by adding the generated patch of the image into the hole of the original image. This allowed the GAN to focus on generating the missing portions of the images, rather than generating new pictures as a whole, and our GAN started working as intended.

For next steps, we can investigate the effects of having two discriminators where one is a local discriminator that analyzes the inpainted (generated) part of the image and a global discriminator that evaluates the image as a whole (as mentioned in S. Iizuka, E. Simo-Serra, H. Ishikawa's paper). The main idea behind this is that the local discriminator will evaluate the details of the inpainting, while the global discriminator evaluates the overall coherence and structure of the image. This will "encourage" the generator to create more detailed inpaintings while still preserving the overall structure of the image.