11-695: AI Engineering
GAN II

LTI/SCS

Spring 2020

Outline

Generator: Decoding

Carnegie Mellon

generator.py

```
1  model = tf.keras.Sequential()
2  model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
3  model.add(layers.BatchNormalization())
4  model.add(layers.LeakyReLU())
5
6  model.add(layers.Reshape((7, 7, 256)))
7  assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size
8
9  model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=Fal
10 assert model.output_shape == (None, 7, 7, 128)
11 model.add(layers.BatchNormalization())
12 model.add(layers.LeakyReLU())
13
14 ...
15 model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
16                                  use_bias=False, activation='tanh'))
17 assert model.output_shape == (None, 28, 28, 1)
```

- Recall: images data → we use Conv/Deconv
- "Expanding" the noise to the desired output dimension
- Similar to AutoEncoder, it's natural to use TransposeConv

Discriminator: (Binary) Classification

Carnegie Mellon

discriminator.py

```
1  model = tf.keras.Sequential()
2  model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
3                                      input_shape=[28, 28, 1]))
4  model.add(layers.LeakyReLU())
5  model.add(layers.Dropout(0.3))
6
7  model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
8  model.add(layers.LeakyReLU())
9  model.add(layers.Dropout(0.3))
10
11 model.add(layers.Flatten())
12 model.add(layers.Dense(1))
```

- As opposed to Generator, it uses Convolution
- By now, you can see the similarity between GAN and AE?

Losses

Carnegie Mellon

discriminator_loss.py

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

real_loss = cross_entropy(tf.ones_like(real_output), real_output)
fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
disc_loss = real_loss + fake_loss
```

- Recall: $C(D) = \frac{1}{m}\sum_{i=1}^{m}[\log D_\theta(x^{(i)}) + \log(1 - D_\theta(G_\psi(z^{(i)})))]$
- There are 2 parts: for real and for fake images

generator_loss.py

```
generator_loss = cross_entropy(tf.ones_like(fake_output), fake_output)
```

- Recall: $C(G) = \frac{1}{m}\sum_{i=1}^{m}[\log(1 - D_\theta(G_\psi(z^{(i)})))]$
- It has only the latter term as compared to Discriminator

Forward: Calculating Loss

Carnegie Mellon

forward.py

```
noise = tf.random.normal([BATCH_SIZE, noise_dim])

with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
    generated_images = generator(noise, training=True)

    real_output = discriminator(images, training=True)
    fake_output = discriminator(generated_images, training=True)

    gen_loss = generator_loss(fake_output)
    disc_loss = discriminator_loss(real_output, fake_output)
```

- Alternating the updates for G and D
- So we need 2 types of gradient $\rightarrow$ 2 gradient tapes

Backward: Update Params

Carnegie Mellon

backward.py

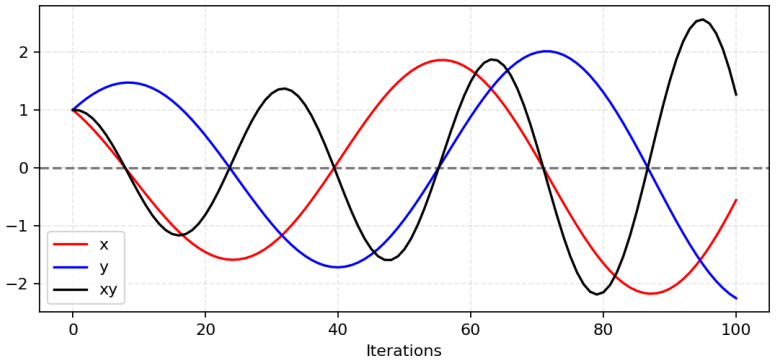
```
# this will be done for every batch, and so G and D are alternatively updated
# note: outside the scopes of GradientTape
gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
gradients_of_discriminator =
    disc_tape.gradient(disc_loss, discriminator.trainable_variables)

generator_optimizer.apply_gradients(zip(gradients_of_generator,
                                        generator.trainable_variables))
discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
                                            discriminator.trainable_variables))
```

- We need to update G and D separately by making use of 2 tapes
- Finally: play more with ▸ code

Outline

So far: easy code, so as training? NO!

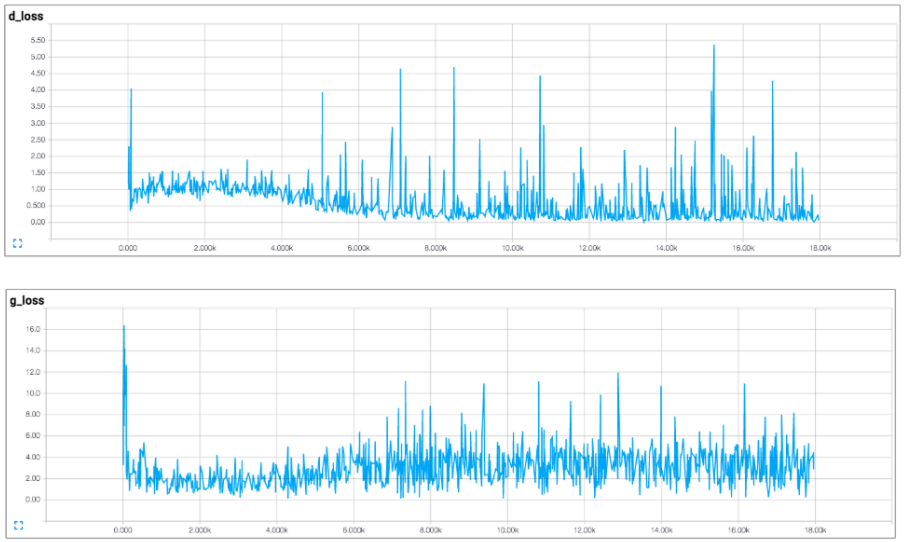
Carnegie Mellon



- We have 2 networks, and we alternate in updating them
  - Can cause "oscillation" or "overshooting"
- To achieve (Nash-)Equilibrium in this non-cooperative game: result is constant regardless any player's actions
- Example: G to update x for $\min_x f_1(x) = \min_x xy$ and D to update y for $\max_y f_2(y) = \max_y xy = \min_y(-xy)$
  - $\nabla_{f_1} = y \Rightarrow x^{t+1} = x^t - \eta y$ while $\nabla_{f_2} = -x \Rightarrow y^{t+1} = y^t + \eta x$
  - Oscillation happens $\rightarrow$ Hard to achieve equilibrium

Image credit: Lil'Log

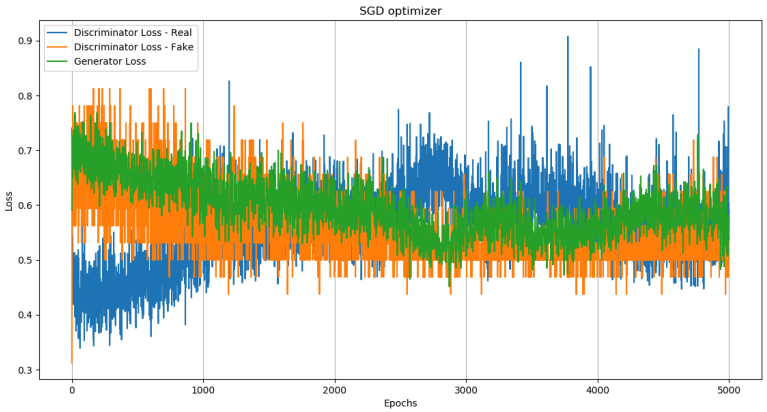
Loss Function Problem

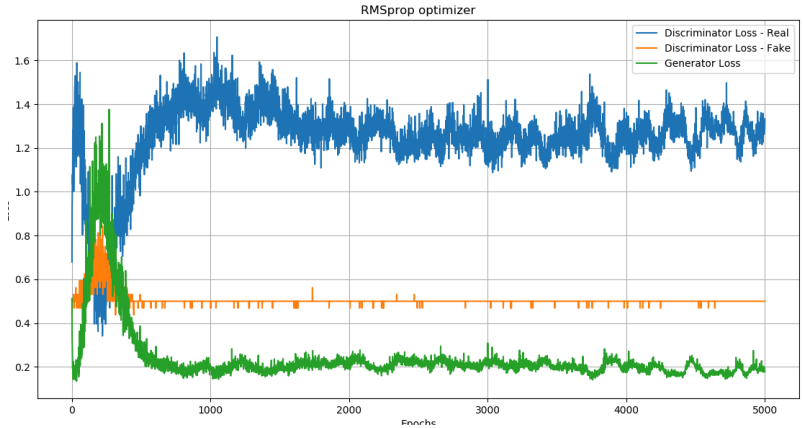


- Recall for G_ψ: $\nabla G_\psi = \nabla_\psi \frac{1}{m} \sum_{i=1}^{m} [\log(1 - D_\theta(G_\psi(z^{(i)})))]$
- Recall: The job of D is easier: binary classification
- Right from the start, G is bad, D can easily outperform:
  - so $\log(1 - D_\theta(G_\psi(z^{(i)}))) = 0$
  - means G learns nothing afterwards

Image credit: carpedm20 github

Loss Function Problem

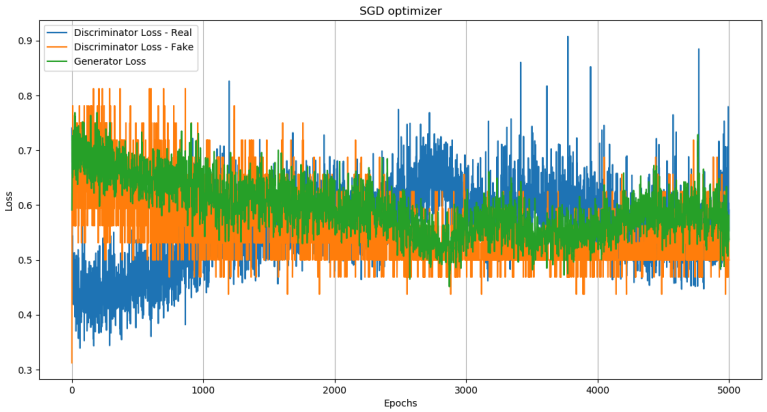


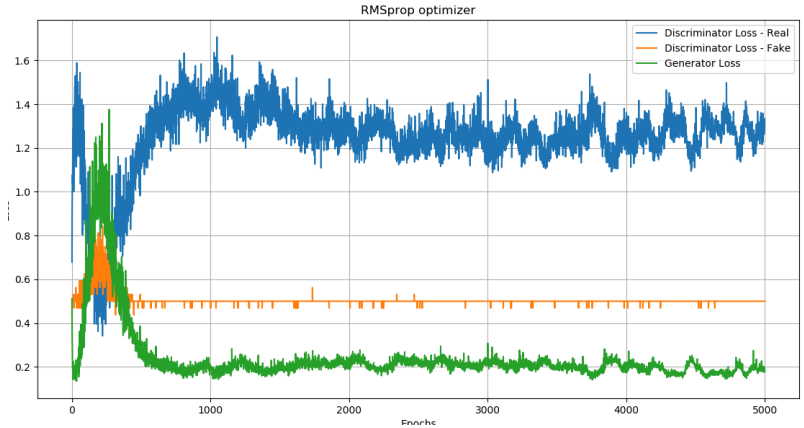


- Be careful: Losses visualization might not be so interpretable!

Image credit: Mirantha Jayathilaka (Medium.com), Soumith Chintala

Loss Function Problem



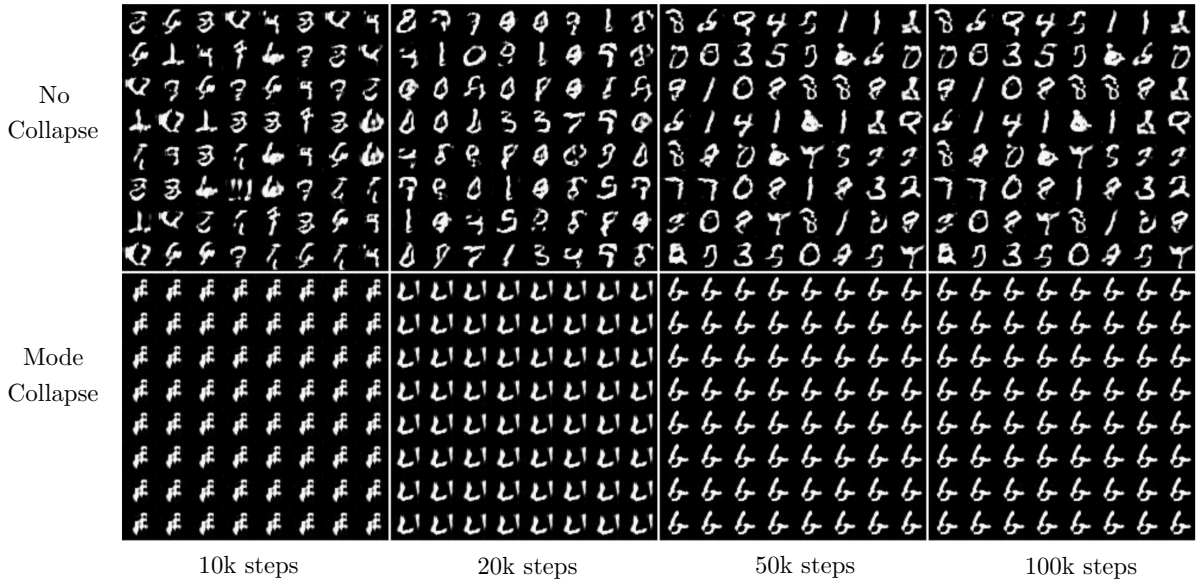


- Be careful: Losses visualization might not be so interpretable!
- The optimization objective is different in GAN
  - In others, you optimize towards the ground truths
  - Now you are optimizing towards your opponent
- It can happen: losses increase, but result quality *also* increases!!!
- But if D loss goes to zero, things might be wrong.

Image credit: Mirantha Jayathilaka (Medium.com), Soumith Chintala

Mode (Modal) Collapse

Carnegie Mellon



- Real world distribution is multi-modal and complex
- Can be "stuck" in one mode by pushing to optimize it hard
  - *E.g.* always generate (almost perfect) digit 1 in MNIST
  - In (theoretial) optimization perspective, nothing's wrong
  - But G over-simplifies the data distribution
- A painfully hard yet very common problem

Image credit: Luke Metz *et al.* 2017

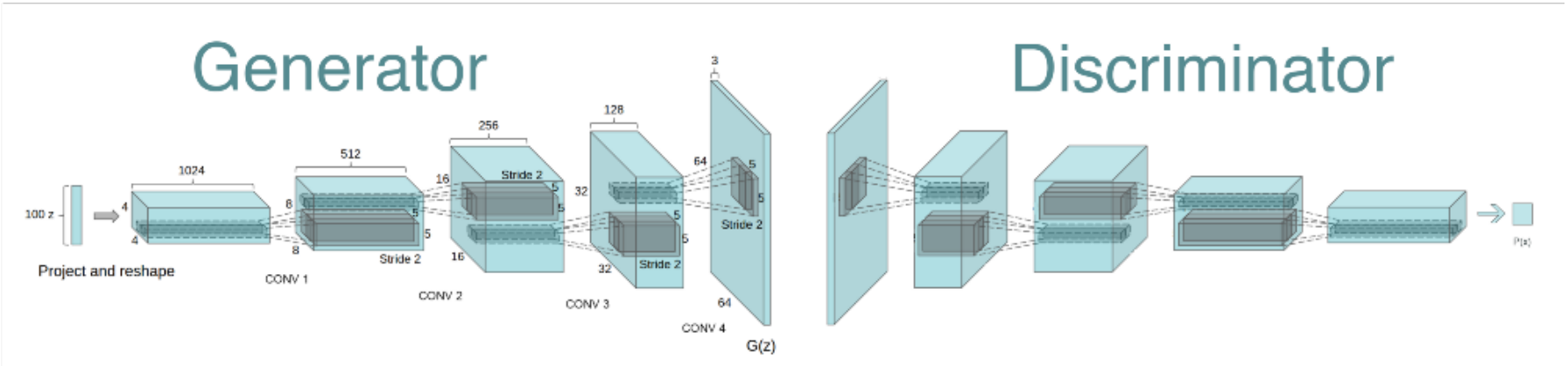
Outline

Improving Architectures

Carnegie Mellon

- Deep Convolutional (DCGAN)
- Stacked GAN (SGAN)
- Progressive GAN

DCGAN[1]



- Strided Convolutional instead of Max Pooling
- Use TransposeConv for upsampling
- No FC layer
- ReLU for G (except last layer uses Tanh), Leaky ReLU for D
- Use Batch normalization BN.

[1] https://arxiv.org/pdf/1511.06434.pdf

Image credit: Alec Radford & Luke Metz & Soumith Chintala 2016, mxnet.io

Recall: DCGAN: Generator

Carnegie Mellon

dcgan_generator.py

```
1  model = tf.keras.Sequential()
2  model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
3  model.add(layers.BatchNormalization())
4  model.add(layers.LeakyReLU())
5
6  model.add(layers.Reshape((7, 7, 256)))
7  assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size
8  model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
9                                   padding='same', use_bias=False))
10 assert model.output_shape == (None, 7, 7, 128)
11 model.add(layers.BatchNormalization())
12 model.add(layers.LeakyReLU())
13
14 model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),
15                                  padding='same', use_bias=False))
16 assert model.output_shape == (None, 14, 14, 64)
17 model.add(layers.BatchNormalization())
18 model.add(layers.LeakyReLU())
19
20 model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2),
21                                  padding='same', use_bias=False, activation='tanh'))
```

- No FC, BatchNorm and LeakyReuLU and Tanh

Recal: DCGAN: Discriminator

Carnegie Mellon

dcgan_discrminator.py

```
1  model = tf.keras.Sequential()
2  model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
3                          input_shape=[28, 28, 1]))
4  model.add(layers.LeakyReLU())
5  model.add(layers.Dropout(0.3))
6
7  model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
8  model.add(layers.LeakyReLU())
9  model.add(layers.Dropout(0.3))
10
11 model.add(layers.Flatten())
12 model.add(layers.Dense(1))
```

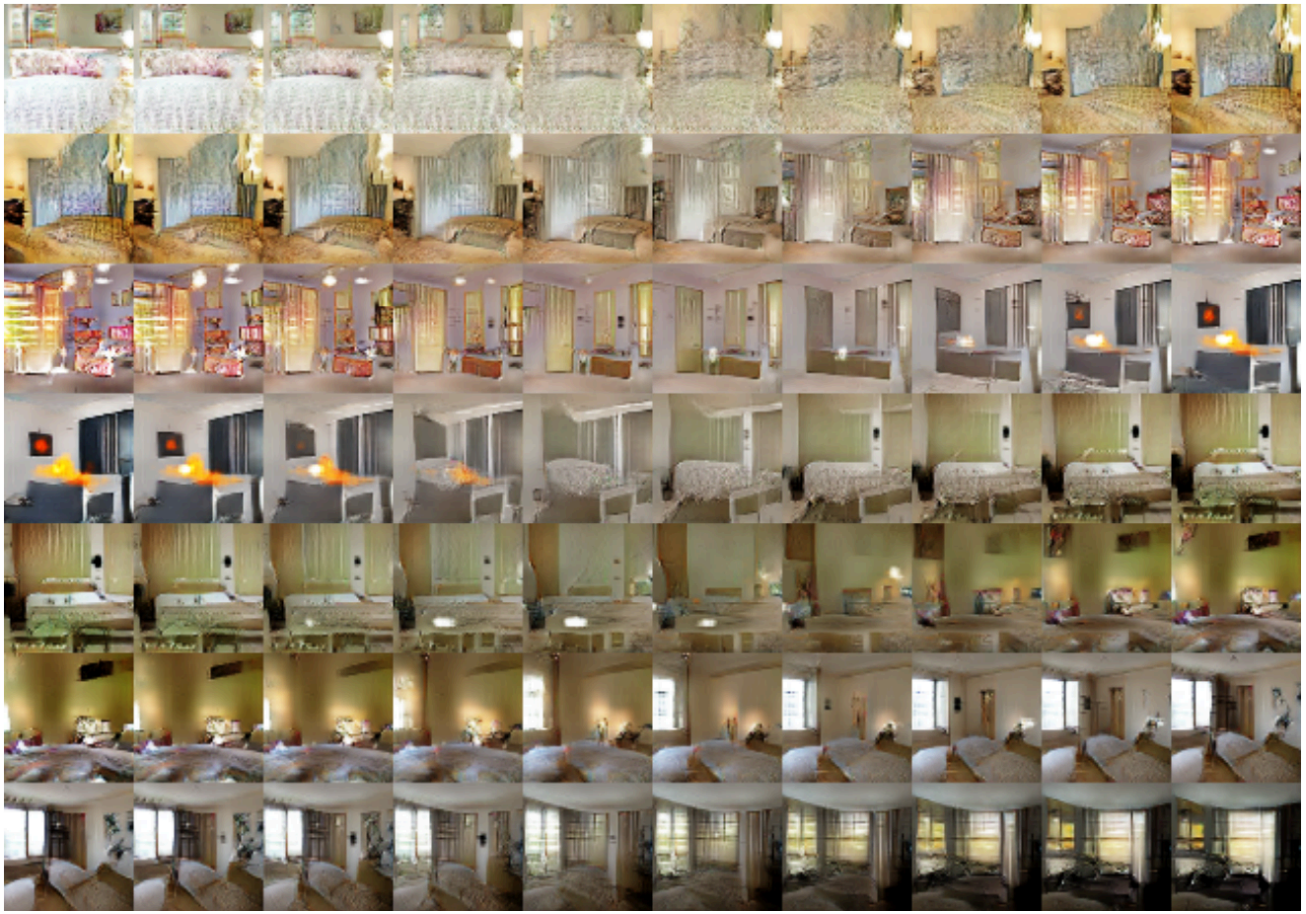
- No intermediate FC
- Have Dropout and Leaky ReLU
- Play more with ▸ code

DCGAN Results

- Mark an important milestone in GAN development
- Still a simple yet popular and widely-adopted one

Image credit: Alec Radford & Luke Metz & Soumith Chintala 2016

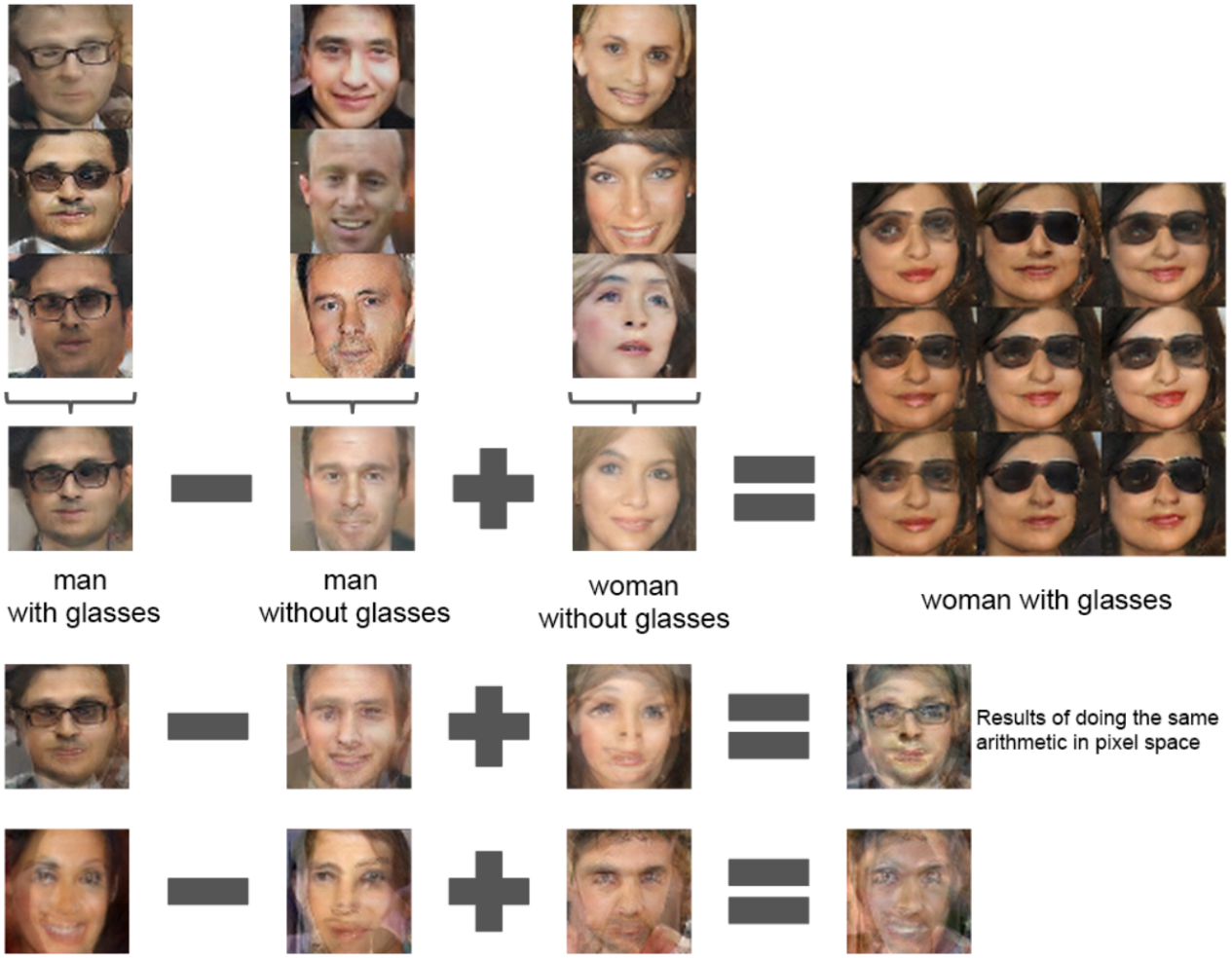
DCGAN Transition



- Transition is smooth
- *E.g.* 6th row: the TV changed smoothly to the window

Image credit: Alec Radford, Luke Metz and Soumith Chintala 2016

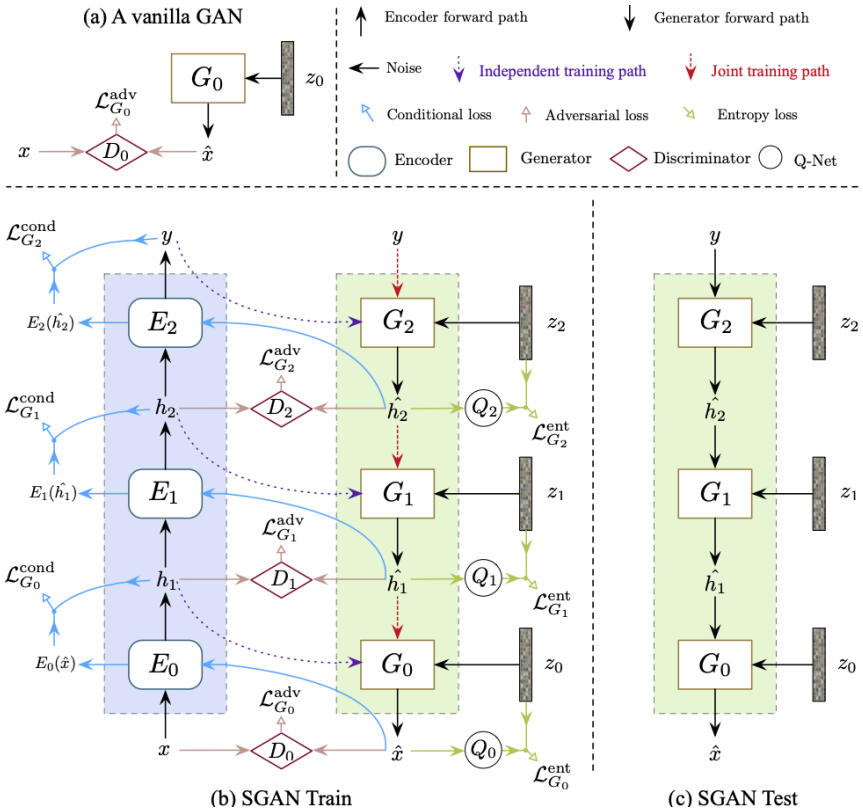
DCGAN Interpolation



- Similar to word2vec interpolation

Image credit: Alec Radford, Luke Metz and Soumith Chintala 2016

Stacked GAN (SGAN[2])



- Stack of GANs and help from stacked pre-trained Encoders
- Train them sequentially (hierarchically) and then train the whole

[2] https://arxiv.org/pdf/1612.04357.pdf

Image credit: Xun Huang *et al.* 2017

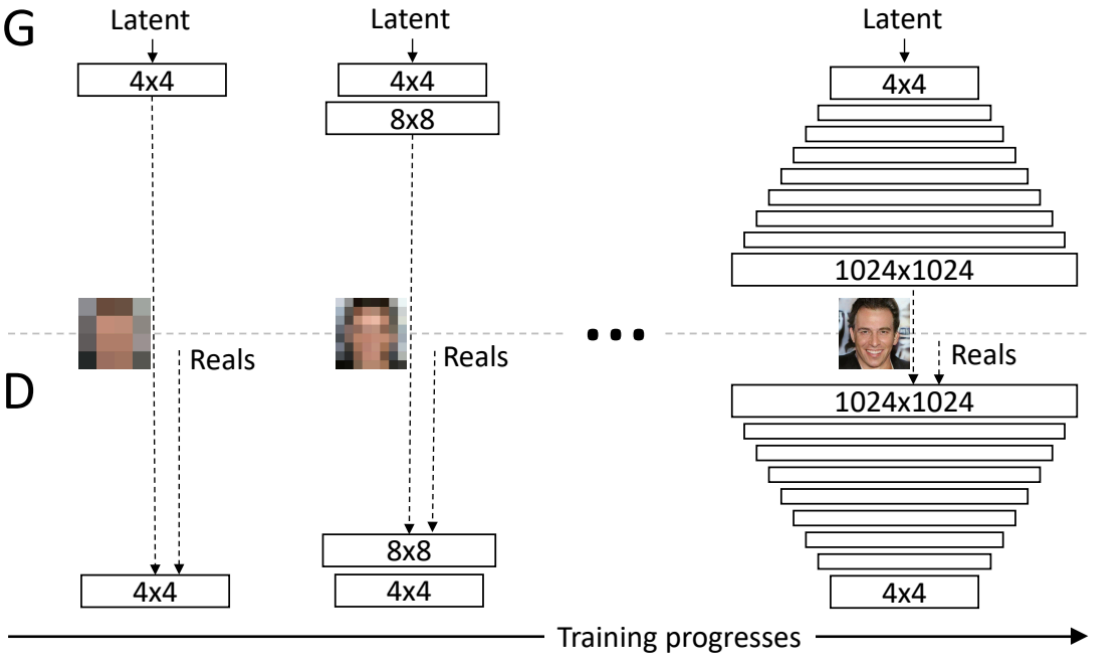
Stacked GAN (SGAN) Results

Carnegie Mellon

- Sharp and fine results

Image credit: Xun Huang *et al.* 2017

Progressive GAN[3]



- Different appoach from SGAN: train the whole model
- But improve the model over time → better resolution

[3] https://arxiv.org/pdf/1710.10196.pdf

Image credit: Karras *et al.* 2017

Progressive GAN

Figure 5: 1024 × 1024 images generated using the CELEBA-HQ dataset. See Appendix F for a larger set of results, and the accompanying video for latent space interpolations.

Image credit: Karras *et al.* 2017