Metody Rozpoznawania Obrazów Dominik Szot Zadanie 03

Tematem laboratorium jest zbudowanie architektury składającej się z dwóch, wzajemnie ze sobą rywalizujących, sieci neuronowych, gdzie pierwsza z nich będzie generować możliwie realistyczne obrazy pasujące do pewnego zbioru, druga - odróżniać tak stworzone podróbki od rzeczywistych fotografii.

Model Dyskryminatora

Dyskryminator przyjmuje na wejście fotografie w rozmiarze **32x32**. Zgodnie z poleceniem warstwy konwolucyjne wykorzystują filtr o rozmiarze 4x4 ze **stride = 2** oraz **paddingiem = 1**, który pozwala osiągnąć podobny efekt w PyTorch'u, co padding = `same` używany w innych frameworkach. Dodatkowo dodałem normalizację spektralną, która ponoć (!) poprawia uczenie się sieci (bardziej stabilne uczenie). Poza tym pojawia się również standardowa Batch Normalization, a jako funkcję aktywacji zastosowano Leaky ReLU.
.

```
class Discriminator(nn.Module):
  def __init__(self, input_channels=3, image_size=32, dropout=0.2):
    super().__init__()
    channels = [input_channels, 32, 64, 64]
    self.layers = nn.Sequential(*[self._layer(channels[i-1],
       channels[i]) for i in range(1, 4)])
    self.dropout = nn.Dropout(p=dropout)

    self.fc = nn.Linear(
      in_features=64 * 4 * 4,
      out_features=1
    )

  def _layer(self, in_channels, out_channels):
    return nn.Sequential(
      spectral_norm(
        nn.Conv2d(
          in_channels=in_channels,
          out_channels=out_channels,
          kernel_size=4,
          stride=2,
          padding=1
        )),
      nn.BatchNorm2d(num_features=out_channels),
      nn.LeakyReLU(negative_slope=0.2)
    )

  def forward(self, x):
    x = self.layers(x)
    x = torch.flatten(x, 1)
    x = self.dropout(x)
    x = self.fc(x)
    return  torch.sigmoid(x)
```

Model Generatora

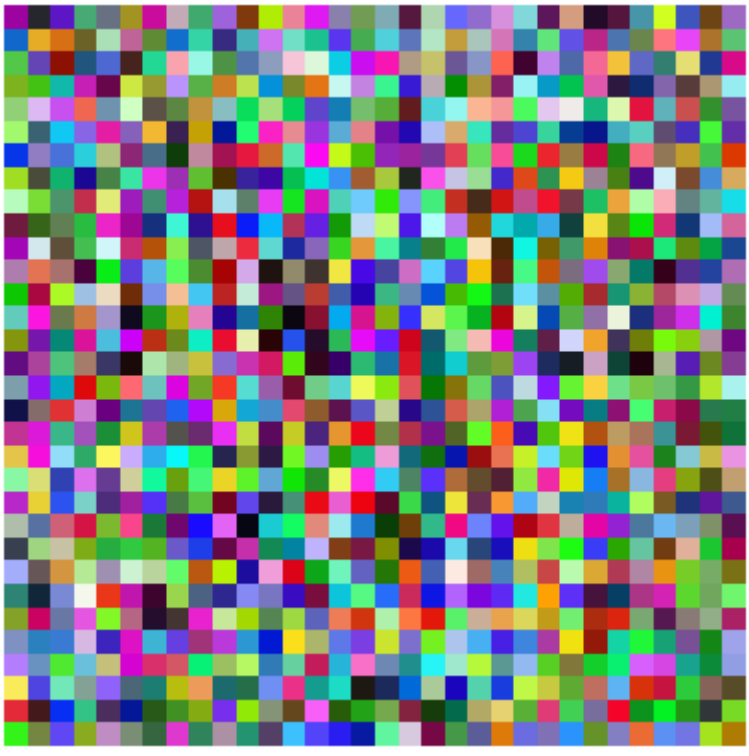
Generator zgodnie z treścią polecenia zaczyna się od wektora składającego się z 64 wartości. Dodajemy warstwę gęsto połączoną o rozmiarze wyjść równym 4x4x64. Układamy wartości w nowe kształty za pomocą unflatten. Warstwy konwolucji transponowanej użyto do zwiększenia rozdzielczości obrazu. Na końcu dodano wartstwę zmniejszającą liczbę kanałów wejściowych z 256 na 3 kanały, odpowiadające kanałom RGB.

```
class Generator(nn.Module):
  def __init__(self, latent_dim=64):
    super().__init__()

    self.latent_dim = latent_dim
    self.channels = [latent_dim, 64, 128, 256]
    self.fc = nn.Linear(latent_dim, 4*4*64)
    self.reshape = nn.Unflatten(1, torch.Size([64, 4, 4]))

    layers = []
    for in_ch, out_ch in zip(self.channels[:-1], self.channels[1:]):
      layers.extend([
        nn.ConvTranspose2d(
          in_channels=in_ch,
          out_channels=out_ch,
          kernel_size=4,
          stride=2,
          padding=1,
          bias=False
        ),
        nn.BatchNorm2d(out_ch),
        nn.LeakyReLU(negative_slope=0.2, inplace=True)
      ])

    self.layers = nn.Sequential(*layers)

    self.conv = nn.Conv2d(
      in_channels=256,
      out_channels=3,  # RGB output
      kernel_size=5,
      stride=1,
      padding='same'
    )

    self.act = nn.Tanh()

  def forward(self, x):
    if x.size(1) != self.latent_dim:
      raise ValueError(f"Expected input size (*, {self.latent_dim}), got {x.shape}")

    x = self.fc(x)
    x = self.reshape(x)
    x = self.layers(x)
    x = self.conv(x)
    return self.act(x)
```

Możemy teraz sprawdzić co wypluje nam generator który przyjmie na wejściu losowy szum. Torch.rand() generuje liczby z **rozkładu normalnego o średniej 0 i odchyleniu standardowym 1**

```
from torchvision.transforms.functional import to_pil_image

def show_image(image, cmap='gray', ax=None):
  if ax:
    ax.imshow(image, cmap=cmap)
    ax.set_axis_off()
  else:
    plt.imshow(image, cmap=cmap)
    plt.axis('off')
    plt.show()

def show(tensor: torch.Tensor, cmap='gray', ax =None):
  show_image(to_pil_image(tensor), cmap, ax=ax)


SAME_NOISE = torch.randn(1,64).to(device)
model = Generator().to(device)

with torch.no_grad():
  generated_image = model(SAME_NOISE)

generated_image = generated_image.squeeze(0).cpu().numpy().transpose(1, 2, 0)
show(generated_image)
```

Wynik, który jest zwracany przez generator, wygląda jak wygląda; trudno go do czegokolwiek porównać



Obraz 1: Wynik działania generatora z losowym szumem

Zbiór Danych

W tym kroku przygotujemy zbiór danych, który będzie nam służyć do generowania nowych obrazków. Nie komplikując sobie życia zbyt bardzo, importujemy zbiór przygotowany i udostępniony przez prowadzącego. Zbiór jest skalowany do rozmiaru 32x32, a kanały normalizowane do wartości [-1, 1].

```
class SimpleImagesDataset(Dataset):
  def __init__(self, img_dir, transform=None, target_transform=None):
    self.img_labels = os.listdir(img_dir)
    self.img_dir = img_dir
    self.transform = transform
    self.target_transform = target_transform

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

  def __getitem__(self, idx):
    img_path = os.path.join(self.img_dir, self.img_labels[idx])
    image = Image.open(img_path).convert("RGB")
    if self.transform:
      image = self.transform(image)
    label = 1
    if self.target_transform:
      label = self.target_transform(label)

    return image, label

from torchvision import transforms

transform = transforms.Compose([
  transforms.Resize((32, 32)),
  transforms.ToTensor(),
  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

dataset = SimpleImagesDataset("crawled_cakes", transform=transform)
dataloader = DataLoader(dataset, batch_size=9, shuffle=True)
```

Możemy teraz sprawdzić co tak naprawdę znajduje się w naszym zbiorze danych. Do tego będę używać bardzo prostej funkcji.

```
def imshow(img):
  npimg = img.numpy()
  plt.figure(figsize=(15, 15))
  plt.imshow(np.transpose(npimg, (1, 2, 0)))
  plt.axis("off")
  plt.show()

def grid_show(images, nrow=2):
  imshow(torchvision.utils.make_grid(images, nrow=nrow))
```

Obrazki rzeczywiście wyglądają jak jakiegoś rodzaju ciasta (nie mnie to oceniać).

Obraz 2: Przykładowe obrazy znajdujące się z zbiorze danych

Prosty Rozruch

W ramach tego punktu zaimplementowano prosty model składający się z 2 warstw gęsto połączonych, prostą funkcją loss i tak dalej. Głównym problemem jest korygowanie wag używając pochodnych.

```
class SimpleModel(nn.Module):
  def __init__(self):
    super(SimpleModel, self).__init__()
    self.layer1 = nn.Linear(3, 3)
    self.layer2 = nn.Linear(3, 3)

  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    return x

model = SimpleModel()
```

```
for epoch in range(10):
  pred = model(x)

  loss = (pred.mean() - 42) ** 2  # Pseudo-loss function
  print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")

  loss.backward()

  with torch.no_grad():
    model.layer1.weight.data -= 0.01 * model.layer1.weight.grad

  model.layer1.weight.grad.zero_()
```

Odpalając kilka iteracji wartości funkcji kosztu rzeczywiście spadają – zakładamy więc że o to chodziło.

Trening Dyskryminatora

Trening dyskryminatora jest prawie tym, czego oczekiwano w poleceniu (prawie!). Brak zaciemniania wektora etykiet (student zapomniał). Paczki batchy zarówno losowego szumu (16 elementów), jak i ze zbioru danych (16 elementów) nie są stricte łączone, ale przetwarzane osobno, a łączone są jedynie wyniki funkcji kosztu — binary cross entropy (mam nadzieję, że na jedno wychodzi). Learning rate jest ustawiony zgodnie z wymogami na 0.00001. Użyty optymizator to Adam. fake_imgs.detach() odpowiada teoretycznie za odłączanie gradientów dla generatora, przez co wagi podczas treningu się nie zmieniają

```
    optimizer_discriminator.zero_grad()

    outputs_real = discriminator(real_imgs)             # Real images
    loss_real = criterion(outputs_real, real_labels)

    noise = torch.randn(current_batch_size, 64).to(device)  # Generate fake images
    fake_imgs = generator(noise)

    outputs_fake = discriminator(fake_imgs.detach())    # Train with fake images
    loss_fake = criterion(outputs_fake, fake_labels)

    dis_loss = (loss_real + loss_fake) / 2   # Combined discriminator loss
    dis_loss.backward()
    optimizer_discriminator.step()
```

Trening Generatora

No więc jeśli chodzi o trening generatora to wygląda on tak:

- Posiadamy już wygenerowany batch fałszywych obrazów (fake_imgs)
- Zamiast fake_labels podajemy odwrotne etykiety, real_labels
- Wagi się zmieniają dla generatora (dyskryminator pozostaje bez zmian)

```
optimizer_generator.zero_grad()

outputs_fake = discriminator(fake_imgs)
gen_loss = criterion(outputs_fake, real_labels)
gen_loss.backward()
optimizer_generator.step()

running_loss_gen += gen_loss.item()
```

Przygotowanie się do treningu

Głównym problemem jaki napotkamy obecnie to ilość czasu potrzebna do wytrenowania naszego modelu. Ponieważ model może się przerwać w różnych okolicznościach implementujemy funkcjonalność pozwalającą zapisać stan modelu co 50 epok

```
[pętla treningowa… - trening dyskryminatora i generatora]
if epoch % 50 == 0:
    save_model_state(epoch, generator, discriminator,
           optimizer_generator, optimizer_discriminator, save_path)
    save_generated_images(generator, fixed_noise, save_path, epoch)
```

```
def save_model_state(epoch, generator, discriminator, optimizer_generator,
optimizer_discriminator, save_path):
  os.makedirs(save_path, exist_ok=True)
  torch.save({
    'epoch': epoch,
    'generator_state_dict': generator.state_dict(),
    'discriminator_state_dict': discriminator.state_dict(),
    'optimizer_generator_state_dict': optimizer_generator.state_dict(),
    'optimizer_discriminator_state_dict': optimizer_discriminator.state_dict(),
  }, os.path.join(save_path, f"checkpoint_epoch_{epoch}.pth"))
```

W ten sposób będziemy w stanie odzyskać najnowszy stan modelu, a następnie kontynuować dalszy trening modelu, bez obawy, że w razie problemów cały nasz trening pójdzie do kosza.

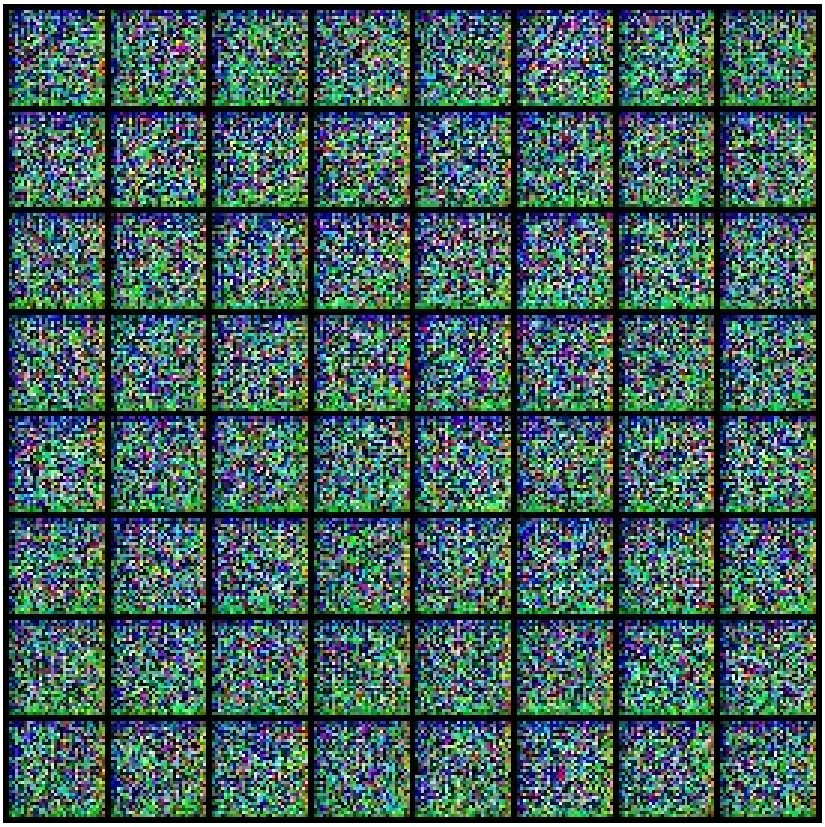
Jeśli chodzi o monitorowanie efektów generacji – będzie to wykonane przez zapisywanie zdjęć generowanych przez generator co 50 epok, podobnie do zapisu najnowszego stanu modelu.

```
dimport torchvision.utils as vutils

def save_image(tensor, filename, nrow=8, padding=2):
  os.makedirs(os.path.dirname(filename), exist_ok=True)
  vutils.save_image(tensor, filename, nrow=nrow, padding=padding)

def save_generated_images(generator, fixed_noise, save_path, epoch):
  os.makedirs(save_path, exist_ok=True)
  with torch.no_grad():
    generated_images = generator(fixed_noise)
    save_image(generated_images, os.path.join(save_path,
f"generated_epoch_{epoch}.png"))
```

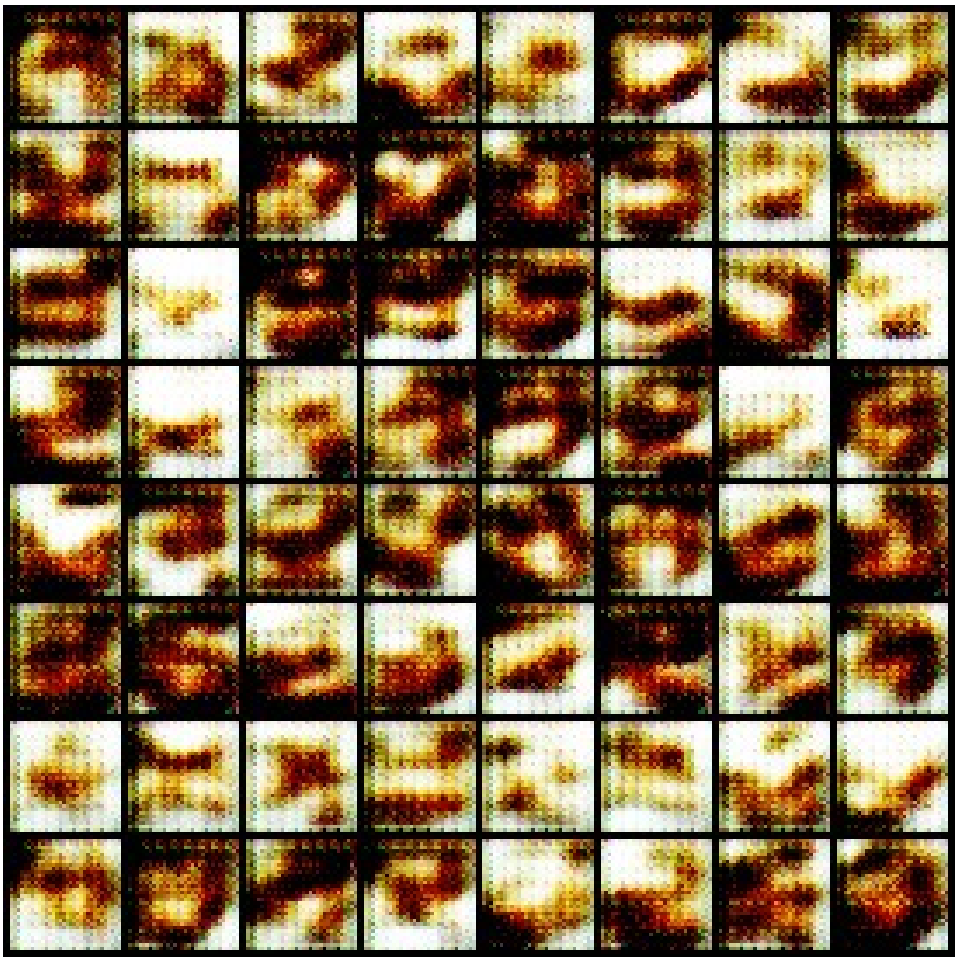
Jeśli chodzi o cykliczne zapisywanie informacji o tym, ile w danej epoce wynosi średni loss dla generatora i dyskryminatora – jest to robione na oko – wypisując po każdej epoce aktualny loss dla obu.

Trening

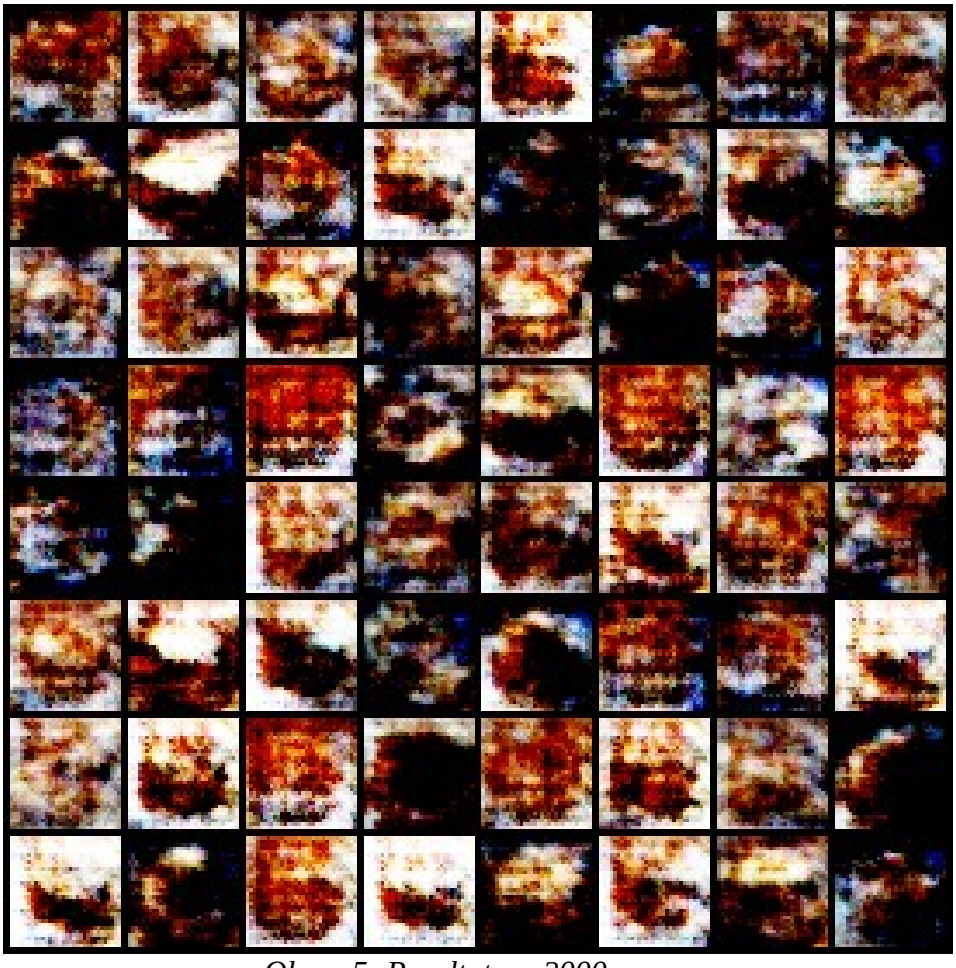
Trening okazał się być frustrujący i katorżniczy, w trakcie którego próbowano zmusić zmianami dyskryminator i generator do działania losowymi zmianami. Nasz model nie posiada żadnych usprawnień typu skrócenie kroku uczącego, ani nic innego co powodowałoby ograniczenie zbyt silnego modelu. Przy pierwszej lepszej wersji modelu – pozostawiłem go na noc aby sobie trochę podziałał.



Obraz 3: Rezultaty w zerowej epoce



Obraz 4: Rezultaty w 1000 epoce



Obraz 5: Rezultaty w 2000 epoce

Obraz 6: Rezultaty w 3000 epoce

Obraz 7: Rezultaty w 4000 epoce

Obraz 8: Rezultaty w 5000 epoce

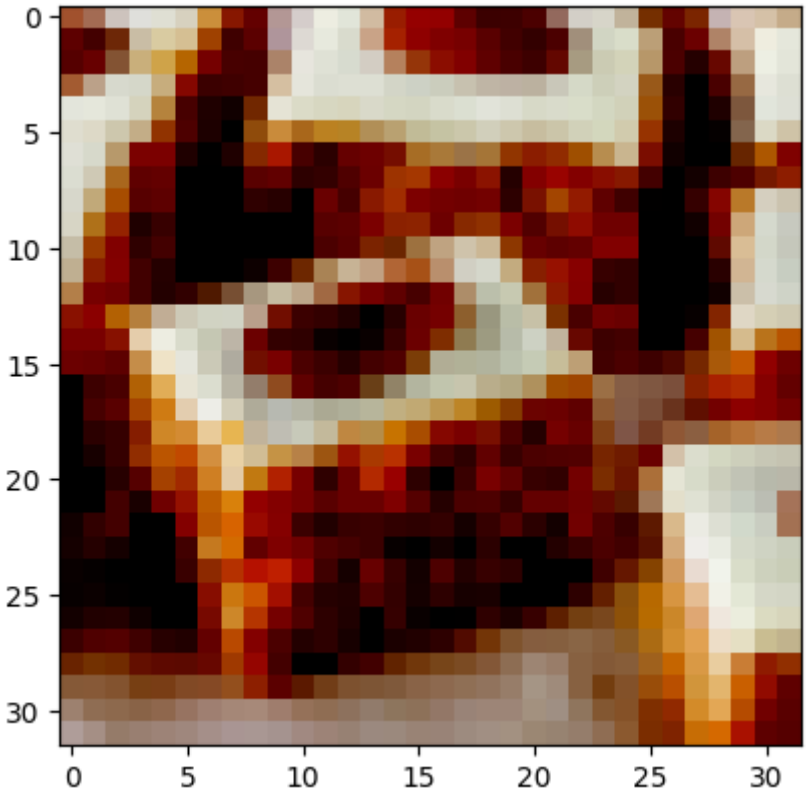
Obraz 9: Rezultaty w 6000 epoce

Wynik

Jak przykłady powyżej pokazują – coś się generuje. Trzeba jednak mieć dużo dobrej woli żeby nazwać to coś udanym ciastem.

Generowanie na podstawie wybranego zdjęcia

W tym podpunkcie będziemy starać się odwzorować wybrane, rzeczywiste zdjęcie. Wybrano najlepszy model generatora, który powstał po 6000 epokach trenowania.

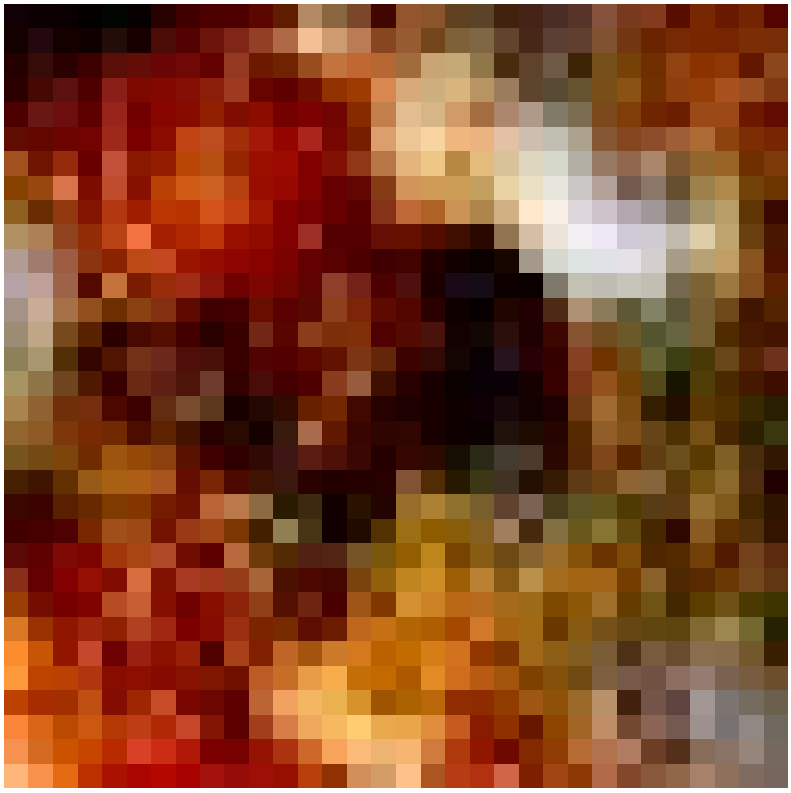


Obraz 10: Wybrane zdjęcie do odwzorowania

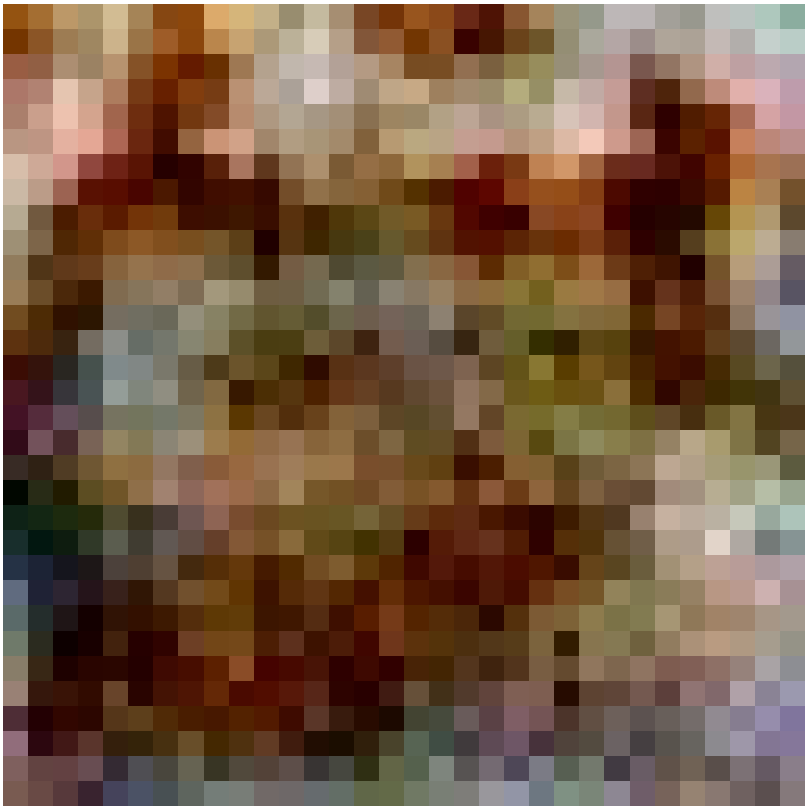
```
latent_vector = torch.randn(1, 64, device=device, requires_grad=True)
optimizer = optim.SGD([latent_vector], lr=0.01, momentum=0.9)

generator.eval()

# Optimization loop
for iteration in range(10000):
  optimizer.zero_grad()
  fake_image = generator(latent_vector)
  loss = F.mse_loss(fake_image, real_image)
  loss.backward()
  optimizer.step()

  if iteration % 1000 == 0:
    save_model_state(iteration, generator, discriminator, optimizer_generator,
optimizer_discriminator, './second_model_checkpoints')
    save_generated_images(generator, latent_vector, "./second_model_checkpoints",
iteration)
    print(f"Iteration {iteration}, Loss: {loss.item()}")
```

Używamy tutaj optymizatora gradientowego, SGD z learning rate = 0.1, oraz momentum = 0.9. Funkcja straty – błąd średnio-kwadratowy – będzie zwracała różnicę pomiędzy fałszywą obserwacją a wybranym wcześniej obrazem.

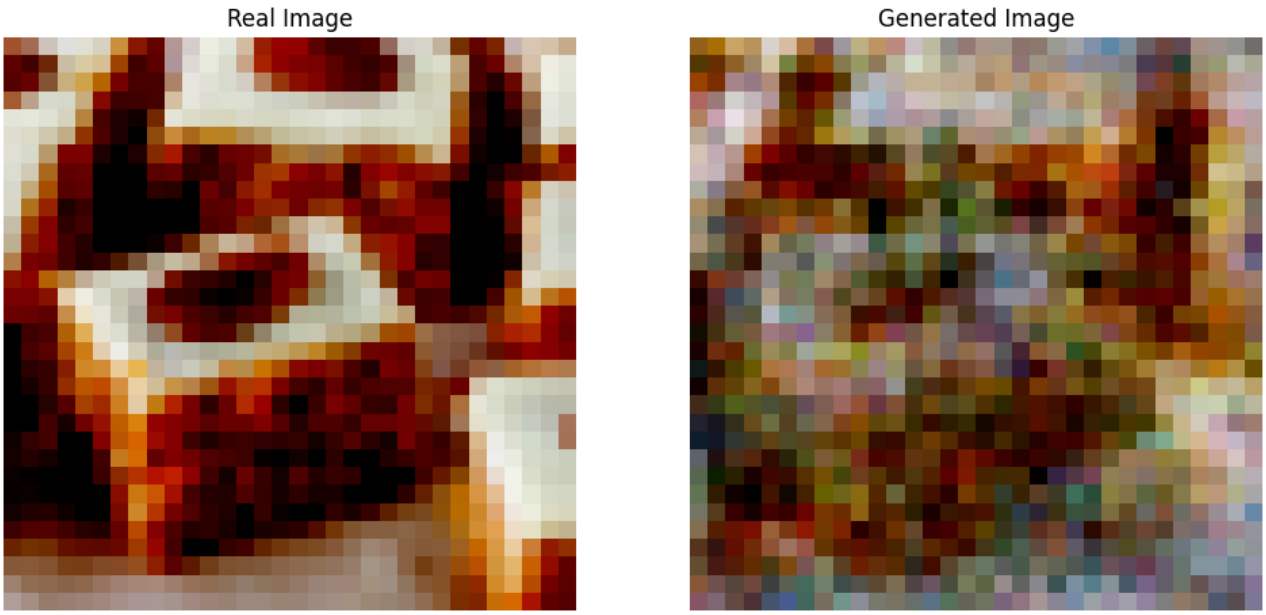
Jeśli chodzi o sam wynik eksperymentu to jest trochę dziwnie – potrzeba naprawdę sporo epok aby wynik jakkolwiek przypominał obraz wejściowy



Obraz 11: Ewolucja wynikowego obrazu



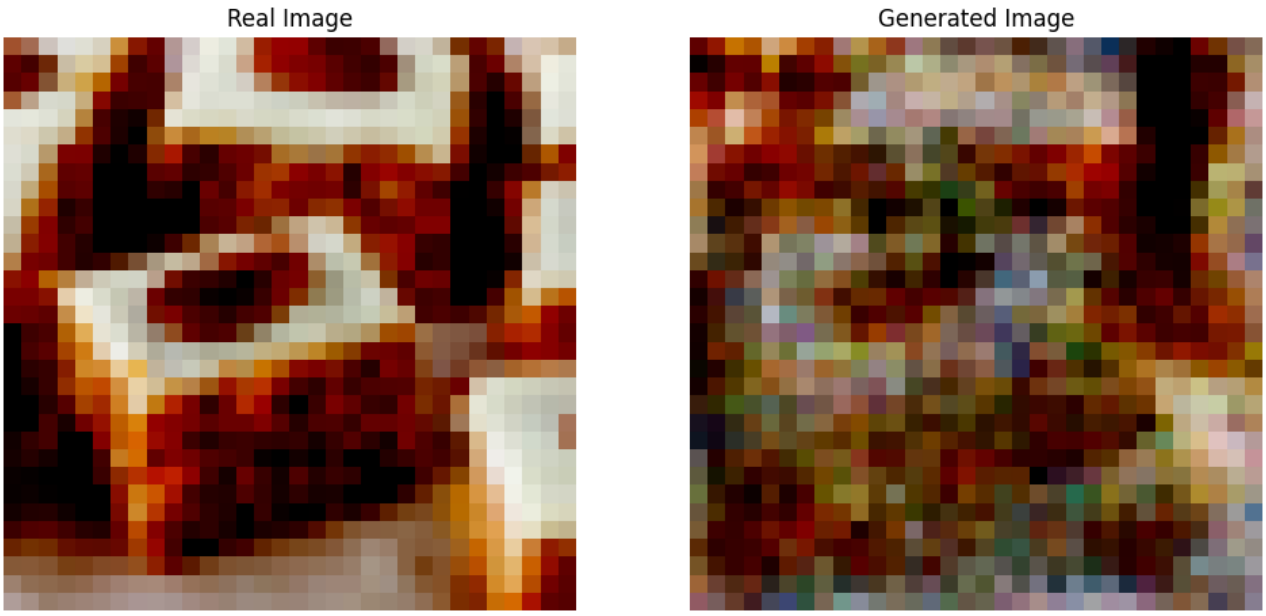
Obraz 12: Ewolucja wynikowego obrazu



Obraz 13: Porównanie rzeczywistego obrazu z wygenerowanym

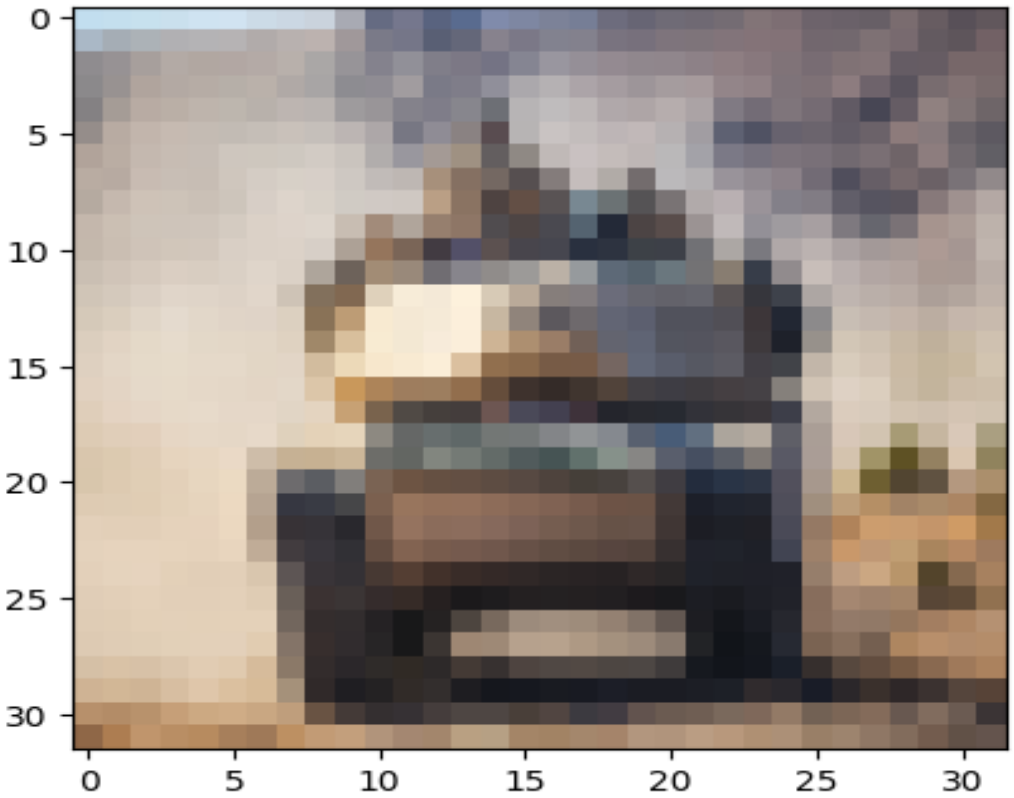
Losowe zmiany nie mają ogromnego wpływu na efekt końcowy, widać jakąś różnicę między wygenerowanymi obrazami, ale generalnie są bardzo zbliżone do siebie

```
real_image = dataset[2][0]
real_image = real_image.permute(1, 2, 0).cpu().numpy()
real_image = np.clip(real_image, 0, 1)

test_vector = latent_vector.clone()
indices_to_change = torch.randperm(latent_vector.size(0))[:10]
test_vector[indices_to_change] += torch.randn(64, device=device) * 0.1

fig, axes = plt.subplots(1, 2, figsize=(12, 6))
axes[0].imshow(real_image)
axes[0].set_title("Real Image")
axes[0].axis('off')

generated_image = generator(test_vector).detach().cpu()

if generated_image.ndim == 4:
  generated_image = generated_image.squeeze(0).permute(1, 2, 0).numpy()

axes[1].imshow(generated_image)
axes[1].set_title("Generated Image")
axes[1].axis('off')
```

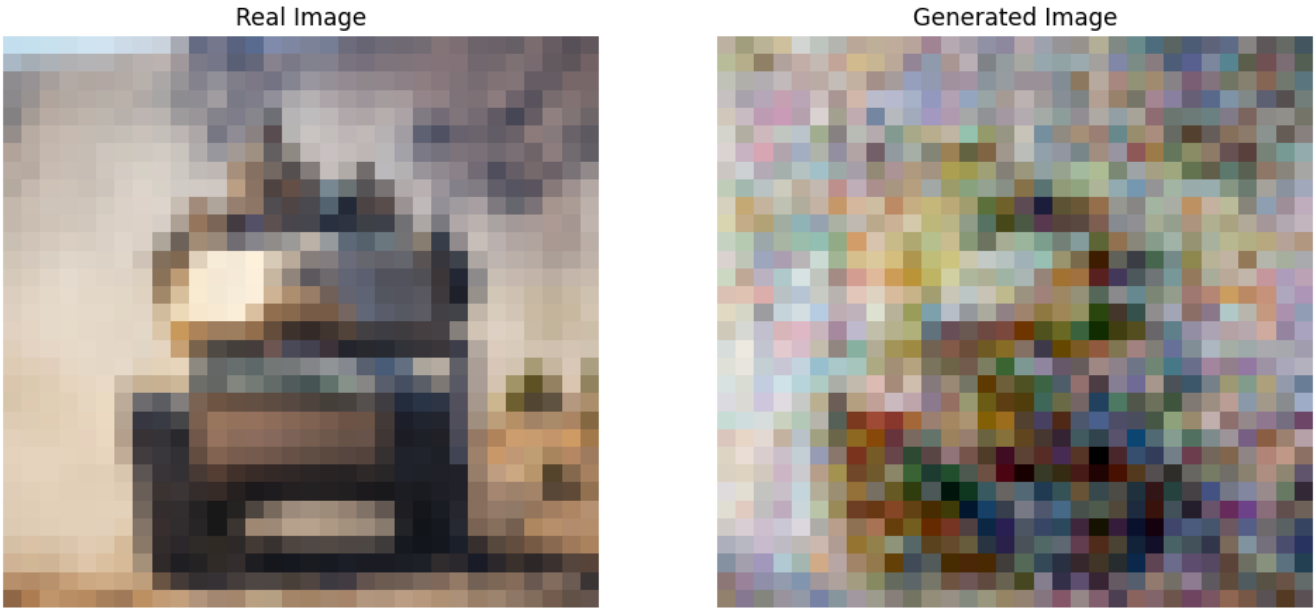


Obraz 14: Porównanie rzeczywistego obrazu z wygenerowanym przy użyciu zmienionego wektora

Jeśli zaś chodzi o generowanie obrazu na podstawie obrazu niezwiązanego z klasą wyniki wyszły dzikie. Wygenerowany obraz w niczym nie przypomina obrazu wejściowego.

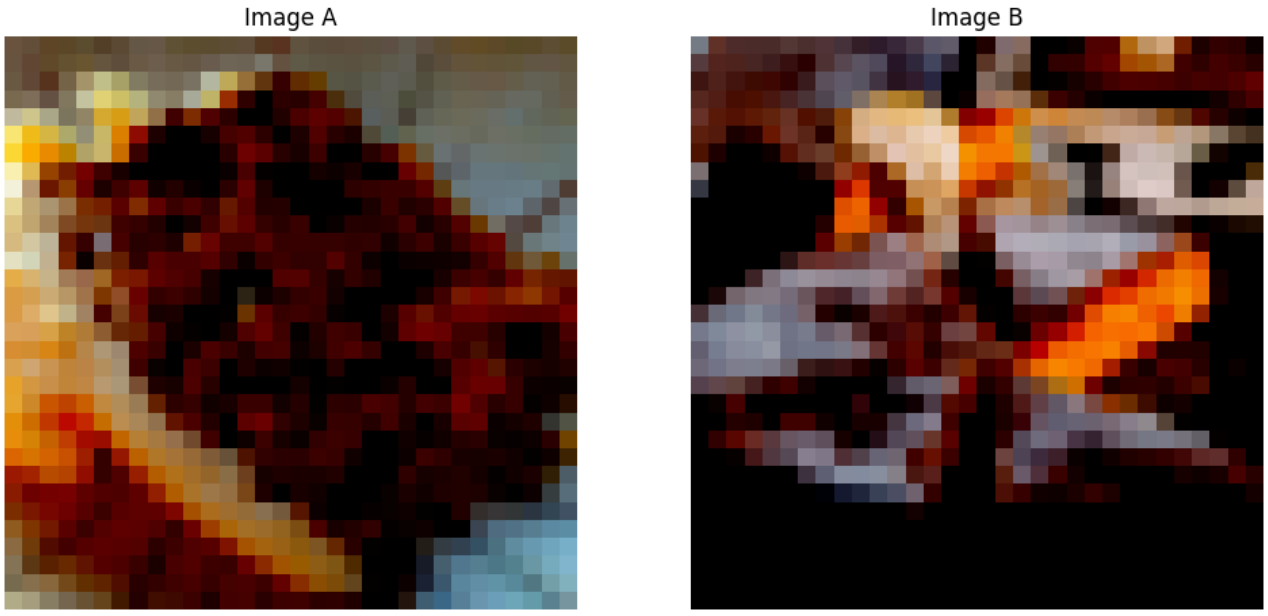


Obraz 15: Wybrane zdjęcie niezwiązane z wcześniej generowaną klasą

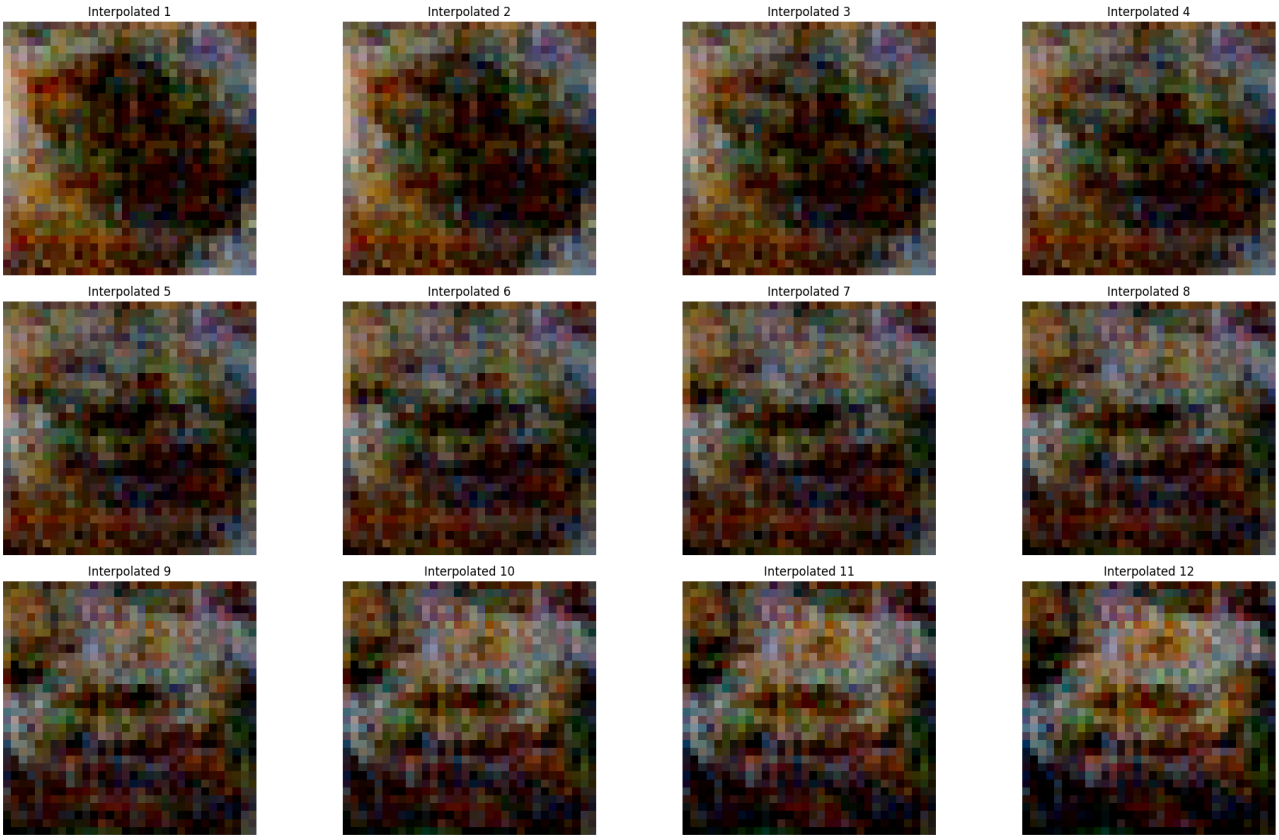


Obraz 16: Porównanie rzeczywistego obrazu z wygenerowanym

Część z interpolacją obrazów prezentuje się następująco:



Początkowa i końcowa interpolacja tylko trochę przypomina odpowiednie obrazy. Jeśli chodzi o obrazy pośrednie to widać jakaś różnicę między kolejnymi przejściami.



Wnioski

Najtrudniejszym elementem laboratorium było implementowanie pętli treningowej dyskryminatora i generatora. Męczyłem się z tym niemiłosiernie – a to przez 800 epok wygenerowany obraz niczym się nie różni, a to jeszcze jakieś błędy, które wszystko psuły. Gdy już model zaczął zwracać coś, co nie jest losowym szumem, poszło generalnie z górki. Z innych przemyśleń – zapisywanie stanu modelu co N epok ratuje zdrowie psychiczne – nie wiem, dlaczego jeszcze z tego nie korzystałem. Ogólnie lab oceniam pozytywnie.