GAN für Katzenbilder

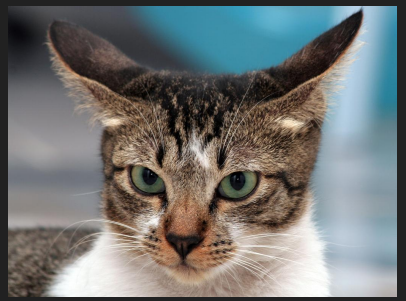
Miguel Sarasa

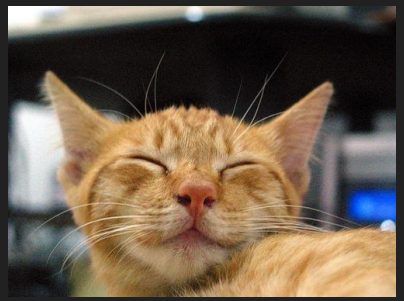
Gliederung

- Ziel und Datensatz
- Generative Adversarial Networks
- Durchführung
  - Preprocessing
  - Neuronale Netze
  - Trainingsloop
  - Ergebnisse
- Ausblick

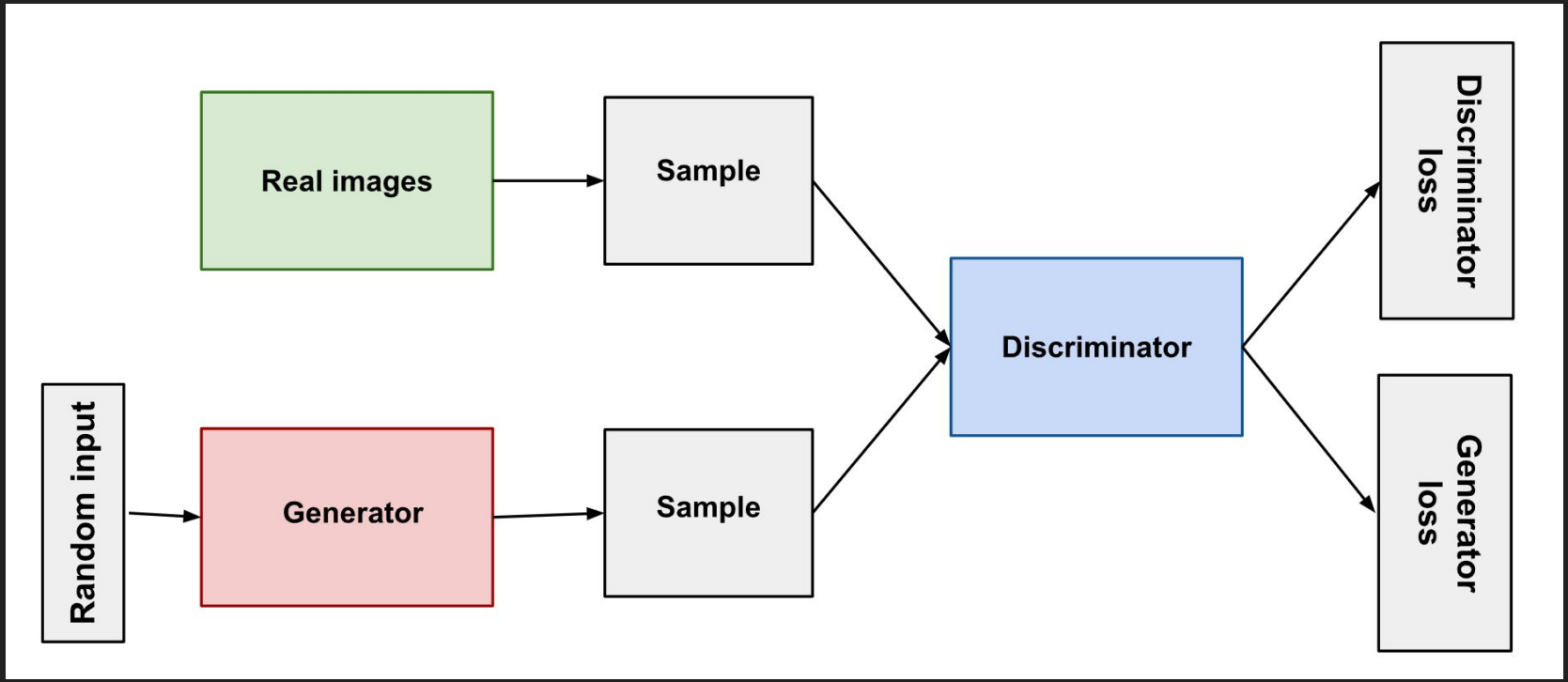
Ziel und Datensatz

- Gibt es genug Katzenbilder im Internet?
  - Es gibt nie genug Katzenbilder, also generieren wir noch mehr
- Datensatz: 10 000 Katzenbilder
  - Gesicht oder ganze Katze
  - Fast keine anderen Tiere oder Menschen
  - zusätzlich Daten zur Position von Mund, Augen und Ohren (uninteressant)



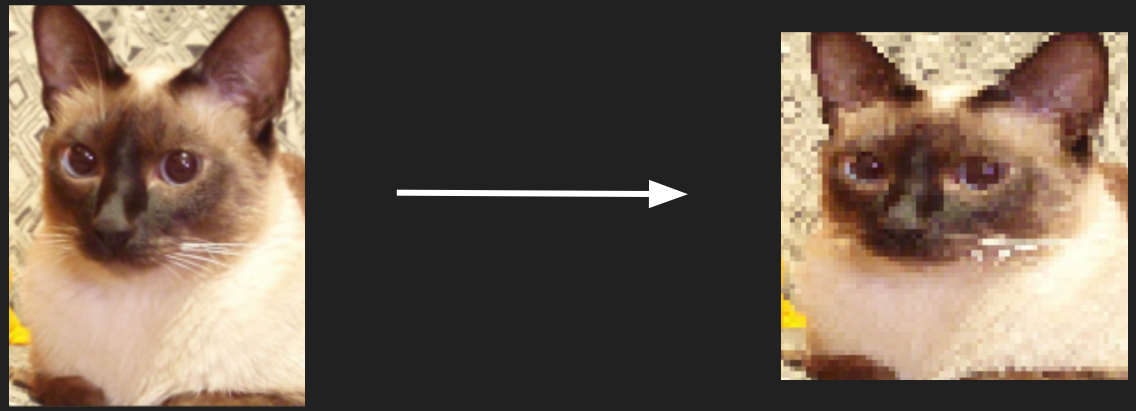


Generative adversarial networks

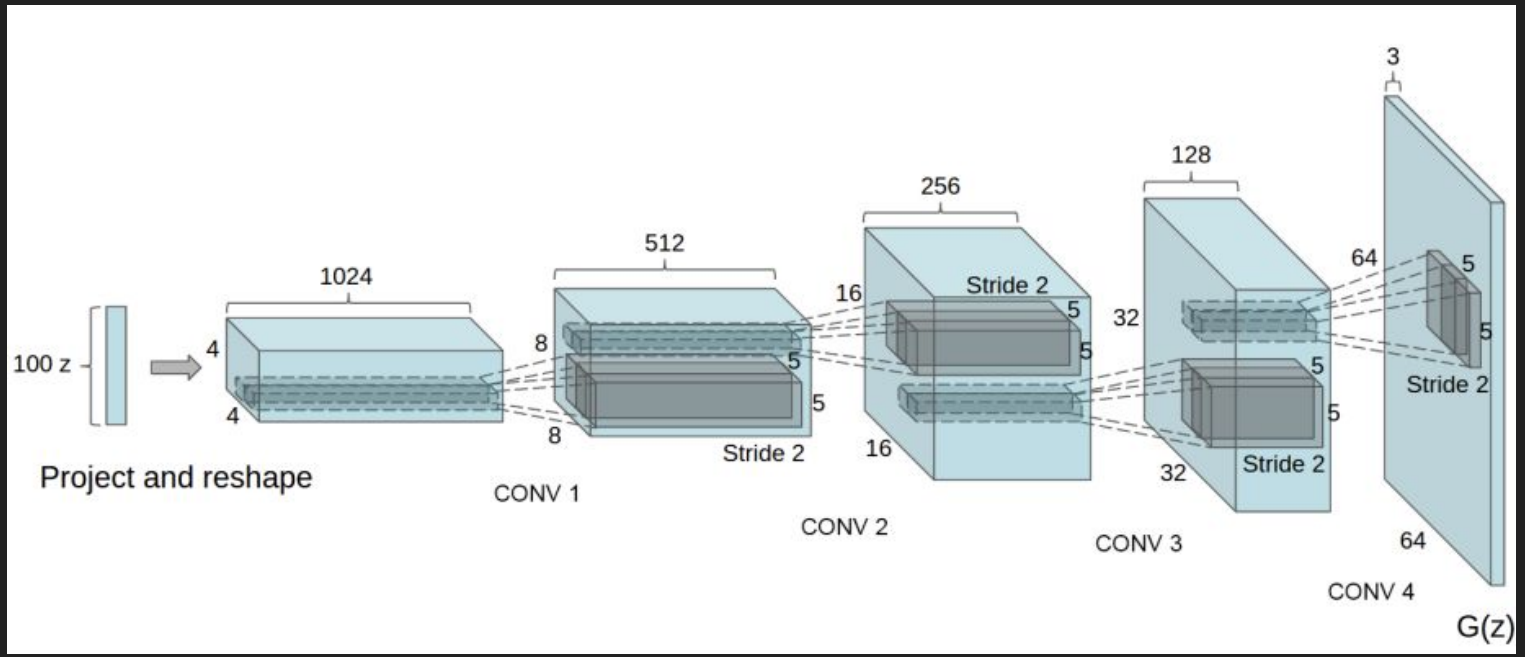


Durchführung: Preprocessing

- Skalieren auf 64x64 (opencv)
- Normieren auf das Intervall [0,1]
- Konvertieren zum Pytorch Tensor-Format
- In den VRAM laden



Durchführung: Generator



https://arxiv.org/pdf/1511.06434.pdf

Durchführung: Generator

```python
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            # State size 100x1x1
            nn.ConvTranspose2d(100, 512, 4, 1, 0, bias=False), # Reverse convolution
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            # state size 512 x 4 x 4
            nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            # state size 256 x 8 x 8
            nn.ConvTranspose2d( 256, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            # state size 128 x 16 x 16
            nn.ConvTranspose2d( 128, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            # state size 64 x 32 x 32
            nn.ConvTranspose2d( 64, 3, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. 3 x 64 x 64
        )

    def forward(self, input):
        return self.main(input)
```

Durchführung: Discriminator

```python
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main=nn.Sequential(
            # State size: 3x64x64 (3 colors, 64x64 pixels)
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # State size: 64x32x32 (stride of 2 halves each dimension of the input)
            nn.Conv2d(64, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            # State size: 128x16x16
            nn.Conv2d(128, 256, 4, 2, 1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            # State Size: 256x8x8
            nn.Conv2d(256, 512, 4, 2, 1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            # State size 512x4x4
            nn.Conv2d(512, 1, 4, 1, 0, bias=False), # single output neuron
            nn.Sigmoid()
            # State size 1x1x1
        )
    def forward(self, input):
        return self.main(input)
```

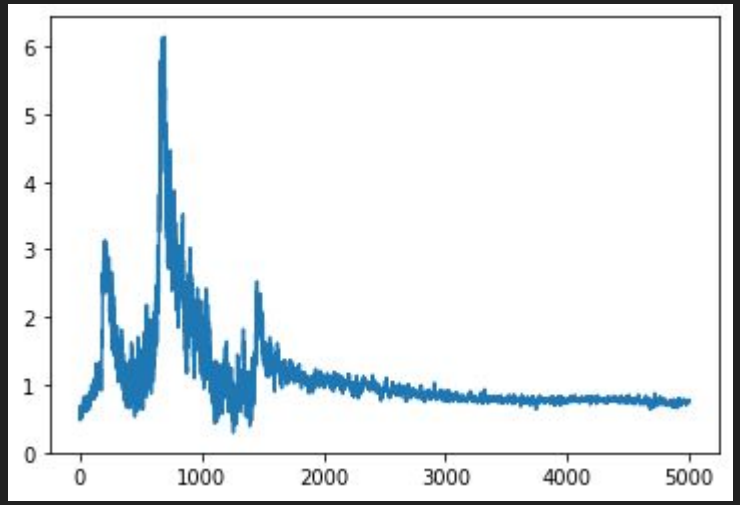
Trainingsloop

- Discriminator: Inferenz auf echtem Bild, Fehler berechnen
- Backpropagation
- Bild generieren (mit dem Generator-Netzwerk)
- Discriminator: Inferenz auf generiertem Bild, Fehler berechnen
- Backpropagation
- Update-Schritt Discriminator

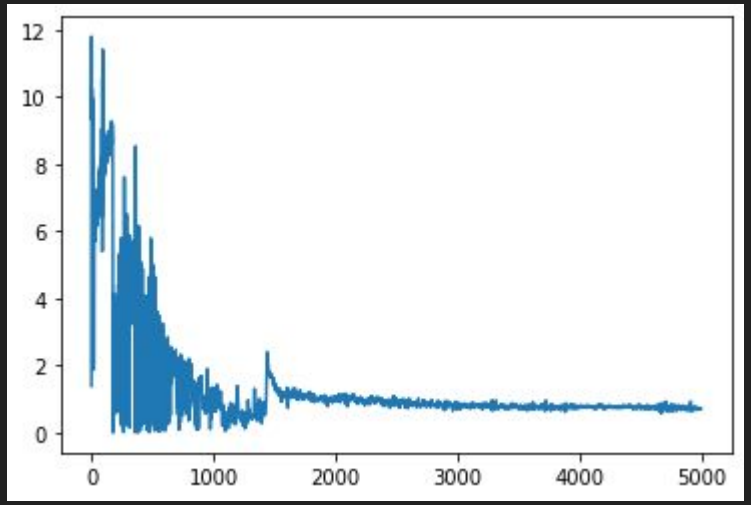
- Discriminator: Inferenz auf generiertem Bild (nach Training)
- Fehler des Generators berechnen
- Backpropagation und Update-Schritt Generator

Ergebnisse

Training: 5000 Batches zu je 100 Bildern (2 Stunden auf 1070 Ti)

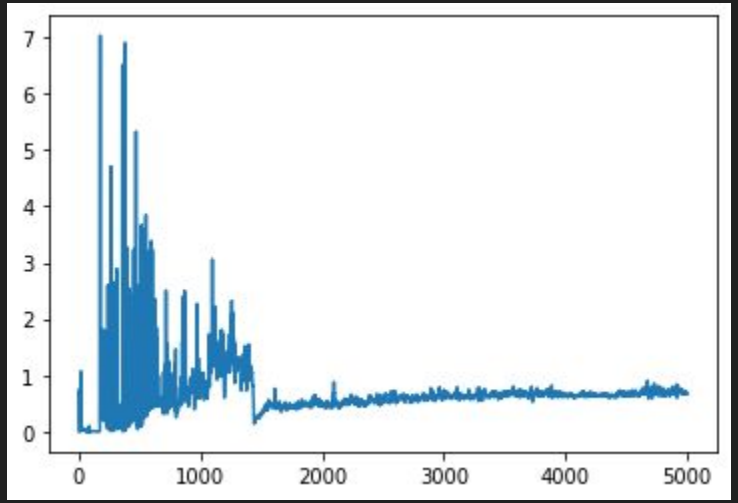


Discriminator: Loss-Funktion
auf echten Bildern



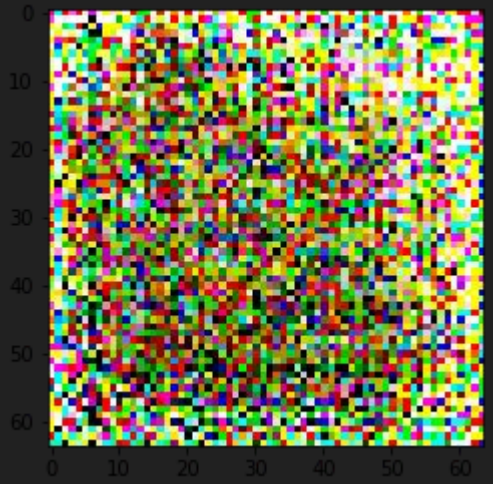
Discriminator: Loss-Funktion
auf generierten Bildern

Ergebnisse

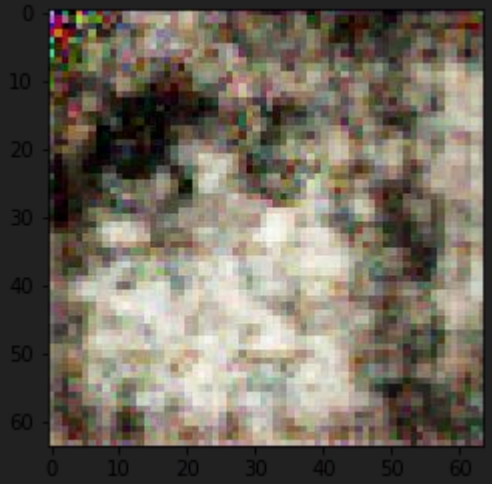


Generator: Loss-Funktion

Ergebnisse



Nach 600 Epochen



Nach 5000 Epochen

Ausblick

- Mehr Training notwendig, um bessere Ergebnisse zu erzielen
- Discriminator wird über 5000 Epochen immer besser
  - Learning rate verringern, damit der Generator hinterherkommt
- 64x64 ist vielleicht zu wenig
  - Mehr Layers in die Netzwerke einfügen oder Strides erhöhen