

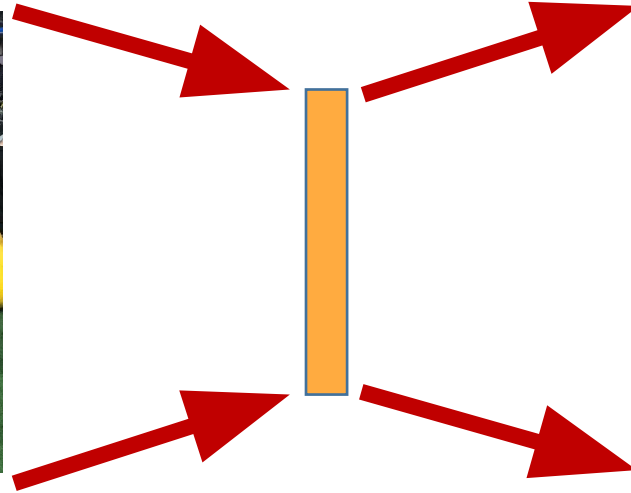
Generative Adversarial Networks

Andrew H. Fagg

Compressing Images



Original Image

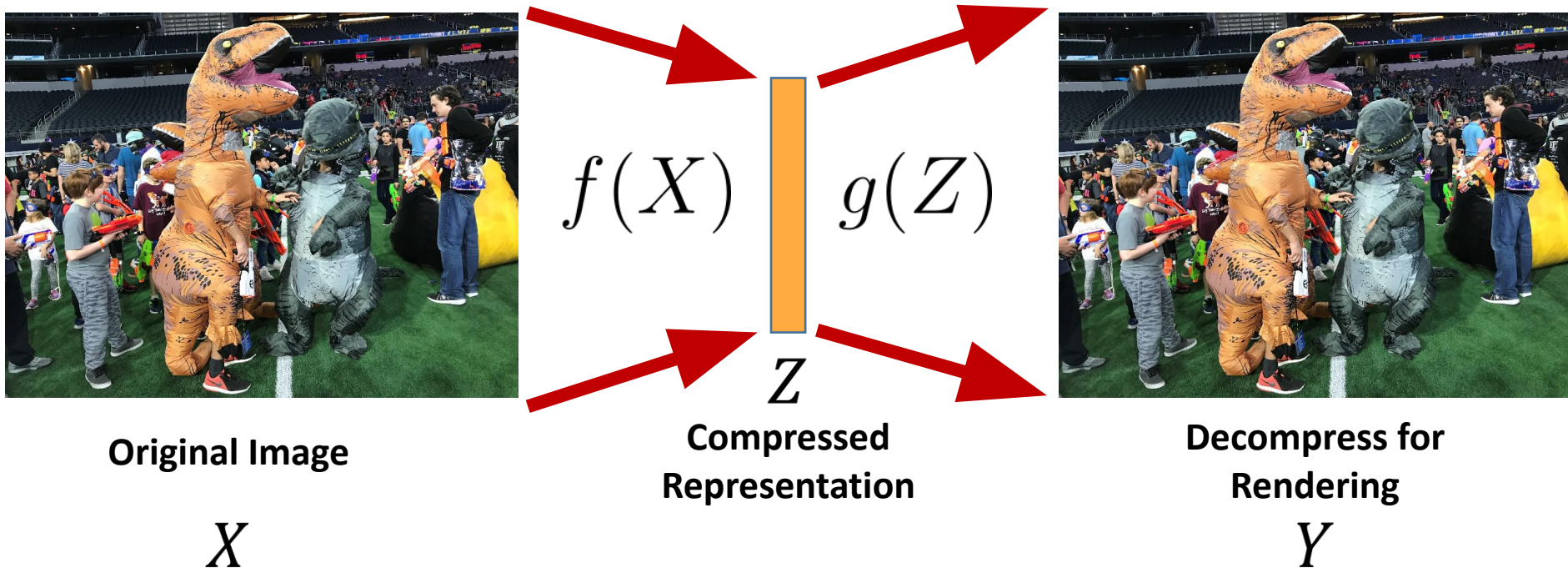


**Compressed
Representation**



**Decompress for
Rendering**

Compressing Images

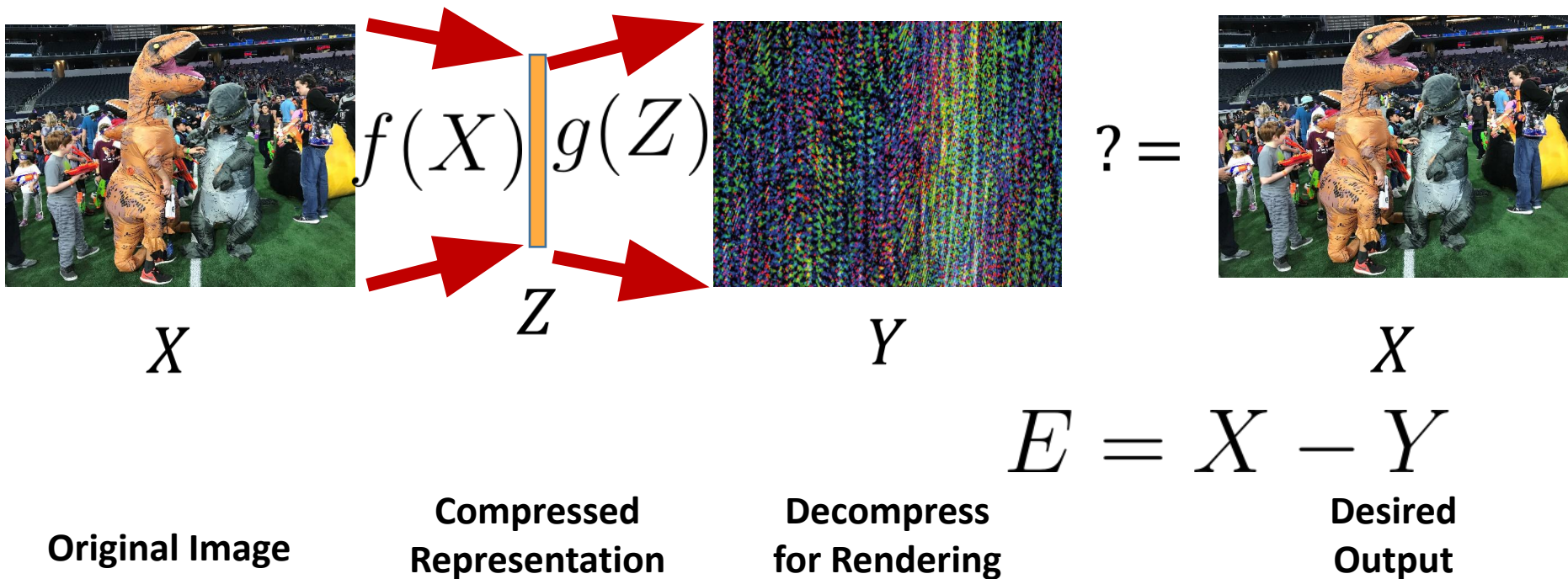


Compressing Images

- We know what X and Y should be
- But, the compressed representation (Z) does not contain any specific semantic information

Autoencoder idea: we can use our gradient descent method to learn these representations

Training Autoencoders for Compressing Images



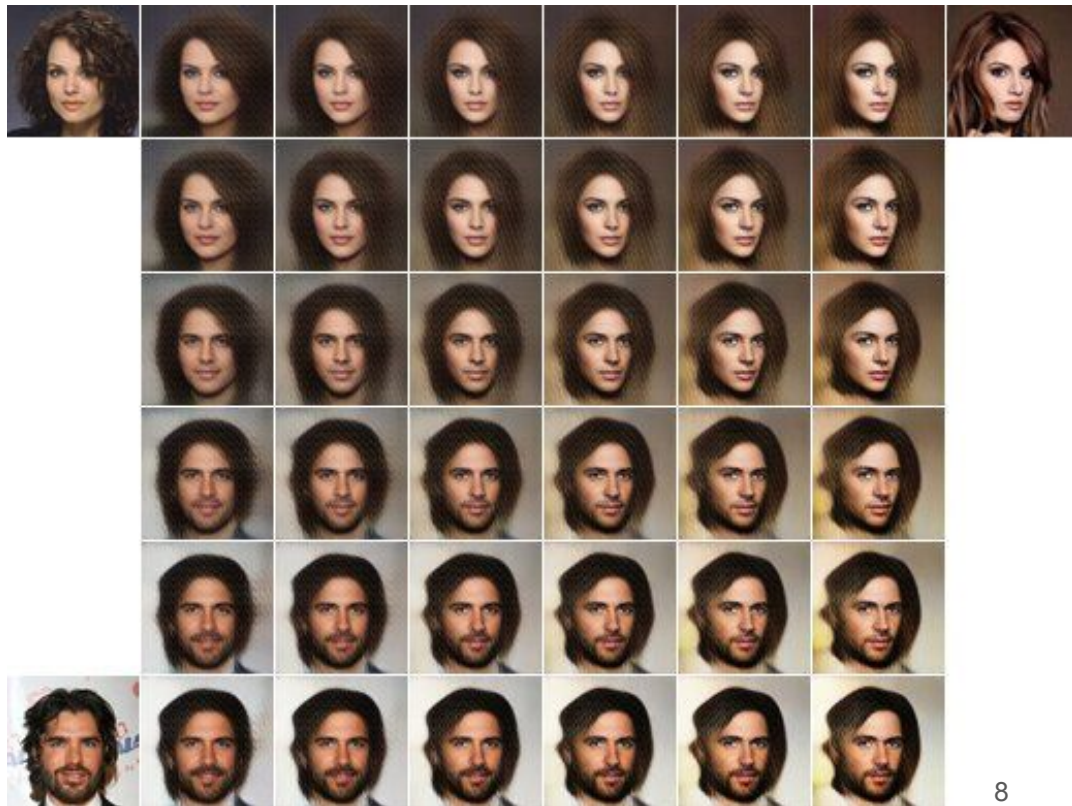
Training Autoencoders for Compressing Images

- Input and desired output are the same thing
 - This is a form of ***unsupervised learning***
 - Nobody determines explicitly what the compressed representation is (the algorithm does this!)
- With a large number of example images and sufficient compression:
- Compressed representation (called ***latent*** representation) begins to have some semantic meaning

Interpolation in Latent Space

Latent dimensions include:

- Head orientation
- Hair color
- Sex



Interpolation in Latent Space

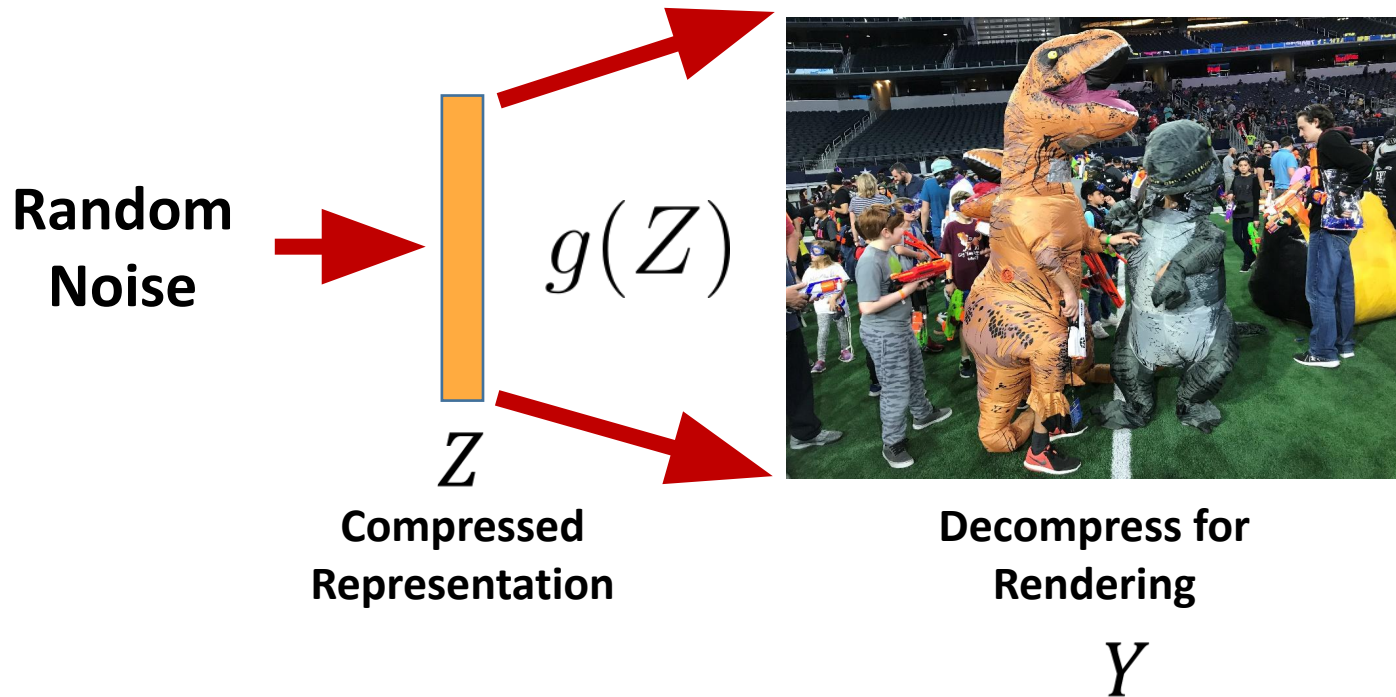
- A weighted average of two valid latent representations must also be a valid latent vector
 - I.E., the entire set of valid latent vectors must form a **compact set**
- When constructing autoencoders, we typically add regularization terms that require the latent representations fall within a Gaussian distribution

Thinking Bigger

- With autoencoders, our representation is limited to the set of examples that we used for training
- Yes, we can interpolate between these examples, but we want to be able to extrapolate, too

Goal: generate images that are realistic examples of scenes

Generating Images from Noise



Generator Evaluation

How might we evaluate the generated images?

Generator Evaluation

One idea:

- Learn another model that discriminates between real example images and the fake generated images
- This is a lot like our earlier image classifier

Discriminator



Input Image

$h(Y)$

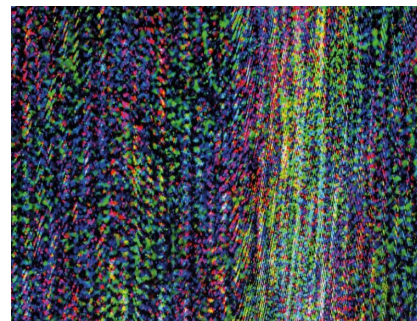


**Probability of
being real**

Discriminator



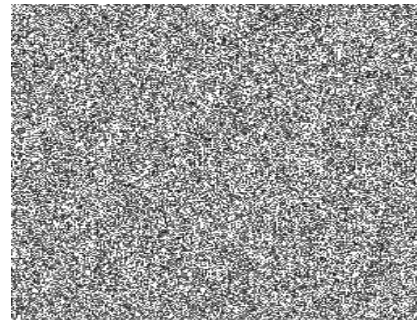
$h(Y)$
→ 0.97



$h(Y)$
→ 0.03



$h(Y)$
→ 0.89



$h(Y)$
→ 0.01

Full Architecture

What does it look like?

Full Architecture

What does it look like?

- Discriminator receives input from one of two sources:
 - Example real images
 - Images produced by the generator
- Learning:
 - Adjust discriminator parameters to better tell real from fake
 - Adjust generator parameters to better fool the discriminator with fakes
 - This sounds like a minimax problem!

Generative Adversarial Network

Discriminator
desired
output

0.0

1.0

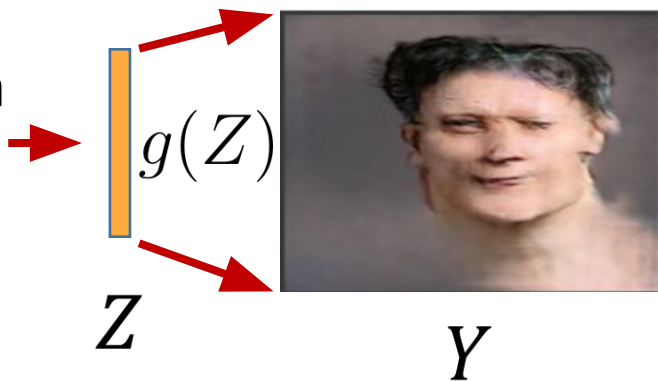
Probability
of being
real

OR

$h(Y)$

$$L_h = \text{crossentropy}(h(Y), D)$$

Random
Noise



Real
Image



Generative Adversarial Network

Generator
desired
output

1.0

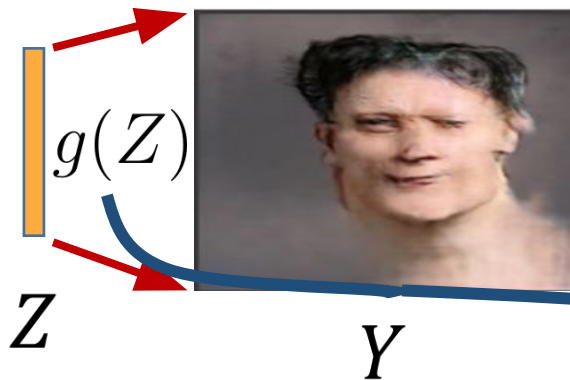
Probability
of being
real

X

OR

$$L_g = \text{crossentropy}(h(Y), D)$$

Random
Noise



Real
Image

Implementation Outline

```
// Create models  
h = create_discriminator()  
  
// Compile model with adjustable  
// parameters  
h.compile(loss='binary_crossentropy', ...)
```

Implementation Outline

```
// Create generator model  
g = build_generator()
```

```
// Future uses of h will not be trainable  
h.trainable = False
```

Implementation Outline

```
// Create the Meta-Model
Z = Input(shape=latent_shape)

// Create fake images
Y= g(Z)

// Apply discriminator to both image sets
p_fake = D(Y)

// Create the meta Model
model = Model(inputs=Z, outputs=p_fake, ...)

// Compile it
model.compile(loss='binary_crossentropy', ...)
```

Implementation Outline

Loop

```
z ~ sample_latent()
```

```
x ~ sample_real()
```

```
x_fake = g.predict(z)
```

```
model.fit(x=z, y=ones(), epochs=1)
```

```
d.fit(x=np.concat([x, x_fake]),  
      y=np.concat([ones(), zeros()]), epochs=1)
```


Adversarial Learning

The discriminator and generator are in an “arms race”:

- Early on, the generator does not produce interesting images.
- It is easy for the discriminator to do its job
- This gives the generator useful training information so it can produce better images
- In turn, the discriminator must catch up with the new generator
- Repeat

GAN Challenges

- Can take a long time to learn to generate even nominally interesting images
 - Especially when the generated images are large
- **Mode collapse:** no matter the randomly selected latent vector, the generator learns to ignore it and produce a single, realistic image
 - Can be a serious problem, especially if noise is injected only at the latent layer
 - Often introduce other regularization terms to force interesting variance

GAN Variations

- Cycle GANs
- Style GANs
- Conditional GANS
 - HW 7

Cycle GAN

Zhu et al., 2017

- Image to image translation
 - Translate an image in one domain into another domain
- We never have example image pairs (one for each domain)
 - Only singleton examples from each domain
- Approach:
 - Use discriminator for each domain to tell whether the translation was right
 - Use a U-Net to translate between domains

Monet $\leftrightarrow$ Photos



Monet $\rightarrow$ photo

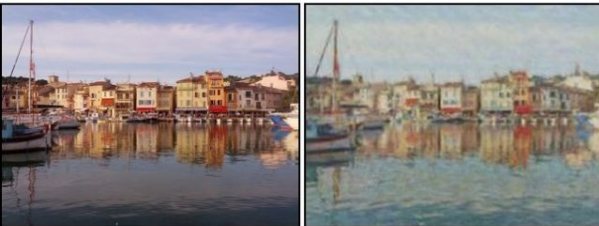
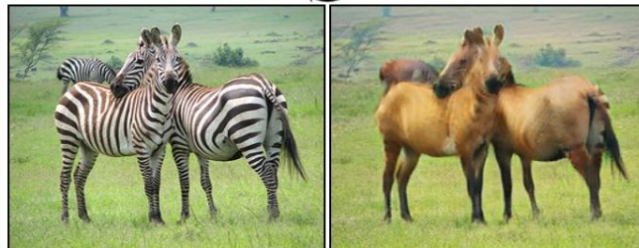


photo $\rightarrow$ Monet

Zebras $\leftrightarrow$ Horses



zebra $\rightarrow$ horse



horse $\rightarrow$ zebra

Summer $\leftrightarrow$ Winter



summer $\rightarrow$ winter



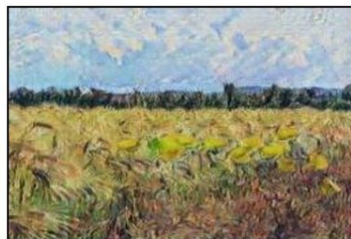
winter $\rightarrow$ summer



Photograph



Monet



Van Gogh



Cezanne



Ukiyo-e

Cycle GAN Implementation

- Must train the image translators and the discriminators at the same time
- Convenient to use Model nesting to make this work
- One tool:
 - `model.trainable` property (a Boolean) controls whether the parameters in the model can be adjusted
 - Catch: this property is **only** read by `model.compile()`

Cycle GAN Implementation

```
// Create new models
dA = create_discriminator()
dB = create_discriminator()

// Compile these models with adjustable
// parameters
dA.compile(loss='mse', ...)
dB.compile(loss='mse', ...)
```

Cycle GAN Implementation

```
// Create individual generator models  
gAB = build_generator()  
gBA = build_generator()
```

```
// Future uses of dA/dB will not be trainable  
dA.trainable = False  
dB.trainable = False
```

Cycle GAN Implementation

```
// Create the Meta-Model
inA = Input(shape=img_shape)
inB = Input(shape=img_shape)

// Create fake images
fakeA = gBA(inB)
fakeB = gAB(inA)

// Create duplicate images from the fakes
reconA = gBA(fakeB)
reconB = gAB(fakeA)

// Create image identities: don't change an image if it is
// already the right type
idA = gBA(inA)
idB = gAB(inB)
```

Cycle GAN Implementation

```
// Evaluate the fake images
validA = dA(fakeA)
validB = dB(fakeB)

// Create the meta Model
model = Model(inputs=[inA, inB]
               outputs=[validA, validB,
                        idA, idB,
                        reconA, reconB], ...)

// Compile it
model.compile(loss=['mse', 'mse',
                   'mae', 'mae',
                   'mae', 'mae'], ...)
```

Cycle GAN Implementation

```
// Train one batch
imgsA, imgsB are the batch

fakeA = gBA(imgsB)
fakeB = gAB(imgsA)

dA.fit(epochs=1, inputs=np.concatenate([imgsA, fakeA]),
        outputs=np.concatenate([1s, 0s]))

dB.fit(epochs=1, inputs=np.concatenate([imgsB, fakeB]),
        outputs=np.concatenate([1s, 0s]))

model.fit(epochs=1, inputs=[imgsA, imgsB]
           outputs=[1s, 1s,
                    imgsA, imgsB,
                    imgsA, imgsB], ...)
```


Style GAN

Two source images:

- Content: Goal is to create an image that maintains the detailed structure of the input image (e.g., where are the edges and other texture? What shapes are there?)
- Style: Goal is to create an image that tries to capture “style” elements in the image (higher-level features)
 - Color
 - Larger shapes and their spatial relationships
 -



