Czy sieci neuronowe śnią o ciastach marchewkowych? - sprawozdanie

1. Model dyskryminatora poszedł dosyć szybko, na późniejszych etapach próbowałem bawić się biasem ze względów optymalizacyjnych, ale koniec końców zostawiłem domyślną implementację. Niestety w PyTorchu padding same nie jest dozwolony dla stride=2, więc ustawiłem domyślny równy 1:

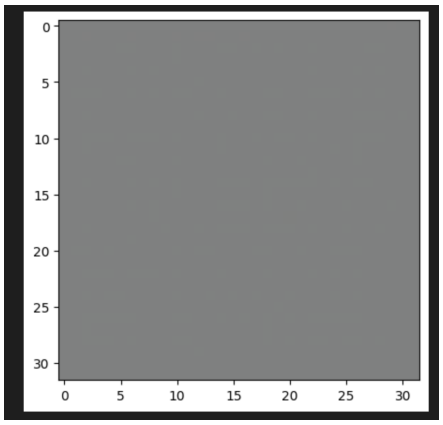
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3,out_channels=32,kernel_size=4, stride=2, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32,out_channels=64,kernel_size=4,stride=2, padding=1)
        self.conv3 = nn.Conv2d(in_channels=64,out_channels=64,kernel_size=4,stride=2, padding=1)
        self.batchNorm1 = nn.BatchNorm2d(32)
        self.batchNorm2 = nn.BatchNorm2d(64)
        self.relu = nn.LeakyReLU(0.2)
        self.flatten = nn.Flatten()
        self.dropout = nn.Dropout(0.2)
        self.dense = nn.Linear(64 * 4 * 4, 1)
        self.sigm = nn.Sigmoid()

    def forward(self, x):
        x = self.relu(self.batchNorm1(self.conv1(x)))
        x = self.relu(self.batchNorm2(self.conv2(x)))
        x = self.relu(self.batchNorm2(self.conv3(x)))
        x = self.flatten(x)
        x = self.dropout(x)
        x = self.sigm(self.dense(x))
        return x
```

2. Podobnie sprawnie poszła budowa generatora (z zastrzeżeniem mojej niezrozumiałości liczby filtrów - bardzo dziękuję za odpowiedź). Jedyną uwagę kieruję do podpunktu 2.3, gdzie chyba przez pomyłkę jest wpisane 128 kanałów 8 × 8 zamiast 64 kanały 4 × 4 (jeszcze przed warstwami konwolucji):

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

        self.fc1 = nn.Linear(64, 1024)

        self.deconv1 = nn.ConvTranspose2d(64, 64, kernel_size=4, stride=2, padding=1)
        self.deconv2 = nn.ConvTranspose2d(64, 128, kernel_size=4, stride=2, padding=1)
        self.deconv3 = nn.ConvTranspose2d(128, 256, kernel_size=4, stride=2, padding=1)

        self.final_deconv = nn.Conv2d(256, 3, kernel_size=5, stride=1, padding=2)
        self.relu = nn.LeakyReLU(0.2)
        self.tanh = nn.Tanh()

    def forward(self, x):
        x = self.fc1(x)
        x = x.view(-1, 64, 4, 4)
        x = self.relu(self.deconv1(x))
        x = self.relu(self.deconv2(x))
        x = self.relu(self.deconv3(x))
        x = self.tanh(self.final_deconv(x))
        return x
```

3. W celu sprawdzenia generatora napisałem prostą funkcję odnormalizującą wyniki i wyświetlającą ją matplotlibem. Cóż, wynik generacji nie należy do najciekawszych:

```
def imshow(img):
    img = img / 2 + 0.5
    npimg = img.squeeze().detach().numpy()
    plt.figure(figsize=(5,5))
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
random_noise = torch.randn(1, 64)
generated_image = generator(random_noise)
imshow(generated_image)
```



4. Kolejnym etapem było wczytanie ‘prawdziwego’ zbioru treningowego i wyświetlenie przykładowych obrazków. Ten etap chyba poszedł najszybciej, chociaż chwilkę kminiłem jak znormalizować obrazki:

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

trainset = torchvision.datasets.ImageFolder(root='train', transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=16, shuffle=True, num_workers=2)

dataiter = iter(trainloader)
images, labels = next(dataiter)
imshow(torchvision.utils.make_grid(images))
```

5. Prosty rozruch nie przyniósł większych problemów (i całe szczęście, bo takie później się pojawiły)

```
class SmallModel(nn.Module):
   def __init__(self):
       super(SmallModel, self).__init__()
       self.dense1 = nn.Linear(3, 3)
       self.dense2 = nn.Linear(3, 3)

   def forward(self, x):
       x = self.dense1(x)
       x = self.dense2(x)
       return x

def loss_function(x):
   return torch.mean(x) - 42

model = SmallModel()
x = torch.randn(3, 3)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

for i in range(10):
   output = model(x)
   loss = loss_function(output)
   loss.backward()
   optimizer.step()
   print(loss)
```

6. Teraz etap, który koniec końców pochłonął 3 dni, niezliczoną liczbę wulgaryzmów i sporo zasobów na Google Collab (skończyły się na szczęście zaraz po finalnym treningu i mogłem odtworzyć generator lokalne), czyli uczenie modelu, a w szczególności dyskryminatora. Możliwości, czemu nie działało jest kilka:
    - wspólne przekazywanie prawdziwej i wygenerowanej próbki bez odpięcia tej drugiej (detach())
    - błędne nałożenie szumu na etykiety (mniej prawdopodobne)

```
real_images = data[0]
real_labels = torch.ones(16)
fake_noise = torch.randn(16, 64)
fake_images = generator(fake_noise)
fake_labels = torch.zeros(16)

images = torch.cat((real_images, fake_images))
labels = torch.cat((real_labels - torch.rand(1) * 0.05, fake_labels + torch.rand(1) * 0.05))
```

Po wielu próbach, konsultacjach z prawdziwą jak i sztuczną inteligencją, udało się dojść do działającego (mniej więcej) kroku:

```
def train(discriminator, generator, trainloader, epochs, save_every, save_dir):
   criterion = nn.BCELoss()
   discriminator_optimizer = torch.optim.Adam(discriminator.parameters(), lr=0.00001)
   generator_optimizer = torch.optim.Adam(generator.parameters(), lr=0.00001)
   for epoch in range(epochs):
       loss1, loss2 = 0, 0
       for i, data in enumerate(trainloader, 0):
           b_size = data[0].size(0)

           discriminator.zero_grad()
           real_images = data[0].to(cuda0)
           real_labels = 0.95 + 0.05 * torch.rand(b_size, device=cuda0)
           outputs = discriminator(real_images).view(-1)
           loss1 = criterion(outputs, real_labels)
           loss1.backward()

           fake_noise = torch.randn(b_size, 64, device=cuda0)
           fake_images = generator(fake_noise)
           fake_labels = 0.05 * torch.rand(b_size, device=cuda0)
           outputs = discriminator(fake_images.detach()).view(-1)
           loss2 = criterion(outputs, fake_labels)
           loss2.backward()
           discriminator_optimizer.step()
```

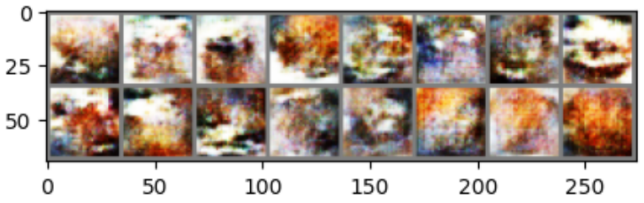
7. Trening generatora był prostszy w ogarnięciu i stanowi dalszą część powyższego kodu:

```
generator.zero_grad()
new_fake_noise = torch.randn(b_size, 64, device=cuda0)
new_fake_images = generator(new_fake_noise)
outputs = discriminator(new_fake_images).view(-1)
loss2 = criterion(outputs, torch.ones(b_size, device=cuda0))
loss2.backward()
generator_optimizer.step()
```

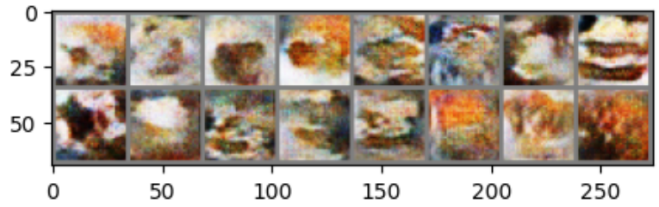
8. Monitoring efektów generacji robiłem co 50 epok, zapisując przy okazji stan obu sieci i obu optymalizatorów na Dysk Google (przydało się, bo Collab się raz wyłączył). Przed rozpoczęciem treningu wygenerowałem zestaw random_vectors, na którym śledziłem postępy:

```
if epoch % save_every == 0:
           torch.save({
           'epoch': epoch,
           'model_state_dict': discriminator.state_dict(),
           'optimizer_state_dict': discriminator_optimizer.state_dict(),
           'loss': loss1,
           }, './' + save_dir + '/discriminator_checkpoint' + str(epoch) + '.tar')
           torch.save({
           'epoch': epoch,
           'model_state_dict': generator.state_dict(),
           'optimizer_state_dict': generator_optimizer.state_dict(),
           'loss': loss2,
           }, './' + save_dir + '/generator_checkpoint' + str(epoch) + '.tar')
           print(loss1, loss2)
           generated_image = generator(random_vectors)
           imshow(torchvision.utils.make_grid(generated_image.cpu())
```

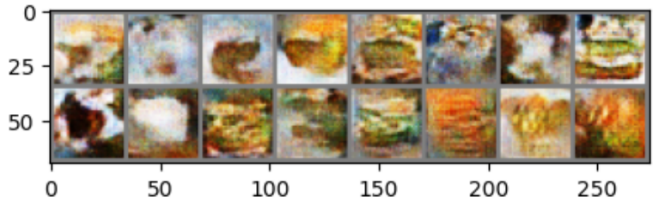
9. Rezultaty po 2000 epok z daleka, przy zamkniętym jednym oku i zmrużonym drugim oku wyglądały nawet jak ciasto (pomarańczowe plamy zapewne mają podkreślić marchewkę), tylko bardziej już po próbie zjedzenia przez niemowlaka. Niemniej po całym przebiegu treningu widać, że coś się działo:
    a) epoka 300:



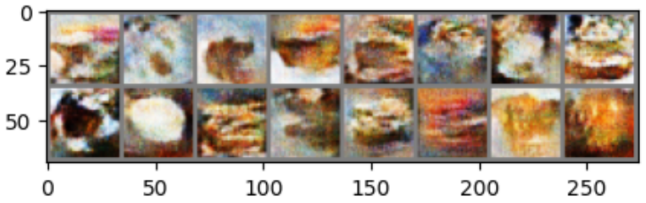
    b) epoka 1000:



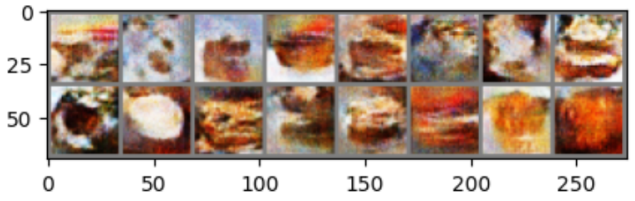
    c) epoka 1500:



    d) epoka 2000:

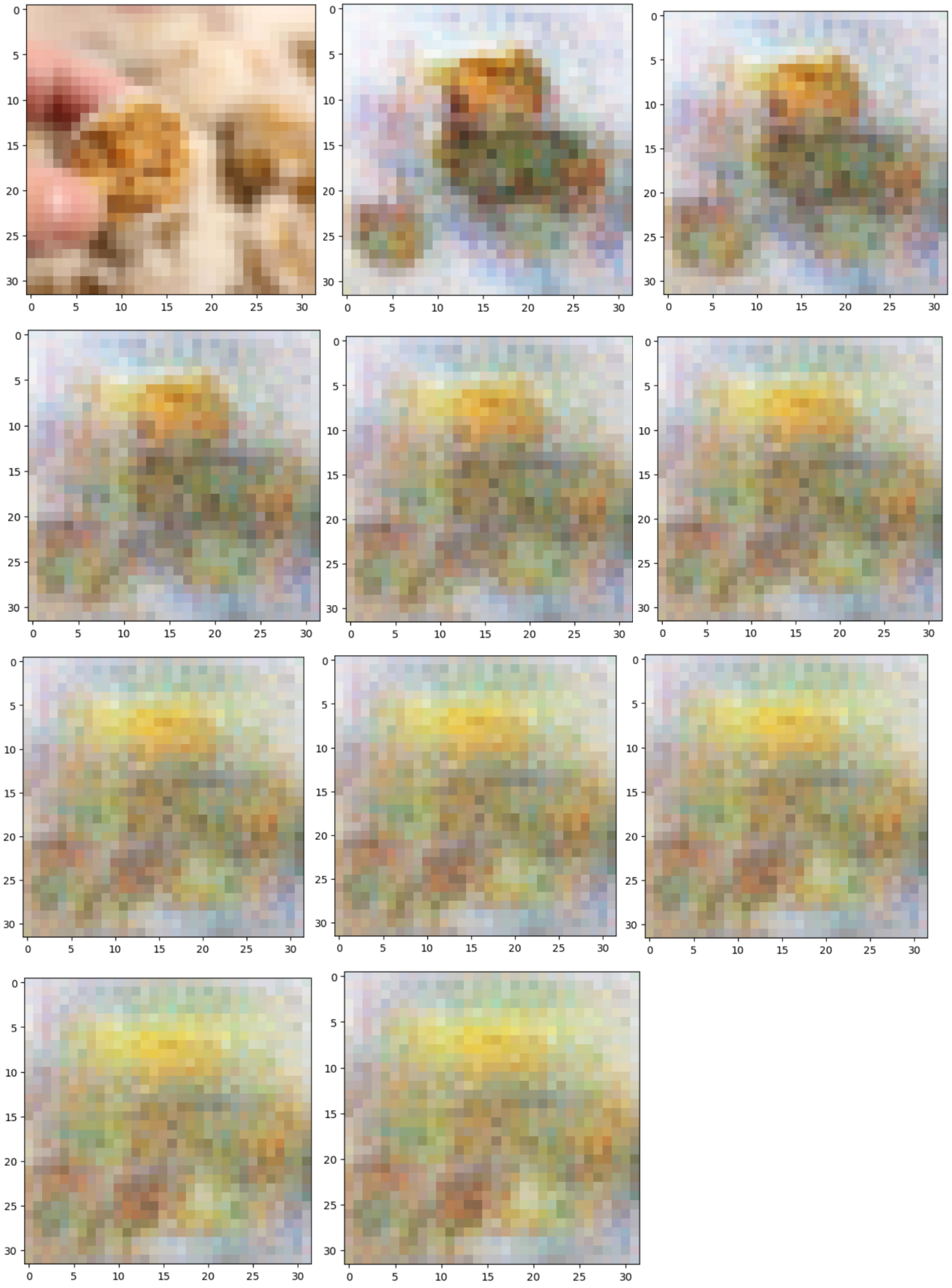


    e) epoka 2550 (najlepszy wynik):



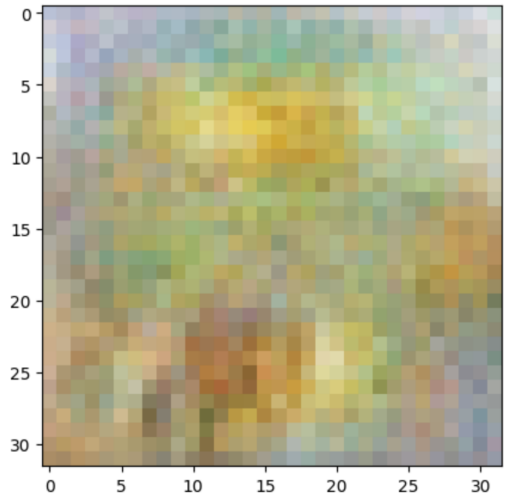
10. Jak wyszła próba odwzorowania dowolnego rzeczywistego obrazu? Średnio, poważniejsze zmiany niestety dało się zauważyć dopiero nie po kilkunastu, lecz kilkuset epokach, więc dla pewności dałem 1000:

```
real_image = images[0]
random_vector = torch.randn(1, 64, requires_grad=True)
loss_function = nn.MSELoss()
optimizer = torch.optim.SGD([random_vector], lr=0.01, momentum=0.9)

for i in range(1000):
   optimizer.zero_grad()
   fake_image = generator(random_vector)
   if i % 100 == 0:
       imshow(fake_image)
   loss = loss_function(fake_image.squeeze(), real_image)
   loss.backward()
   optimizer.step()
final_vector = random_vector.detach().clone()
```



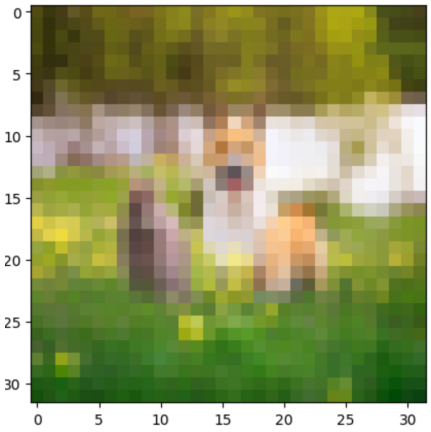
11. Zmiana wartości w finalnym wektorze za dużo nie zmieniła. W niektórym fragmentach widać pociemnienie lub pojaśnienie:

```
import random
changed_vector = final_vector.clone()
random_indices = random.sample(range(64), 10)
changed_vector[0, random_indices] = torch.randn(10)
imshow(generator(changed_vector))
```

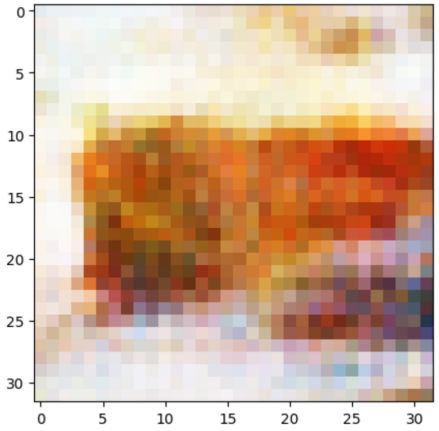


12. Przy wzorcowym obrazie spoza ciast generator chyba dostał głupawki, bo po prostu rozmazał pierwotną generację i dodał trochę zieleni (wzorcem jest zdjęcie psa i kota z następnych labów):

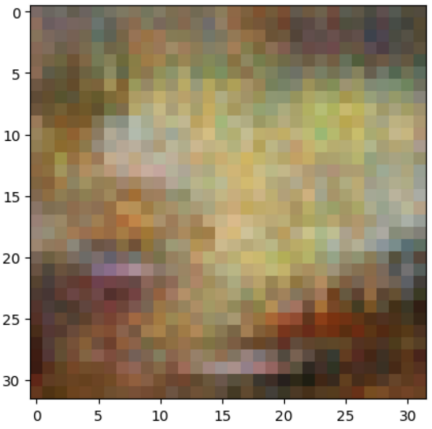
Rzeczywisty obraz do imitacji:



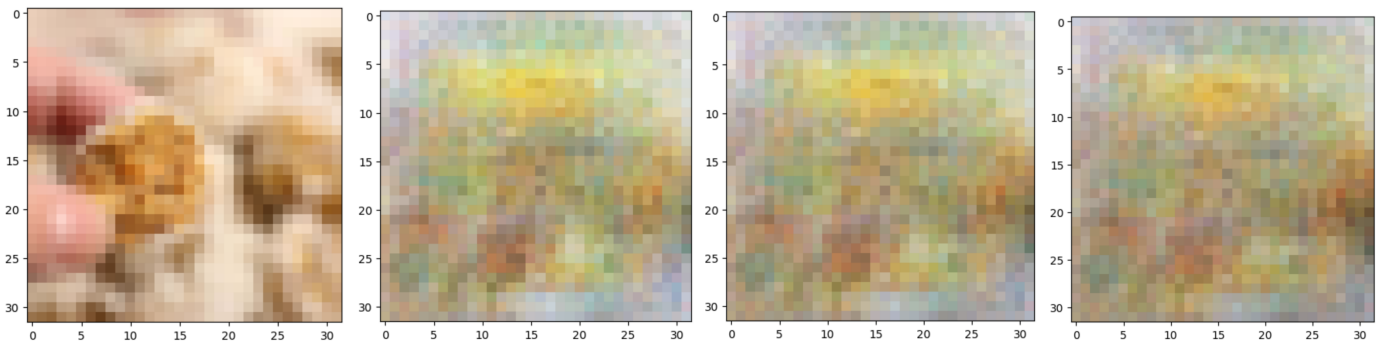
Pierwsza generacja:

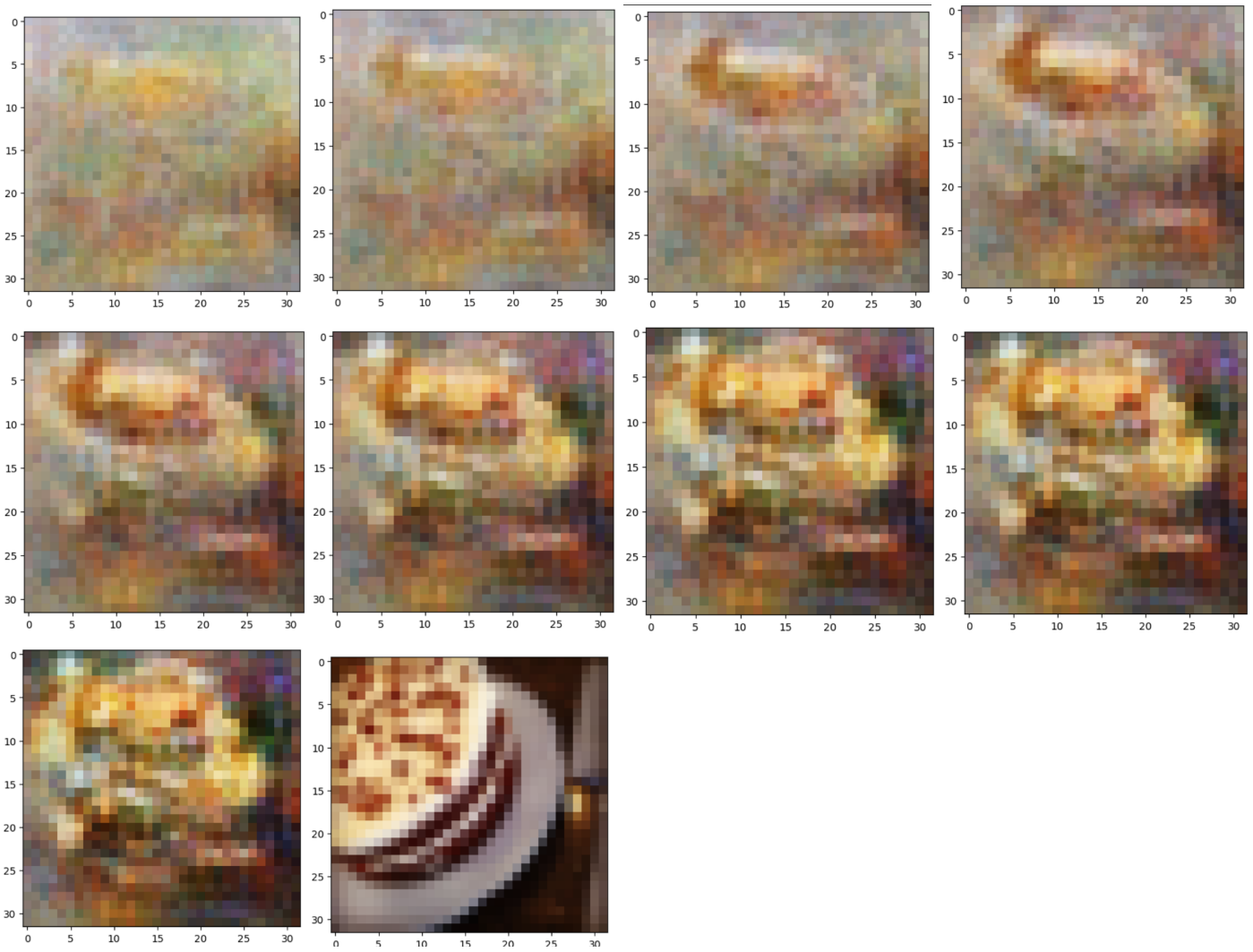


Generacja po treningu:



13. A jak przebiegła interpolacja? Cóż, o ile próbę dostosowania się do innego obrazu treningowego skwituję chwilą konsternacji nad swoim dziełem, to interpolacja między jedną a drugą próbą wyszła całkiem nieźle:





14. Uh, ten lab pochłonął dużo czasu i nerwów, ale koniec końców coś się udało wytrenować i wyświetlić. Zdecydowanie najwięcej czasu pochłonęło sprawdzanie rzetelności treningu, całą resztę jako tako udało się skleić (interpolację wykonałem poprzez kolejne sklejane wartości dwóch wektorów w różnych proporcjach, nie jestem pewien czy to była idealna ścieżka, ale przejście jako takie jest widoczne). Koniec końców trening na Google Colab trwał ponad 3 godziny i ostatnie 500 epok nie przynosiło znaczącej zmiany (doszedłem do wyniku 4 z kawałkiem dla generatora zaczynając od mniej niż 1). Gdyby nie durne błędy związane z treningiem dyskryminatora (nazwa całkiem zabawna, kojarzy mi się z maszynami dr. Heinza Dundersztyca z Fineasza i Ferba), to pewnie udałoby się to szybciej skończyć i wysłać w terminie :P Ale przynajmniej na długo będę pamiętał o co w tych GANach chodzi

Michał Wójcik