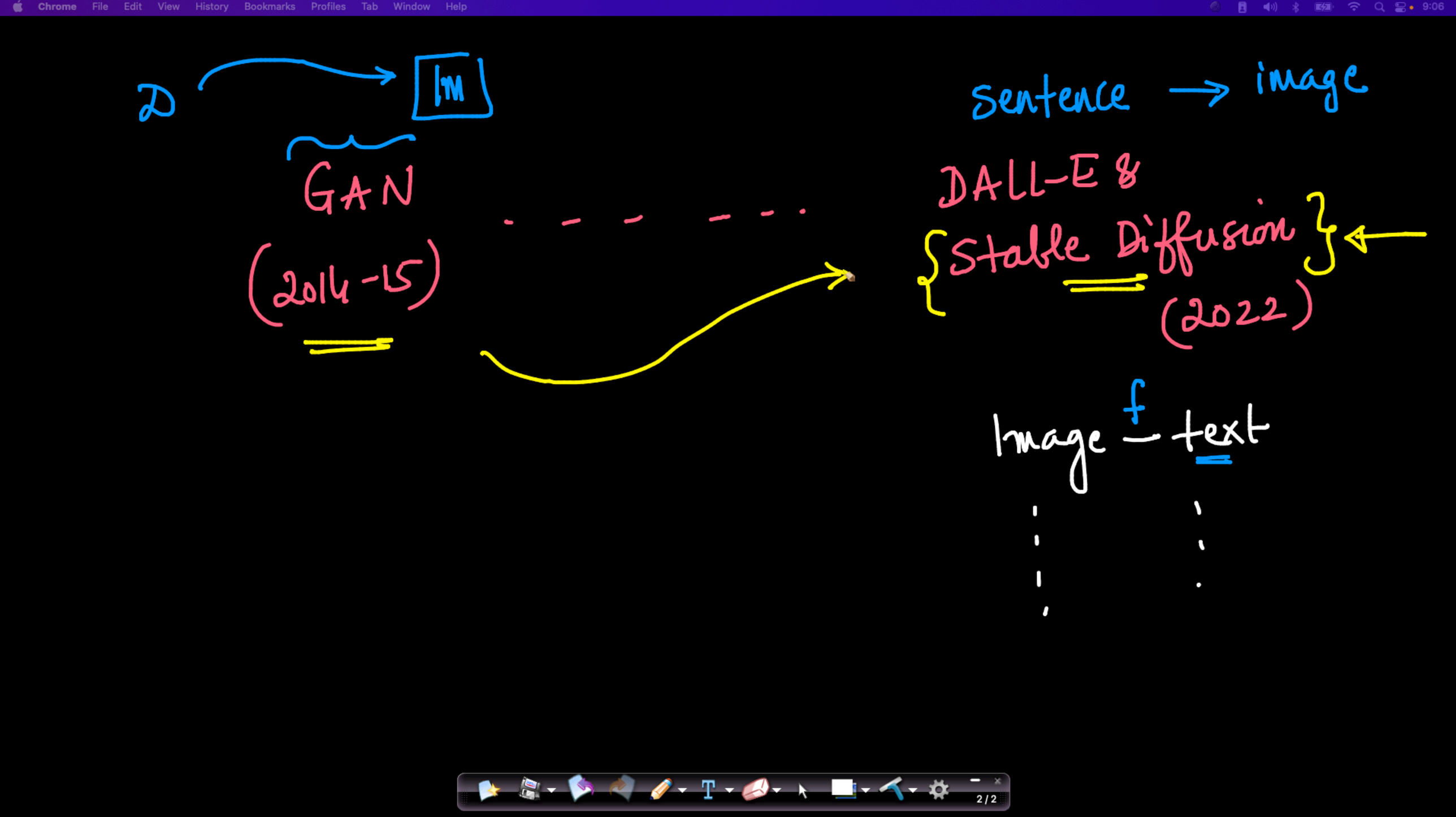
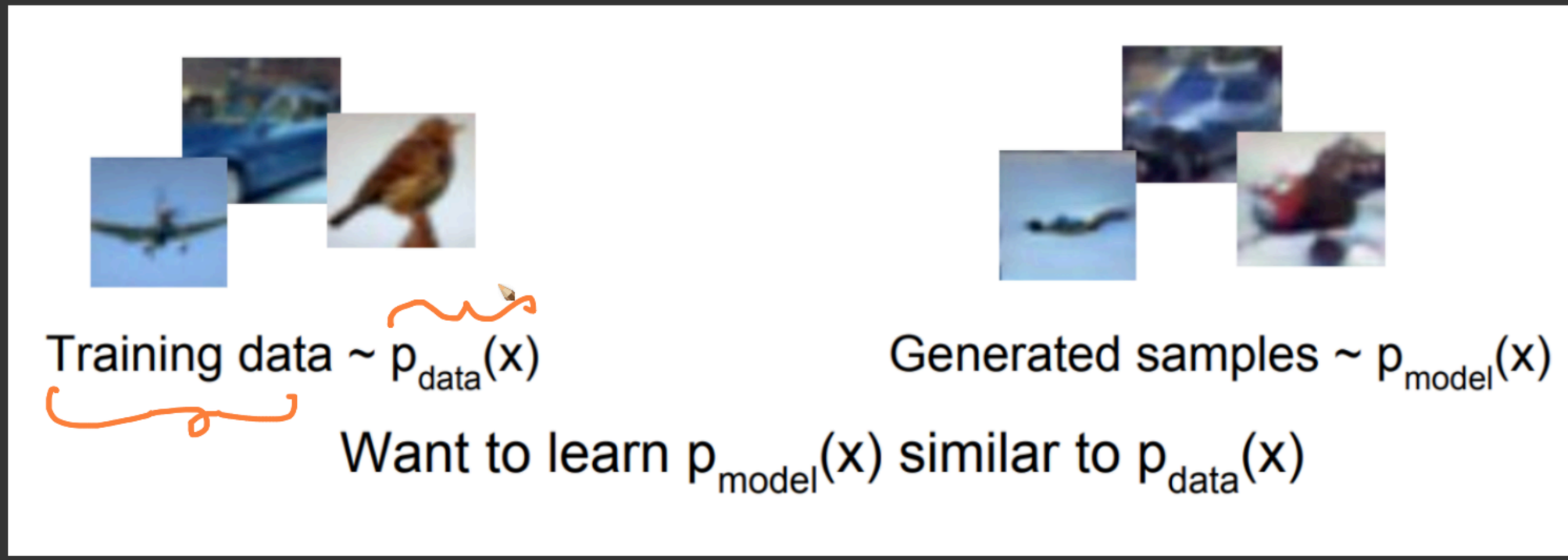
Agenda:

CV

1. Image-generation Problem → "anime"
2. GAN [2014-15] → architecture; Math; code from scratch
   [Generative Adversarial Network]
3. Limitations of GANs
4. Variations → Cycle GAN; SR-GAN; StyleGAN
   (2017-18) (overviews)


D
Im
GAN
(2014-15)
Sentence → image
DALL-E &
{Stable Diffusion}
(2022)
Image – text

- **More concretely, these models learn the distribution of the given training data & it generates new samples of data of the same distribution.**

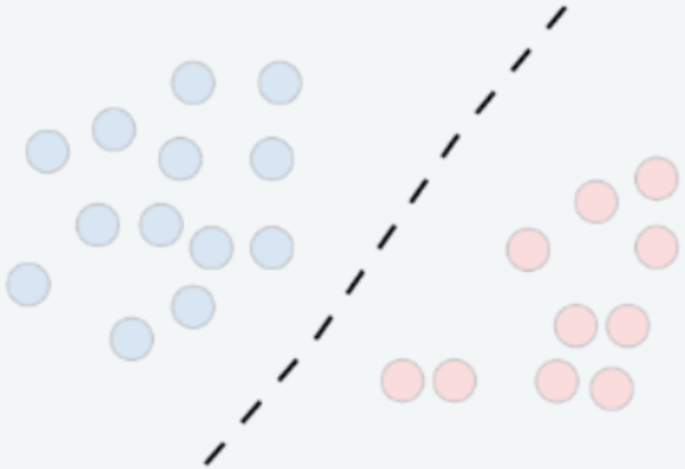
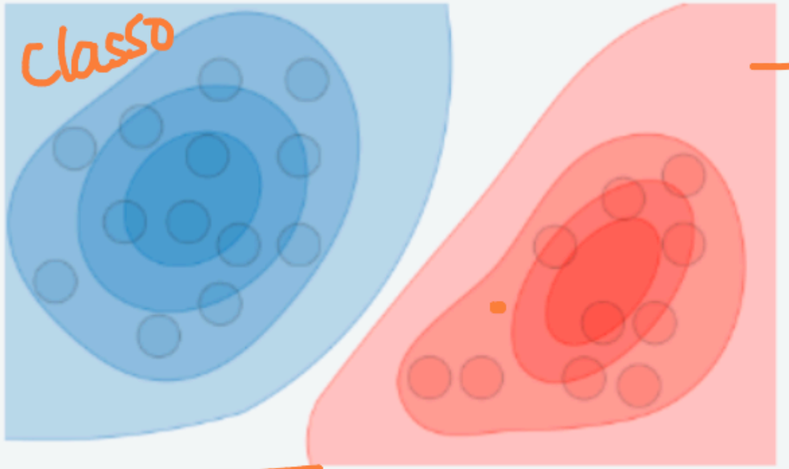


Discriminative vs Generative models?

- Discriminative models draw boundaries in the data space, while generative models try to model how data is placed throughout the space.
- A generative model focuses on explaining how the data was generated, while a discriminative model focuses on predicting the labels of the data.

Discriminative vs Generative models?

- Discriminative models draw boundaries in the data space, while generative models try to model how data is placed throughout the space.
- A generative model focuses on explaining how the data was generated, while a discriminative model focuses on predicting the labels of the data.

	Discriminative model	Generative model
Goal	Directly estimate $P(y\|x)$	Estimate $P(x\|y)$ to then deduce $P(y\|x)$
What's learned	Decision boundary	Probability distributions of the data
Illustration		
Examples	Regressions, SVMs	GDA, Naive Bayes

GMM

1

Bayes Thm

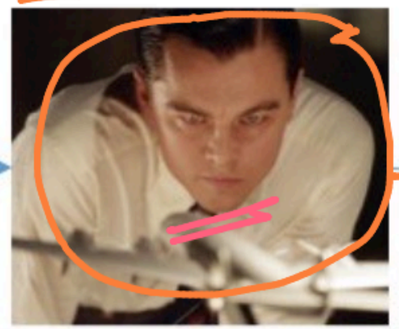
→ class 1

Random Input

random vector
d-dim
128-dim (let)

Manipulator or 'Generator Network'

G



{ Generative Adversarial N/W

D

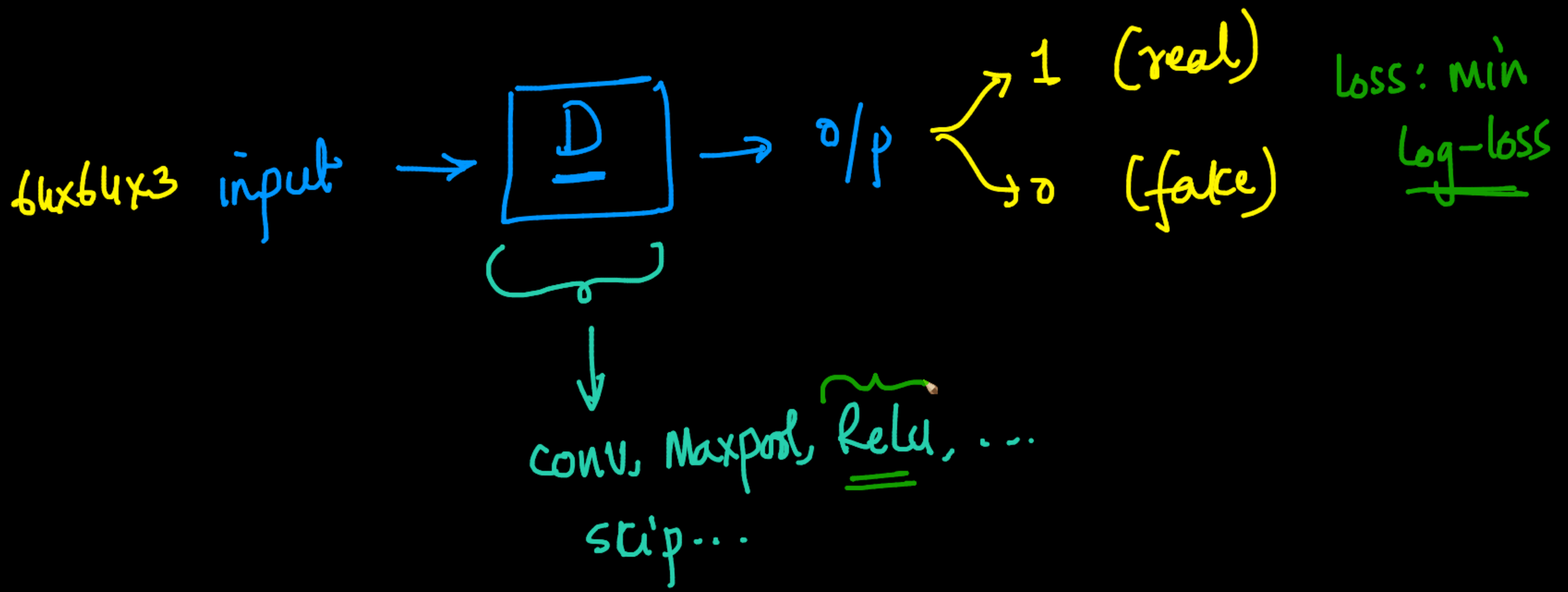
Fake

Real

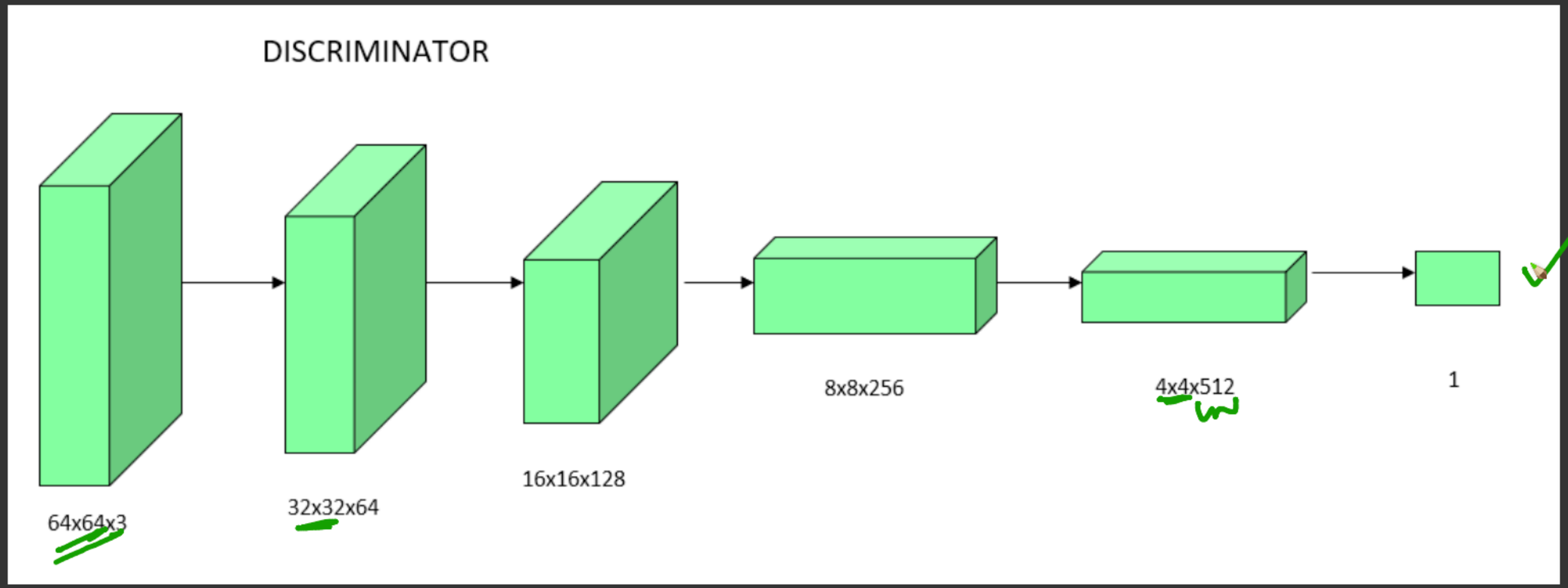
Enforcer or 'Discriminator Network'

Real Currency

G & D

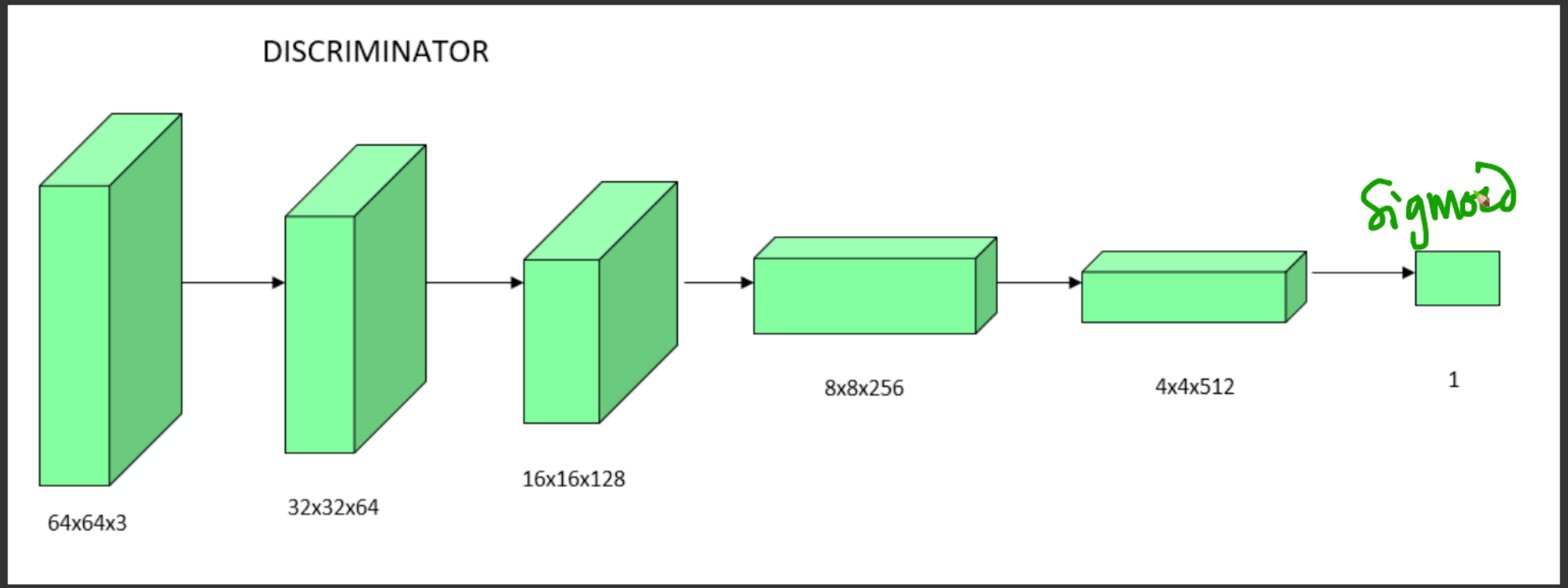


Architecture of Discriminator




```
[ ]  def get_disc_normal(image_shape=(64,64,3)):
         image_shape = image_shape

         dropout_prob = 0.4
```

Architecture of Discriminator




```
[ ]  def get_disc_normal(image_shape=(64,64,3)):
         image_shape = image_shape

         dropout_prob = 0.4
```

Generator

Train is tricky

i/p

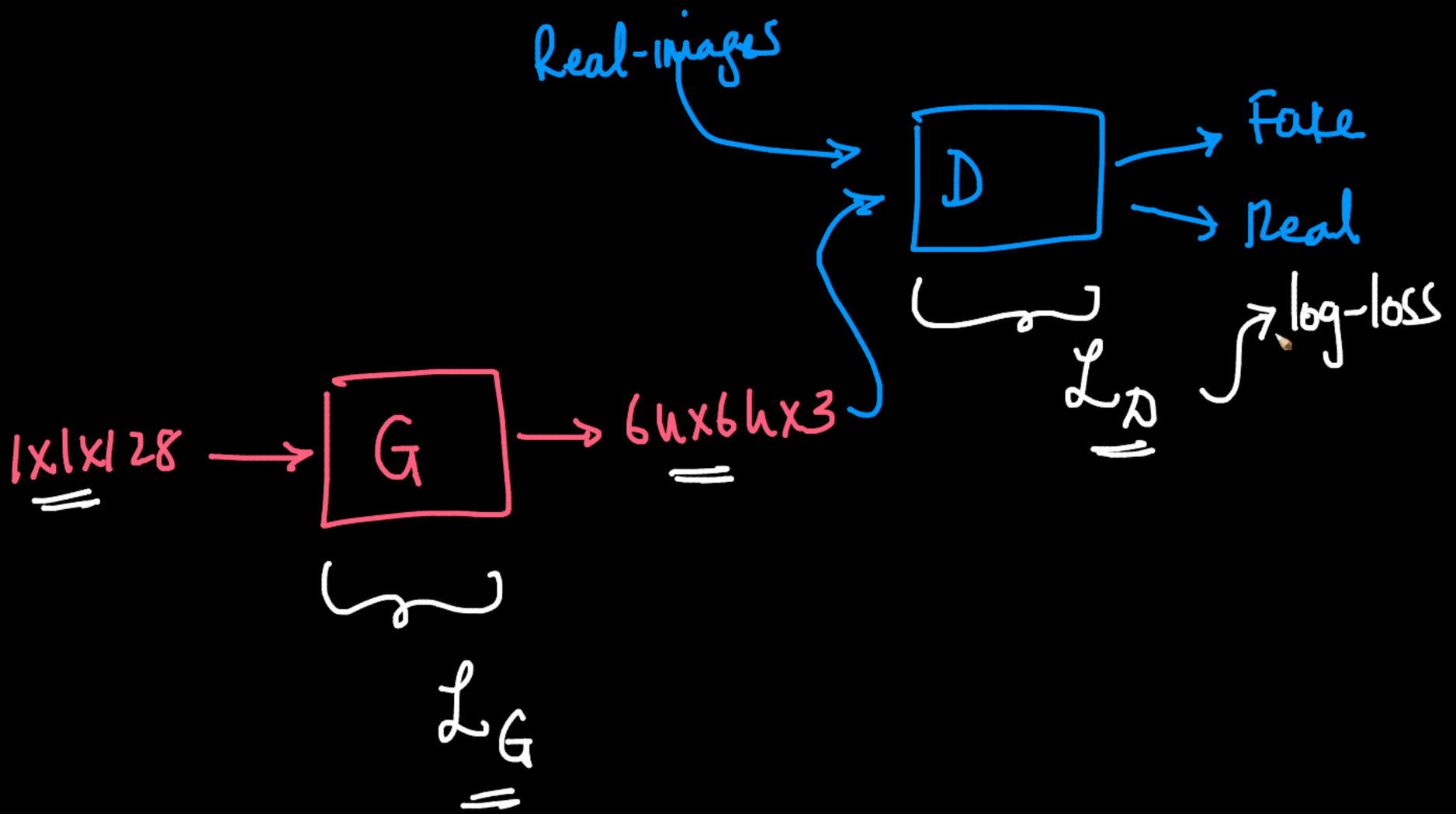
G
CNN

(anime)
o/p

64x64x3

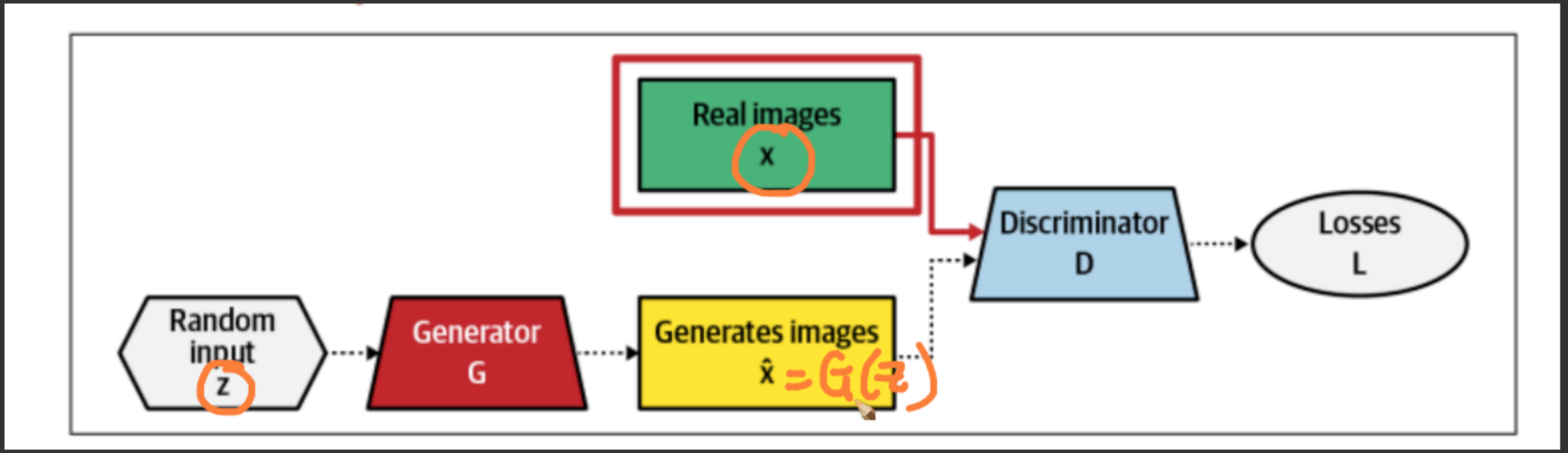
✓ Transposed Conv (FCN)

Upsampling 2D (U-nets)

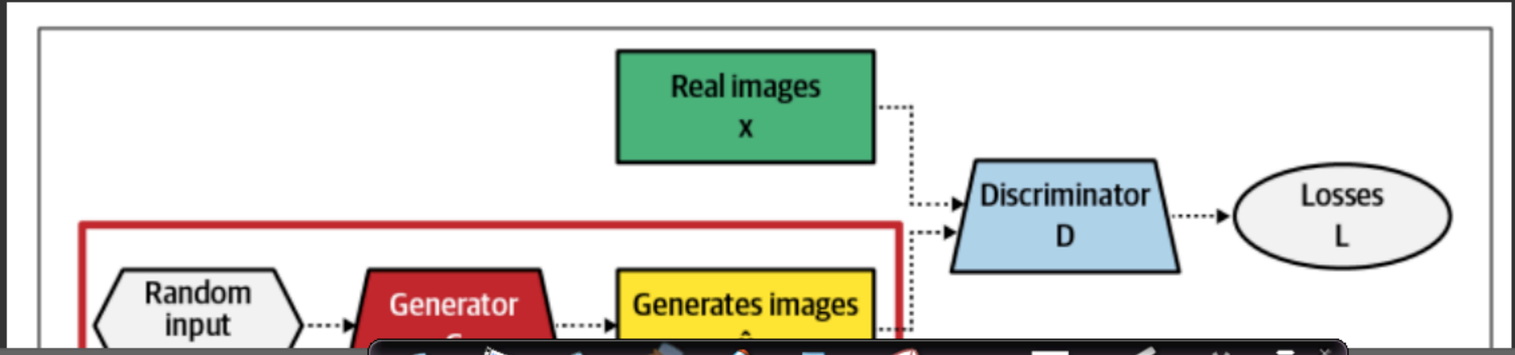

Real-images
D
Fake
Real
log-loss
$\mathcal{L}_D$
1x1x128
G
64x64x3
$\mathcal{L}_G$

Here are the steps involved in training the discriminator:

- We expect the discriminator to output 1 if the image was picked from the real Anime Face dataset, and 0 if it was generated using the generator network.
- We first pass a batch of real images, and compute the loss, setting the target labels to 1.

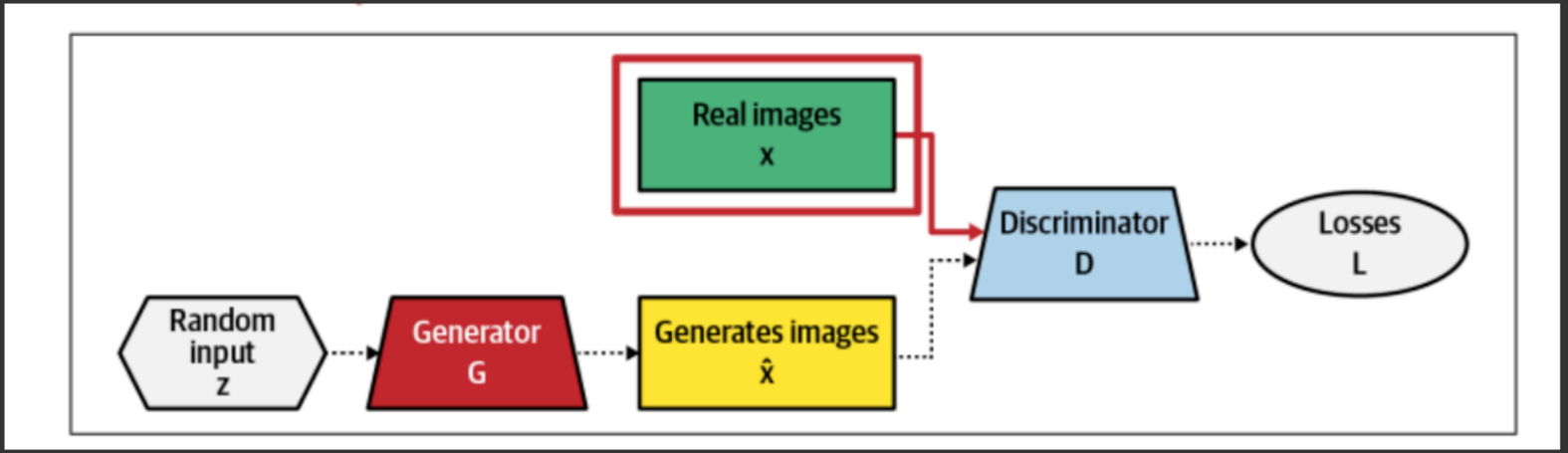


- Then we pass a batch of fake images of anime (generated using the generator) pass them into the discriminator, and compute the loss, setting the target labels to 0.

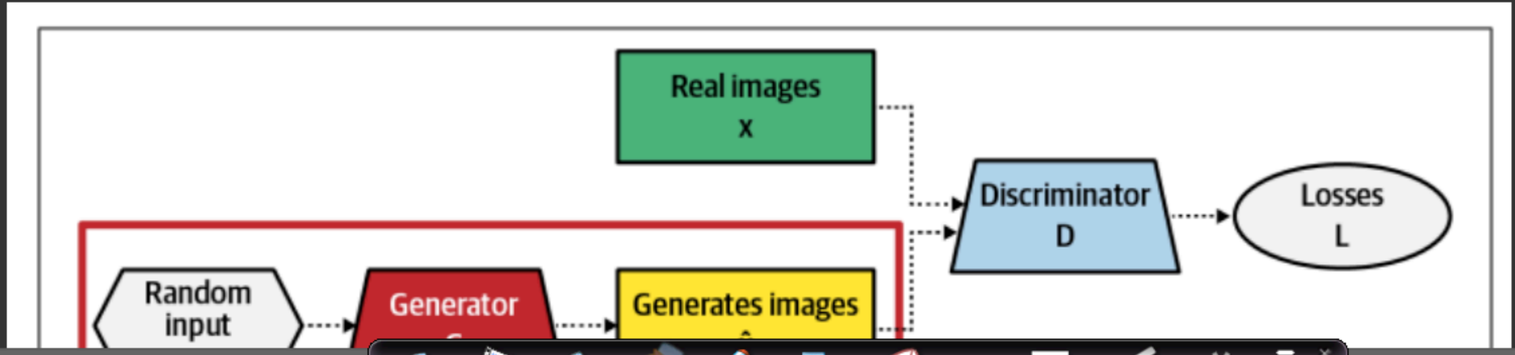


Here are the steps involved in training the discriminator:

- We expect the discriminator to output 1 if the image was picked from the real Anime Face dataset, and 0 if it was generated using the generator network.
- We first pass a batch of real images, and compute the loss, setting the target labels to 1.

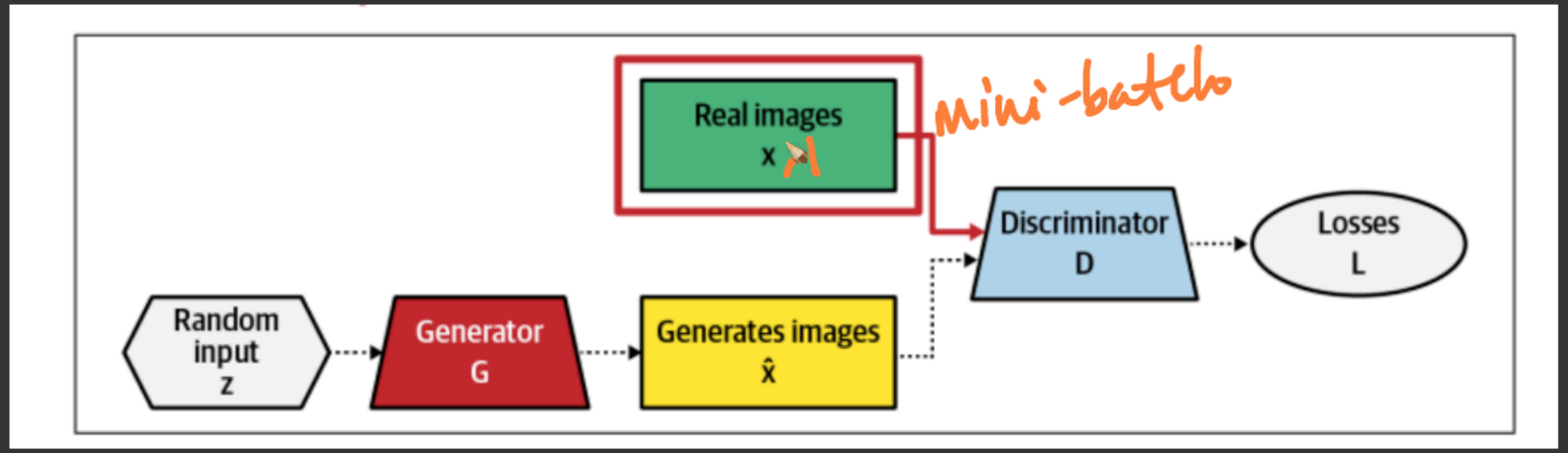


- Then we pass a batch of fake images of anime (generated using the generator) pass them into the discriminator, and compute the loss, setting the target labels to 0.

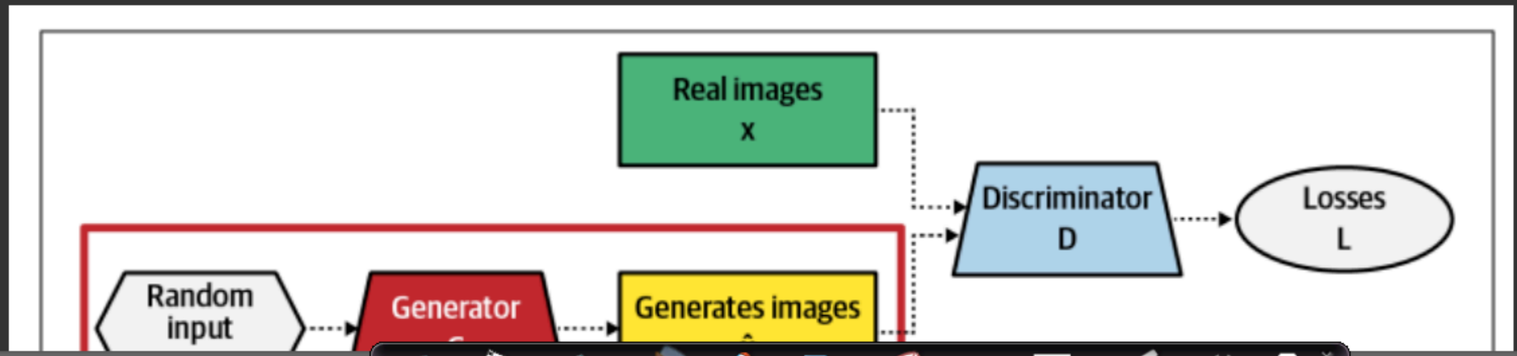


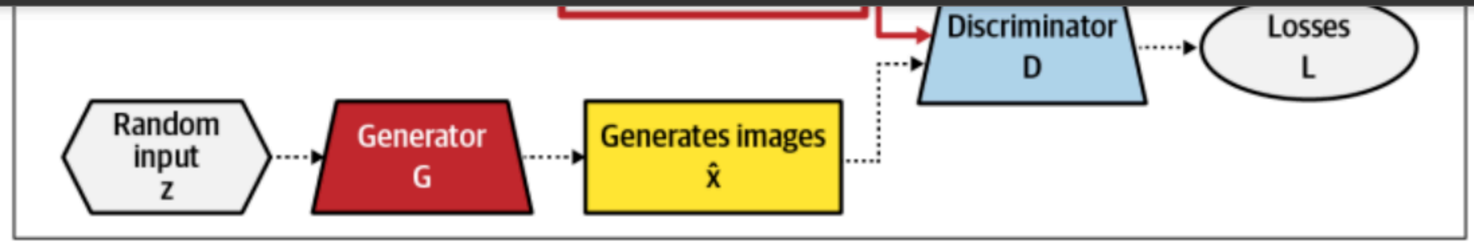
Here are the steps involved in training the discriminator:

- We expect the discriminator to output 1 if the image was picked from the real Anime Face dataset, and 0 if it was generated using the generator network.
- We first pass a batch of real images, and compute the loss, setting the target labels to 1.

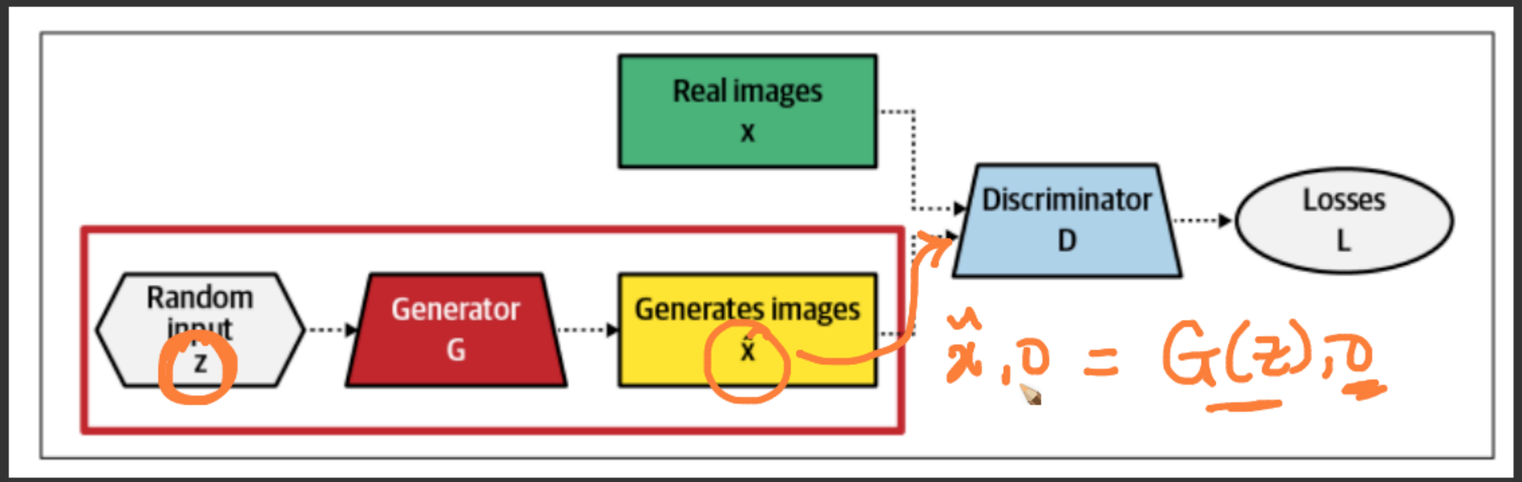


- Then we pass a batch of fake images of anime (generated using the generator) pass them into the discriminator, and compute the loss, setting the target labels to 0.

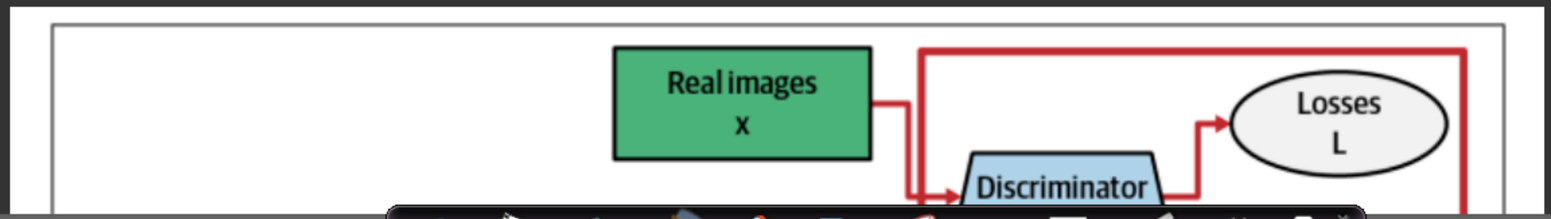




- Then we pass a batch of fake images of anime (generated using the generator) pass them into the discriminator, and compute the loss, setting the target labels to 0.

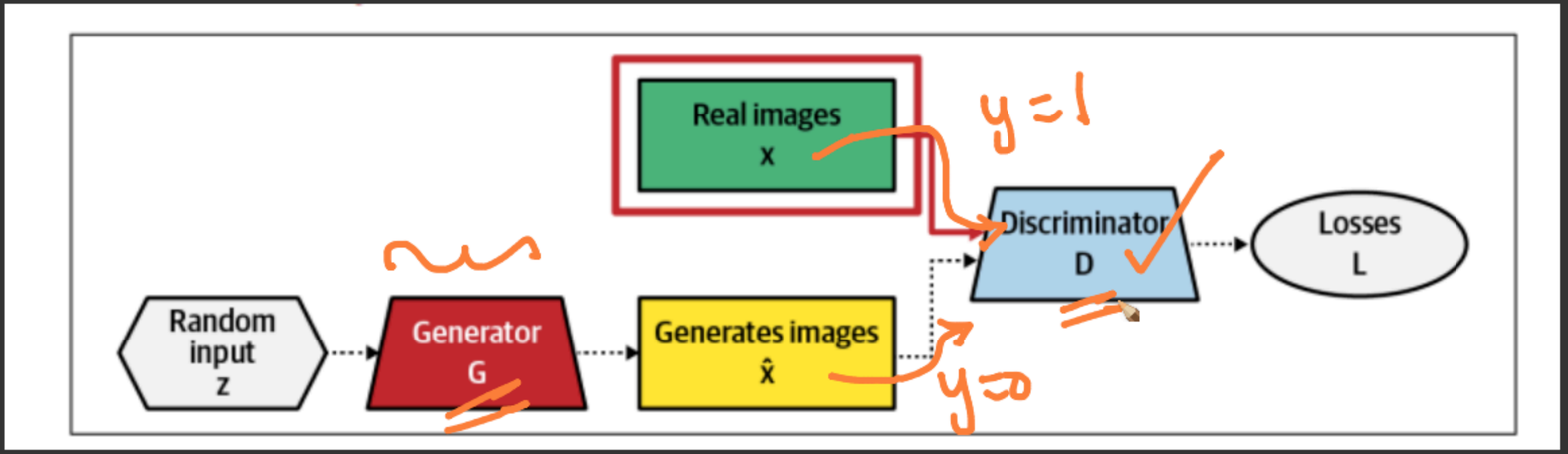


- Finally we add the two losses and use the overall loss to perform gradient descent to adjust the weights of the discriminator.

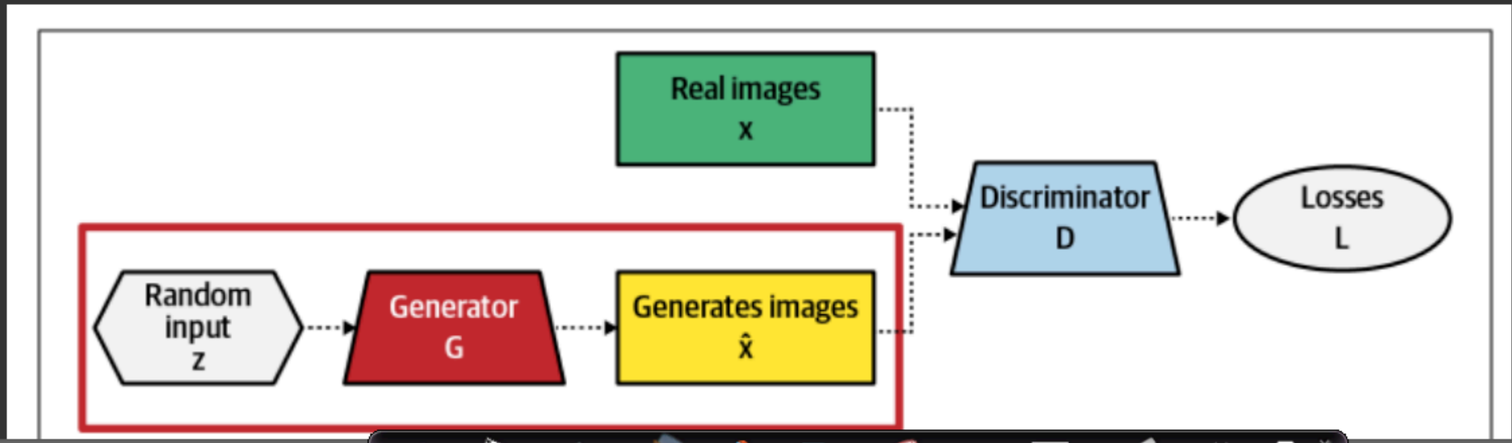


generator network.

- We first pass a batch of real images, and compute the loss, setting the target labels to 1.



- Then we pass a batch of fake images of anime (generated using the generator) pass them into the discriminator, and compute the loss, setting the target labels to 0.



1. Given, real image as input X , the predicted value is $D(X)$ and the label is 1. Therefore the loss function is,
   - $L(D(X), 1) = 1.logD(X) + (1-1).log(1-D(X)) = logD(X)$
   - The discriminator should maximize Log(D(X)), and as Log is a monotonic function so log(D(X)) will automatically get maximized if the discriminator maximized D(X).
2. Given, $G(Z)$ as input to Discriminator, the predicted value is $D(G(Z))$ and the label is 0. Therefore the loss function is,
   - $L(D(G(Z)), 0) = 0.logD(G(Z)) + (1-0).log(1-D(G(Z))) = log(1-D(G(Z)))$
   - The discriminator needs to maximize log(1 − D(G(Z))), which means it must have to minimize D(G(Z)).

- So, the loss function for the discriminator (for a single sample) becomes,

$$max[logD(X) + log(1 - D(G(Z)))]$$

- Now, the loss function of the discriminator over a batch is,

$$max[E_{(X\sim P(X))}[logD(X)] + E_{Z\sim P(Z)}[log(1-D(G(Z)))]]$$

where,
P(X) is the probability distribution of real training data
P(Z) is the probability distribution of the noise vector Z.
Typically, P(Z) is gaussian or Uniform.

- **The following equation is maximized for training the discriminator**

$$\max_D \frac{1}{m} \sum_{i=1}^{m} [logD(x^i) + log(1 - D(G(z^i)))].$$

x: real
z: random
G(z): Gen-image
D(G(z)): prob of real

Discriminator Training

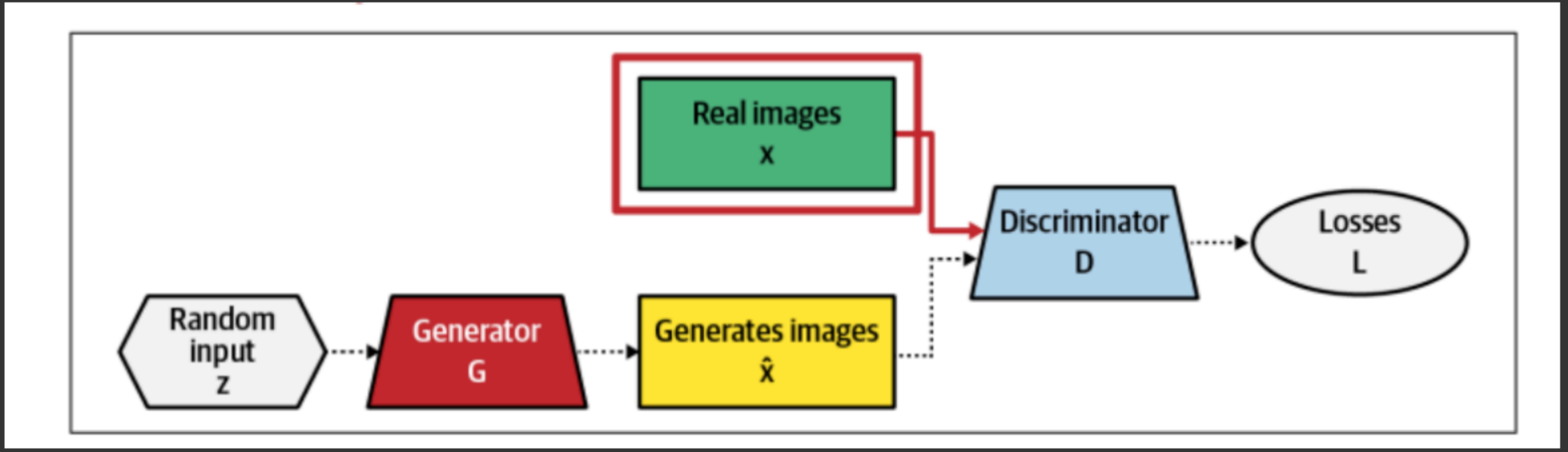
- Since the discriminator is a binary classification model, we can use the `binary cross entropy loss function` to quantify how well it is able to differentiate between real and generated images.
- The binary cross entropy loss is defined as:

$$-\frac{1}{N}\sum_{i=1}^{N} y_i log(h_\theta(x_i)) + (1 - y_i) log(1 - h_\theta(x_i))$$

min = Max + ...

Here are the steps involved in training the discriminator:

- We expect the discriminator to output 1 if the image was picked from the real Anime Face dataset, and 0 if it was generated using the generator network.
- We first pass a batch of real images, and compute the loss, setting the target labels to 1.



- $L(D(G(Z)), 0) = 0.logD(G(Z)) + (1-0).log(1 - D(G(Z))) = log(1 - D(G(Z)))$
- The discriminator needs to maximize log(1 − D(G(Z))), which means it must have to minimize D(G(Z)).

- So, the loss function for the discriminator (for a single sample) becomes,

$$max[logD(X) + log(1 - D(G(Z)))]$$

- Now, the loss function of the discriminator over a batch is,

$$max[E_{(X \sim P(X))}[logD(X)] + E_{Z \sim P(Z)}[log(1 - D(G(Z)))]]$$

where,
P(X) is the probability distribution of real training data
P(Z) is the probability distribution of the noise vector Z.
Typically, P(Z) is gaussian or Uniform.

- **The following equation is maximized for training the discriminator**

$$\max_D \frac{1}{m} \sum_{i=1}^{m} [logD(x^i) + log(1 - D(G(z^i)))].$$

> **Note:** that, D(X) and D(G(Z)) both are probability values and **both of them lie in between 0 and 1.**

Generator Training

- Since the outputs of the generator are images, it's not obvious how we can train the generator.

- $L(D(G(Z)), 0) = 0.logD(G(Z)) + (1-0).log(1-D(G(Z))) = log(1-D(G(Z)))$
- The discriminator needs to maximize log(1 − D(G(Z))), which means it must have to minimize D(G(Z)).

- So, the loss function for the discriminator (for a single sample) becomes,

> $max[logD(X) + log(1-D(G(Z)))]$

+ve ∴ log-loss

- Now, the loss function of the discriminator over a batch is,

> $max[E_{(X\sim P(X))}[logD(X)] + E_{Z\sim P(Z)}[log(1-D(G(Z)))]]$

where,
P(X) is the probability distribution of real training data
P(Z) is the probability distribution of the noise vector Z.
Typically, P(Z) is gaussian or Uniform.

- **The following equation is maximized for training the discriminator**

> $\max_D \frac{1}{m} \sum_{i=1}^{m} [logD(x^i) + log(1-D(G(z^i)))].$

> **Note:** that, D(X) and D(G(Z)) both are probability values and **both of them lie in between 0 and 1.**

Generator Training

- Since the outputs of the generator are images, it's not obvious how we can train the generator.

- $L(D(G(Z)), 0) = 0.logD(G(Z)) + (1-0).log(1-D(G(Z))) = log(1-D(G(Z)))$
- The discriminator needs to maximize log(1 − D(G(Z))), which means it must have to minimize D(G(Z)).

- So, the loss function for the discriminator (for a single sample) becomes,

$$max[logD(X) + log(1 - D(G(Z)))]$$

- Now, the loss function of the discriminator over a batch is,

$$max[E_{(X\sim P(X))}[logD(X)] + E_{Z\sim P(Z)}[log(1 - D(G(Z)))]]$$

where,
P(X) is the probability distribution of real training data
P(Z) is the probability distribution of the noise vector Z.
Typically, P(Z) is gaussian or Uniform.

- **The following equation is maximized for training the discriminator**

$$\max_D \frac{1}{m} \sum_{i=1}^{m} [logD(x^i) + log(1 - D(G(z^i)))].$$

Note: that, D(X) and D(G(Z)) both are probability values and **both of them lie in between 0 and 1.**

Generator Training

- Since the outputs of the generator are images, it's not obvious how we can train the generator.

$$max[E_{(X\sim P(X))}[logD(X)] + E_{Z\sim P(Z)}[log(1 - D(G(Z)))]]$$ ← theory

where,
P(X) is the probability distribution of real training data
P(Z) is the probability distribution of the noise vector Z.
Typically, P(Z) is gaussian or Uniform.

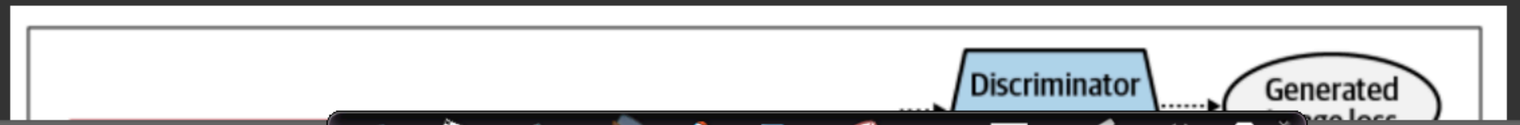
- **The following equation is maximized for training the discriminator**

$$\max_D \frac{1}{m} \sum_{i=1}^{m} [logD(x^i) + log(1 - D(G(z^i)))].$$

Note: that, D(X) and D(G(Z)) both are probability values and **both of them lie in between 0 and 1.**

Generator Training

- Since the outputs of the generator are images, it's not obvious how we can train the generator.
- This is where we employ a rather elegant trick, which is to use the discriminator as a part of the loss function.
- Here's how it works:
  - We generate a batch of images using the generator, pass the into the discriminator.



Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

4.1 Global Optimality of $p_g = p_{\text{data}}$

We first consider

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

4.1 Global Optimality of $p_g = p_{\text{data}}$

We first consider

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

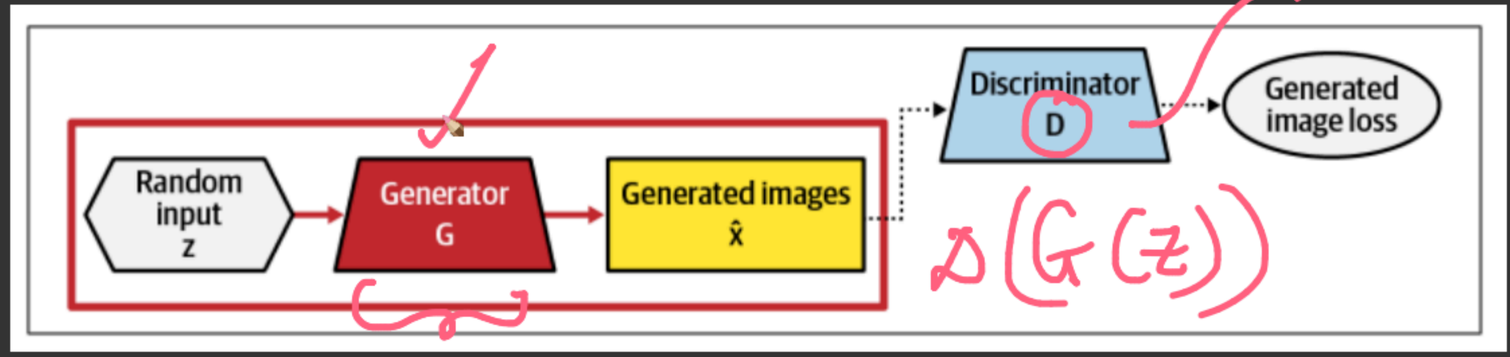
end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

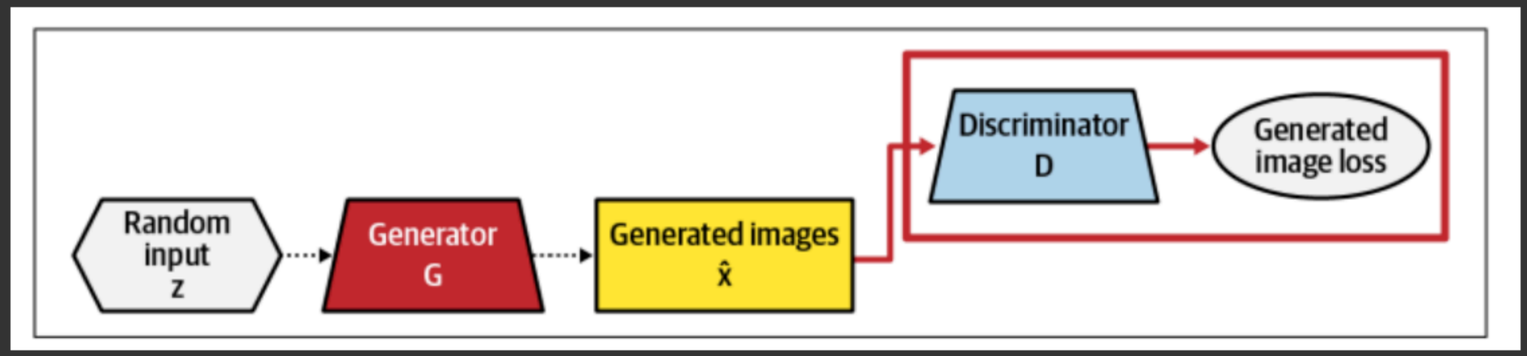
4.1 Global Optimality of $p_g = p_{\text{data}}$

We first consider

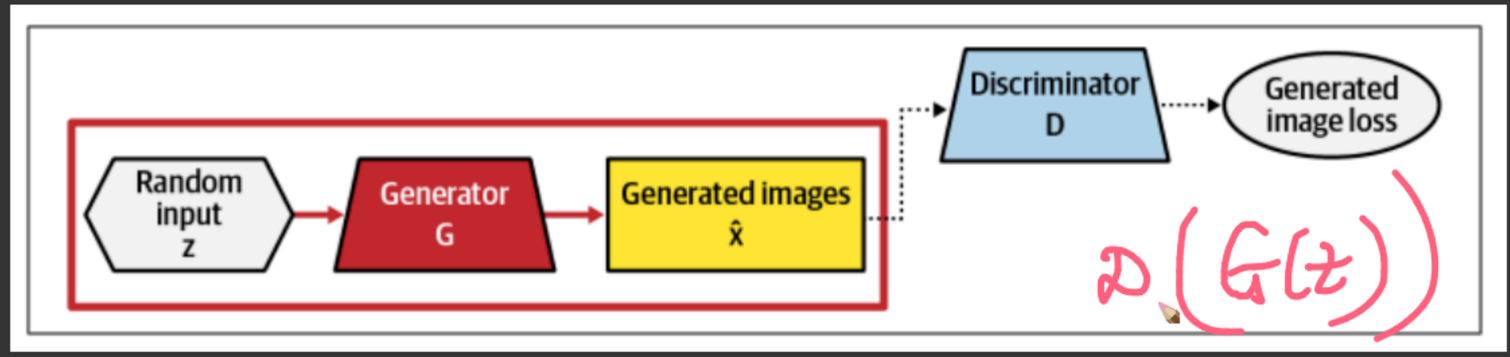
- Here's how it works:
  - We generate a batch of images using the generator, pass the into the discriminator.



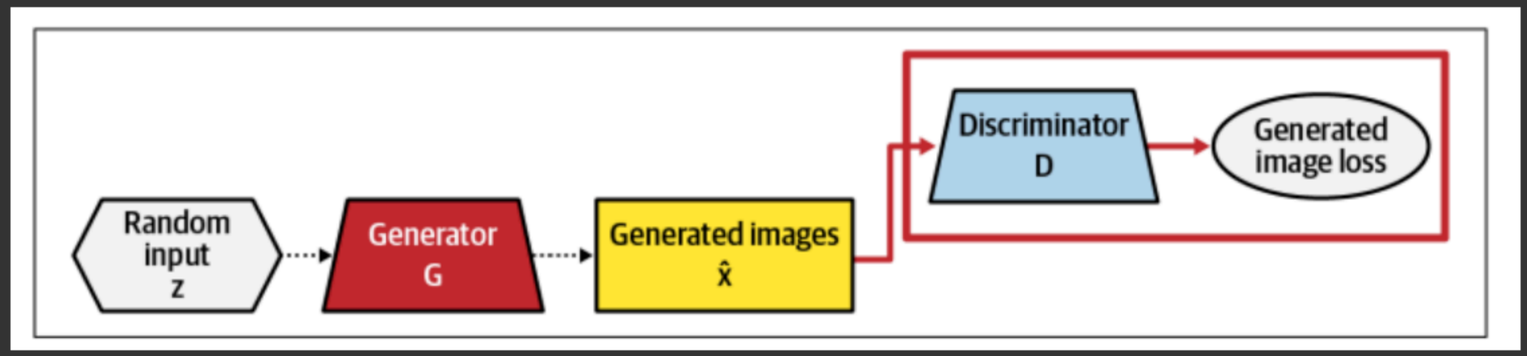
  - We calculate the loss by setting the target labels to 1 i.e. real. We do this because the generator's objective is to "fool" the discriminator.
  - We use the loss to perform gradient descent i.e. change the weights of the generator, so it gets better at generating real-like images to "fool" the discriminator.



- Here's how it works:
  - We generate a batch of images using the generator, pass the into the discriminator.



  - We calculate the loss by setting the target labels to 1 i.e. real. We do this because the generator's objective is to "fool" the discriminator.
  - We use the loss to perform gradient descent i.e. change the weights of the generator, so it gets better at generating real-like images to "fool" the discriminator.



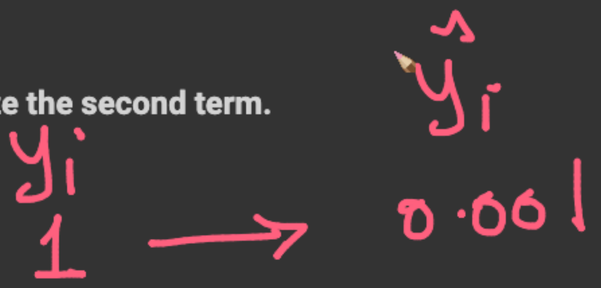
- $L(D(G(Z)), 0) = log(1 - D(G(Z)))$

- The above one is for just one sample. Over a batch, it will be,
  - $L(D(G(Z)), 0) = E_{Z \sim P(Z)}[log(1 - D(G(Z)))]$
- The generator will minimize the above loss function and to minimize the above the generator must maximize D(G(Z)). Now, it is very clear that **the discriminator wants to minimize D(G(Z)) and the generator wants to maximize D(G(Z)).**
- Understand that, the generator is never going to see any real data but for completeness, the generator loss function can be written as follows!

> $$min[E_{(X \sim P(X))}[logD(X)] + E_{(Z \sim P(Z))}[log(1 - D(G(Z)))]]$$

> **Note:** the generator has no control over the first term so the **generator will only minimize the second term.**

- **The following equation is minimized for training the generator:**
  - $\min_G \frac{1}{m} \sum_{i=1}^{m} log(1 - D(G(Z^i)))$.
- **Or, we can maximize the following for training the generator:**
  - $\max_G \frac{1}{m} \sum_{i=1}^{m} log(D(G(Z^i)))$.



Quiz-4

For training the GAN generator, which of the following is used :

1. $\max \frac{1}{m} \sum^{m} log(D(G(Z^i)))$

- $L(D(G(Z)), 0) = log(1 - D(G(Z)))$

- The above one is for just one sample. Over a batch, it will be,
  - $L(D(G(Z)), 0) = E_{Z \sim P(Z)}[log(1 - D(G(Z)))]$
- The generator will minimize the above loss function and to minimize the above the generator must maximize D(G(Z)). Now, it is very clear that **the discriminator wants to minimize D(G(Z)) and the generator wants to maximize D(G(Z)).**
- Understand that, the generator is never going to see any real data but for completeness, the generator loss function can be written as follows!

> $min[E_{(X \sim P(X))}[logD(X)] + E_{(Z \sim P(Z))}[log(1 - D(G(Z)))]]$

> **Note:** the generator has no control over the first term so the **generator will only minimize the second term.**

- **The following equation is minimized for training the generator:**
  - $\min_G \frac{1}{m} \sum_{i=1}^{m} log(1 - D(G(Z^i)))$.
- **Or, we can maximize the following for training the generator:**
  - $\max_G \frac{1}{m} \sum_{i=1}^{m} log(D(G(Z^i)))$.

$\mathcal{L}_G :$

Quiz-4

For training the GAN generator, which of the following is used :

1. $\max \frac{1}{m} \sum^{m} log(D(G(Z^i)))$

- $L(D(G(Z)), 0) = log(1 - D(G(Z)))$

- The above one is for just one sample. Over a batch, it will be,
  - $L(D(G(Z)), 0) = E_{Z \sim P(Z)}[log(1 - D(G(Z)))]$
- The generator will minimize the above loss function and to minimize the above the generator must maximize D(G(Z)). Now, it is very clear that **the discriminator wants to minimize D(G(Z)) and the generator wants to maximize D(G(Z)).**
- Understand that, the generator is never going to see any real data but for completeness, the generator loss function can be written as follows!

> $$min[E_{(X \sim P(X))}[logD(X)] + E_{(Z \sim P(Z))}[log(1 - D(G(Z)))]]$$

> **Note:** the generator has no control over the first term so the **generator will only minimize the second term.**

- **The following equation is minimized for training the generator:**
  - $\min_G \frac{1}{m} \sum_{i=1}^{m} log(1 - D(G(Z^i)))$.
- **Or, we can maximize the following for training the generator:**
  - $\max_G \frac{1}{m} \sum_{i=1}^{m} log(D(G(Z^i)))$.

Quiz-4

For training the GAN generator, which of the following is used :

1. $\max \frac{1}{m} \sum^{m} log(D(G(Z^i)))$

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

4.1 Global Optimality of $p_g = p_{\text{data}}$

We first consider

Proposition 1.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

-ve log-loss

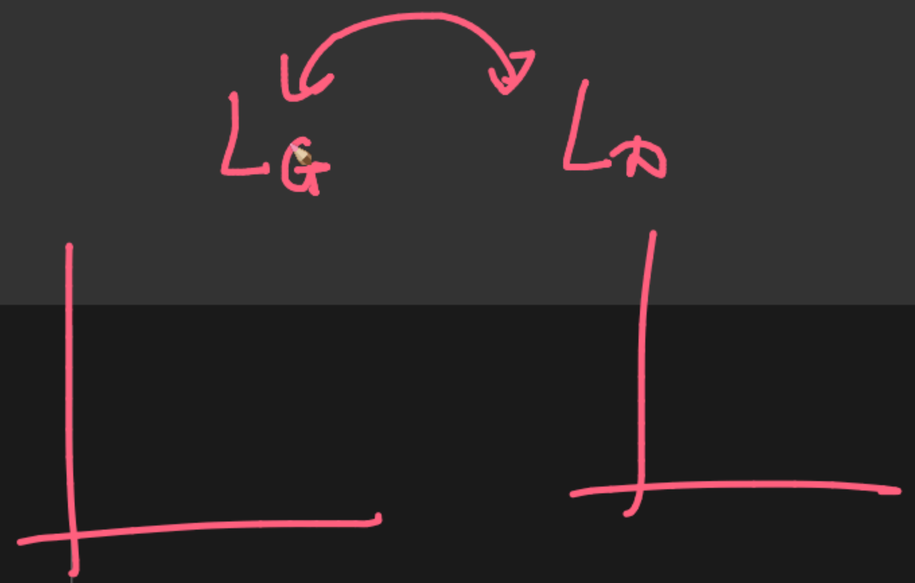
4.1 Global Optimality of $p_g = p_{\text{data}}$

We first consider

Proposition 1.

- Update the discriminator by ascending its stochastic gradient:
  - $\nabla_{\Theta_d} \frac{1}{m} \sum_{i=1}^{m} [log D(x^i) + log(1 - D(G(z^i)))]$.
- Sample minibatch of m noise samples $\{z^1, \ldots\ldots, z^m\}$ from noise prior $p_g(z)$.
- Update the generator by ascending its stochastic gradient:
  - $\nabla_{\Theta_g} \frac{1}{m} \sum_{i=1}^{m} log(D(G(Z^i)))$.

end for




```
[ ] class GAN(keras.Model):
        def __init__(self, discriminator, generator, latent_dim):
            super(GAN, self).__init__()
            self.discriminator = discriminator
            self.generator = generator
            self.latent_dim = latent_dim

        def compile(self, d_optimizer, g_optimizer, loss_fn):
            super(GAN, self).compile()
            self.d_optimizer = d_optimizer
            self.g_optimizer = g_optimizer
            self.loss_fn = loss_fn
            self.d_loss_metric = keras.metrics.Mean(name="d_loss")
            self.g_loss_metric = keras.metrics.Mean(name="g_loss")

        @property
        def metrics(self):
            return [self.d_loss_metric, self.g_loss_metric]
```

- $\nabla_{\Theta_g} \frac{1}{m} \sum_{i=1}^{m} log(D(G(Z^i)))$.

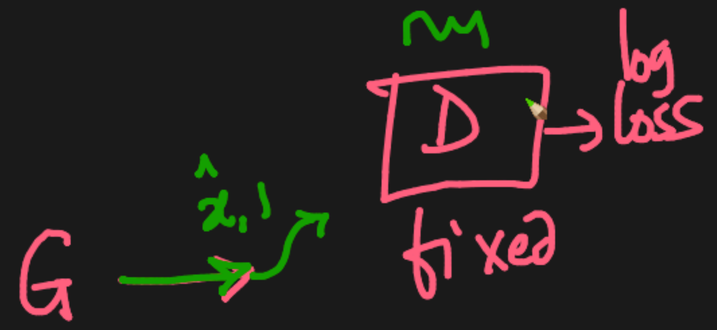
end for

```
[ ] class GAN(keras.Model):
        def __init__(self, discriminator, generator, latent_dim):
            super(GAN, self).__init__()
            self.discriminator = discriminator
            self.generator = generator
            self.latent_dim = latent_dim

        def compile(self, d_optimizer, g_optimizer, loss_fn):
            super(GAN, self).compile()
            self.d_optimizer = d_optimizer
            self.g_optimizer = g_optimizer
            self.loss_fn = loss_fn
            self.d_loss_metric = keras.metrics.Mean(name="d_loss")
            self.g_loss_metric = keras.metrics.Mean(name="g_loss")

        @property
        def metrics(self):
            return [self.d_loss_metric, self.g_loss_metric]

        def train_step(self, real_images):
            # Sample random points in the latent space
            batch_size = tf.shape(real_images)[0]
            random_latent_vectors = tf.random.normal(shape=(batch_size, 1, 1, self.latent_dim))

            # Decode them to fake images
```

```
[ ]             [tf.zeros((batch_size, 1)), tf.ones((batch_size, 1))], axis=0
        )

        # Add random noise to the labels - important trick!
        # labels += 0.05 * tf.random.uniform(tf.shape(labels))

        # Train the discriminator
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply_gradients(
            zip(grads, self.discriminator.trainable_weights)
        )

        # Sample random points in the latent space
        random_latent_vectors = tf.random.normal(shape=(batch_size, 1, 1, self.latent_dim))

        # # Assemble labels that say "all real images"
        # misleading_labels = tf.zeros((batch_size, 1))
        # Assemble labels that say "all real images"
        misleading_labels = tf.ones((batch_size, 1))

        # Train the generator (note that we should *not* update the weights
        # of the discriminator)!
        with tf.GradientTape() as tape:
            predictions = self.discriminator(self.generator(random_latent_vectors))
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weights))
```

```
[ ]             [tf.zeros((batch_size, 1)), tf.ones((batch_size, 1))], axis=0
            )

            # Add random noise to the labels - important trick!
            # labels += 0.05 * tf.random.uniform(tf.shape(labels))

            # Train the discriminator
            with tf.GradientTape() as tape:
                predictions = self.discriminator(combined_images)
                d_loss = self.loss_fn(labels, predictions)
            grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
            self.d_optimizer.apply_gradients(
                zip(grads, self.discriminator.trainable_weights)
            )

            # Sample random points in the latent space
            random_latent_vectors = tf.random.normal(shape=(batch_size, 1, 1, self.latent_dim))

            # # Assemble labels that say "all real images"
            # misleading_labels = tf.zeros((batch_size, 1))
            # Assemble labels that say "all real images"
            misleading_labels = tf.ones((batch_size, 1))

            # Train the generator (note that we should *not* update the weights
            # of the discriminator)!
            with tf.GradientTape() as tape:
                predictions = self.discriminator(self.generator(random_latent_vectors))
                g_loss = self.loss_fn(misleading_labels, predictions)
            grads = tape.gradient(g_loss, self.generator.trainable_weights)
            self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weights))
```

```
            zip(grads, self.discriminator.trainable_weights)
        )

        # Sample random points in the latent space
        random_latent_vectors = tf.random.normal(shape=(batch_size, 1, 1, self.latent_dim))

        # # Assemble labels that say "all real images"
        # misleading_labels = tf.zeros((batch_size, 1))
        # Assemble labels that say "all real images"
        misleading_labels = tf.ones((batch_size, 1))

        # Train the generator (note that we should *not* update the weights
        # of the discriminator)!
        with tf.GradientTape() as tape:
            predictions = self.discriminator(self.generator(random_latent_vectors))
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weights))

        # Update metrics
        self.d_loss_metric.update_state(d_loss)
        self.g_loss_metric.update_state(g_loss)
        return {
            "d_loss": self.d_loss_metric.result(),
            "g_loss": self.g_loss_metric.result(),
        }
```



The function save_img_batch will save the batch of images generated by GANS.

> (c) To achieve more stability in training the GAN

Ans: (c) To achieve more stability in training the GAN . Refer to section DETAILS OF ADVERSARIAL TRAINING in the paper

```
[ ] epochs = 60  #  try ~100 epochs
    latent_dim = 128
    # training_log_dir = "./drive/MyDrive/Datasets - DSML/IntroToGANs/logs/training_logs"

    # Sets up a timestamped log directory.
    # epoch_end_logdir = "./drive/MyDrive/Datasets - DSML/IntroToGANs/logs/epoch_end_logs"
    # Creates a file writer for the log directory.
    file_writer = tf.summary.create_file_writer(epoch_end_logdir)

    gan = GAN(discriminator=discriminator, generator=generator, latent_dim=latent_dim)
    gan.compile(
      d_optimizer=keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5),
      g_optimizer=keras.optimizers.Adam(learning_rate=0.00015, beta_1=0.5),
      loss_fn=keras.losses.BinaryCrossentropy(),
    )

    tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=training_log_dir, histogram_freq=1)
```

- Start training the GANS and observe the generated output after every epoch

```
[ ] %%time
    history = gan.fit(
```

Chrome File Edit View History Bookmarks Profiles Tab Window Help

L11:Introductory_Lecture_on_ | plot log(x) - Google Search | 1406.2661.pdf | 1812.04948.pdf | 1703.10593.pdf | CycleGAN | TensorFlow Core | CS 230 - Convolutional Neural

colab.research.google.com/drive/1g_XWY5tcd6M9JtWSHwFkkeOnl4W8-tiH#scrollTo=fHKNFZdwVqB3

+ Code + Text Last saved at 10:32 Connect

TensorBoard SCALARS GRAPHS DISTRIBUTIONS HISTOGRAMS TIME SERIES INACTIVE

Show data download links

Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis

STEP RELATIVE WALL

Runs

Write a regex to filter runs

TOGGLE ALL RUNS

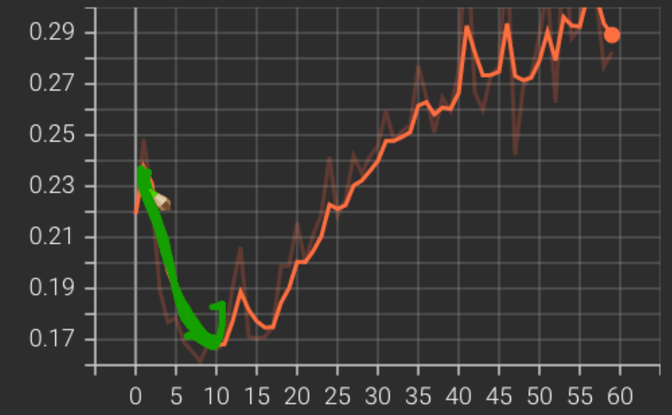
./logs/training_logs/train

Filter tags (regular expressions supported)

epoch_d_loss

D model

epoch_d_loss
tag: epoch_d_loss



G: is very basic

epoch_g_loss

Name	Smoothed	Value	Step	Time	Relative
.	4.768	4.899	7	Sat Sep 10, 18:16:44	14m 5s

epoch_g_loss
tag: epoch_g_loss

38 / 38

Chrome File Edit View History Bookmarks Profiles Tab Window Help

L11:Introductory_Lecture_on_ | plot log(x) - Google Search | 1406.2661.pdf | 1812.04948.pdf | 1703.10593.pdf | CycleGAN | TensorFlow Core | CS 230 - Convolutional Neural

colab.research.google.com/drive/1g_XWY5tcd6M9JtWSHwFkkeOnl4W8-tiH#scrollTo=fHKNFZdwVqB3

+ Code + Text Last saved at 10:32 Connect

[ ]

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis

STEP RELATIVE WALL

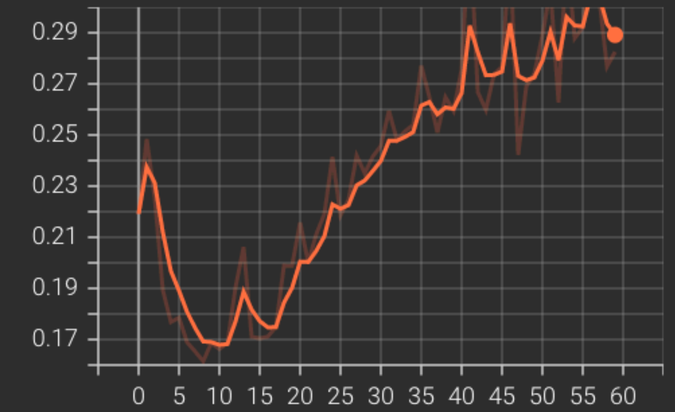
Runs

Write a regex to filter runs

TOGGLE ALL RUNS

./logs/training_logs/train

epoch_d_loss
tag: epoch_d_loss



epoch_g_loss

epoch_g_loss
tag: epoch_g_loss

39 / 39

```
[ ]
    # Saving as gif file
    for filename in filenames:
        images.append(imageio.imread(filename))
    imageio.mimsave('/content/movie.gif', images)
```

```
[ ] from IPython.display import Image
    # Display gif
    Image(open('/content/movie.gif','rb').read())
```

We can visualize the training process by combining the sample images generated after each epoch into a video using OpenCV.

Chrome File Edit View History Bookmarks Profiles Tab Window Help

L11:Introductory_Lecture_on_ | plot log(x) - Google Search | 1406.2661.pdf | 1812.04948.pdf | 1703.10593.pdf | CycleGAN | TensorFlow Core | CS 230 - Convolutional Neural

colab.research.google.com/drive/1g_XWY5tcd6M9JtWSHwFkkeOnl4W8-tiH#scrollTo=fHKNFZdwVqB3

+ Code + Text Last saved at 10:32 Connect

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis

STEP RELATIVE WALL

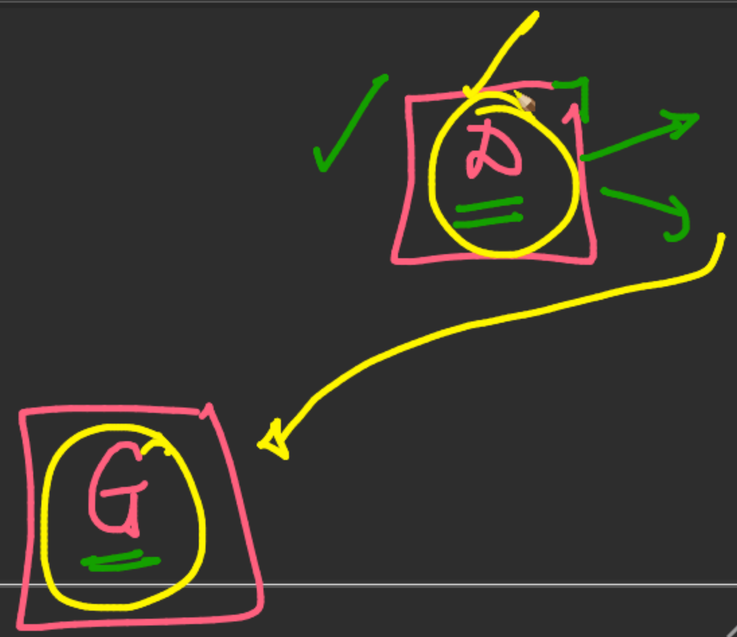
Runs

Write a regex to filter runs

TOGGLE ALL RUNS

./logs/training_logs/train

epoch_d_loss
tag: epoch_d_loss



epoch_g_loss

epoch_g_loss
tag: epoch_g_loss

41 / 41

GANs however, can be notoriously difficult to train, and are extremely sensitive to hyperparameters, activation functions and regularization.

What are the difficulties with training GANs?

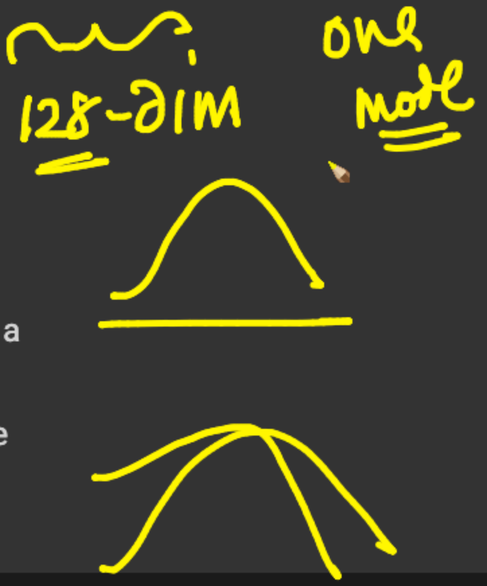
1. Vanishing Gradient

- If the Discriminator is too good , then generator training can fail due to vanishing gradients.
- An optimal discriminator doesn't provide enough information for the generator to progress.
- When we apply backpropagation, we use the chain rule of differentiation, which has a multiplying effect.
- Thus, gradient flows backward, from the final layer to the first layer. As it flows backward, it gets increasingly smaller.
- Sometimes, the gradient is so small that the initial layers learn very slowly or stop learning completely.
- Hence the generator fails to generate real quality images.

2. Mode Collapse:

- Most of the times the data we have is multimodal, which means it has multiple modes where each mode represents data which share similar features.
- A very common problem while training GANs is Mode Collapse in which the Generator produces images belonging to only a few modes of the data and ignores all other modes, hence the generated images are very similar and lack variety.
- Experimenting with the Learning Rate of both Generator and Discriminator could help in overcoming this problem. So while training your GANs you should monitor the generated samples to see if your model is suffering from this problem.

- An optimal discriminator doesn't provide enough information for the generator to progress.
- When we apply backpropagation, we use the chain rule of differentiation, which has a multiplying effect.
- Thus, gradient flows backward, from the final layer to the first layer. As it flows backward, it gets increasingly smaller.
- Sometimes, the gradient is so small that the initial layers learn very slowly or stop learning completely.
- Hence the generator fails to generate real quality images.



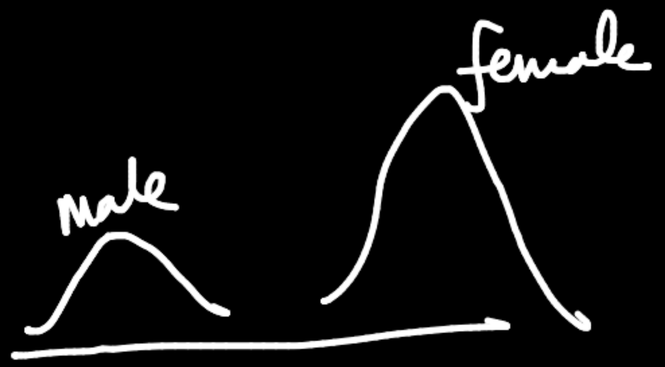
2. Mode Collapse:

- Most of the times the data we have is multimodal, which means it has multiple modes where each mode represents data which share similar features.
- A very common problem while training GANs is Mode Collapse in which the Generator produces images belonging to only a few modes of the data and ignores all other modes, hence the generated images are very similar and lack variety.
- Experimenting with the Learning Rate of both Generator and Discriminator could help in overcoming this problem. So while training your GANs you should monitor the generated samples to see if your model is suffering from this problem.

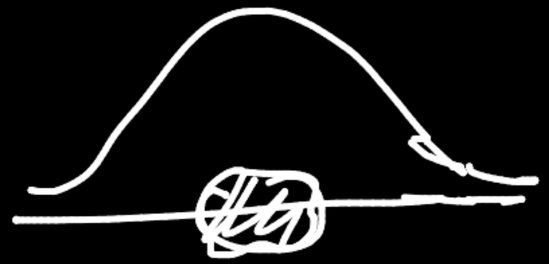
```
[ ]  # Hyperparameters
     batch_size = 512
     epochs = 60
     latent_dim = 100
     image_dimensions = (64, 64)

     # dataloader
     dataset_path = "./animefacedataset"
     dataset = keras.preprocessing.image_dataset_from_directory(
         dataset_path, label_mode=None, image_size=image_dimensions, batch_size=batch_size
     )
```

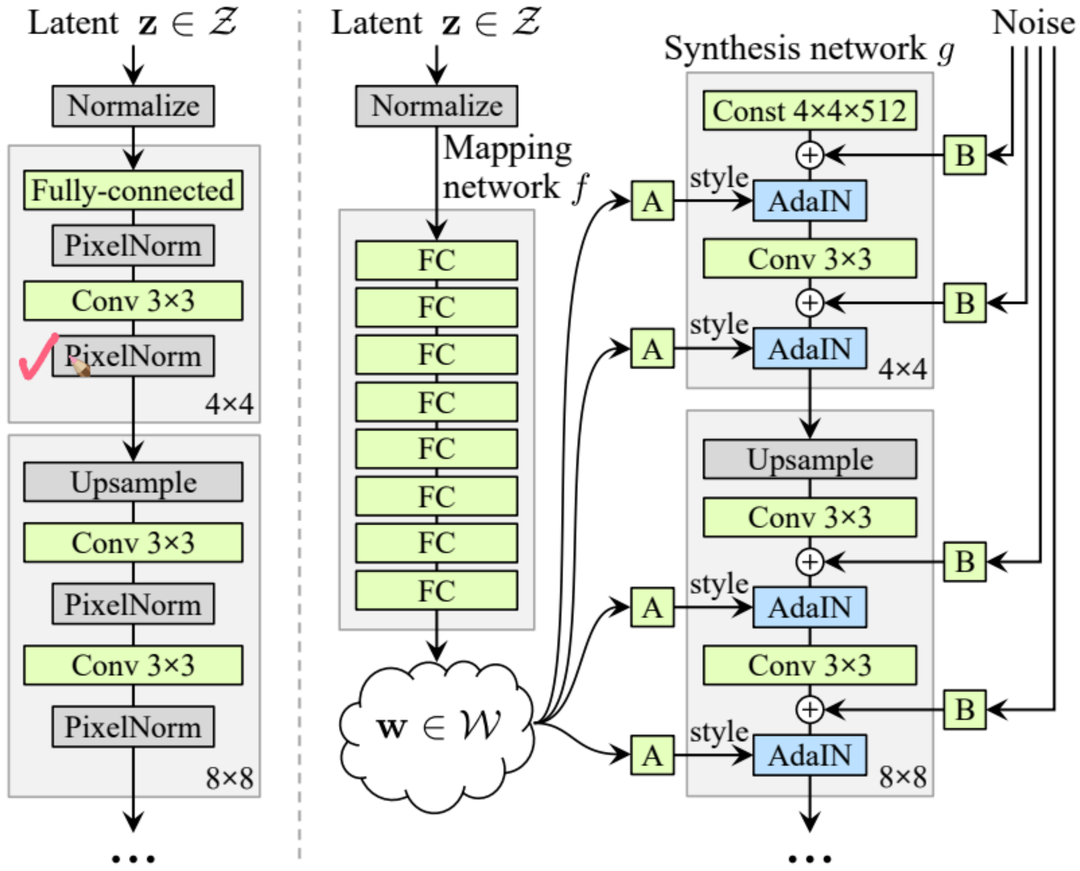
data's dist

anime → male
anime → Female



Female-anime





(a) Traditional (b) Style-based generator

Figure 1. While a traditional generator [30] feeds the latent code though the input layer only, we first map the input to an intermediate latent space $\mathcal{W}$, which then controls the generator through adaptive instance normaliz

	Method	CelebA-HQ	FFHQ
A	Baseline Progressive GAN [30]	7.79	8.04
B	+ Tuning (incl. bilinear up/down)	6.11	5.25
C	+ Add mapping and styles	5.34	4.85
D	+ Remove traditional input	5.07	4.88
E	+ Add noise inputs	**5.06**	4.42
F	+ Mixing regularization	5.17	**4.40**

Table 1. Fréchet inception distance (FID) for various generator designs (lower is better). In this paper we calculate the FIDs using 50,000 images drawn randomly from the training set, and report the lowest distance encountered over the course of training.

Finally, we provide our generator with a direct means to generate stochastic detail by introducing explicit *noise inputs*. These are single-channel images consisting of uncorrelated Gaussian noise, and we feed a dedicated noise image to each layer of the synthesis network. The noise image is broadcasted to all feature maps using learned per-feature scaling factors and then added to the output of the corresponding convolution, as illustrated in Figure 1b. The implications of adding the noise inputs are discussed in Sec-

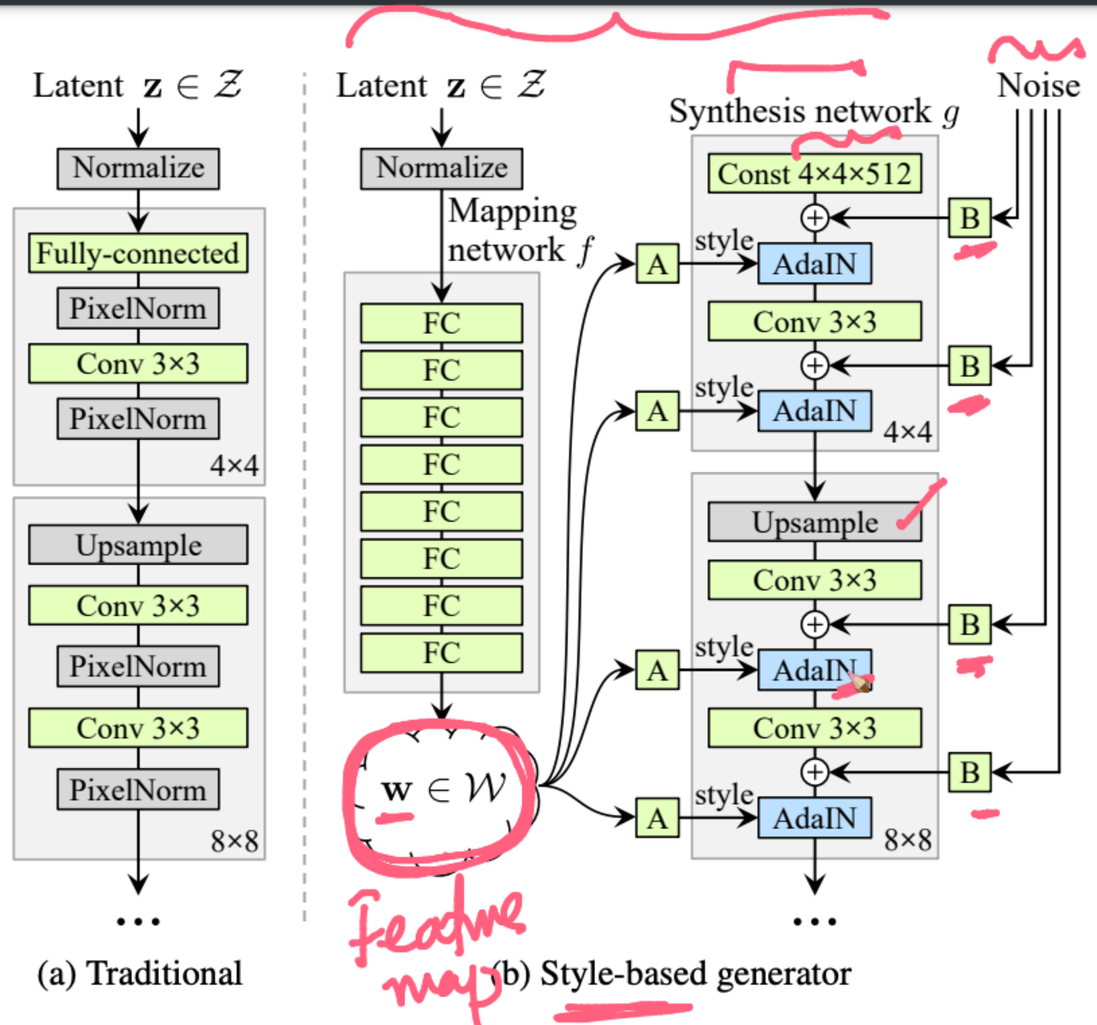


Figure 1. While a traditional generator [30] feeds the latent code though the input layer only, we first map the input to an intermediate latent space $\mathcal{W}$, which then controls the generator through adaptive instance normaliz

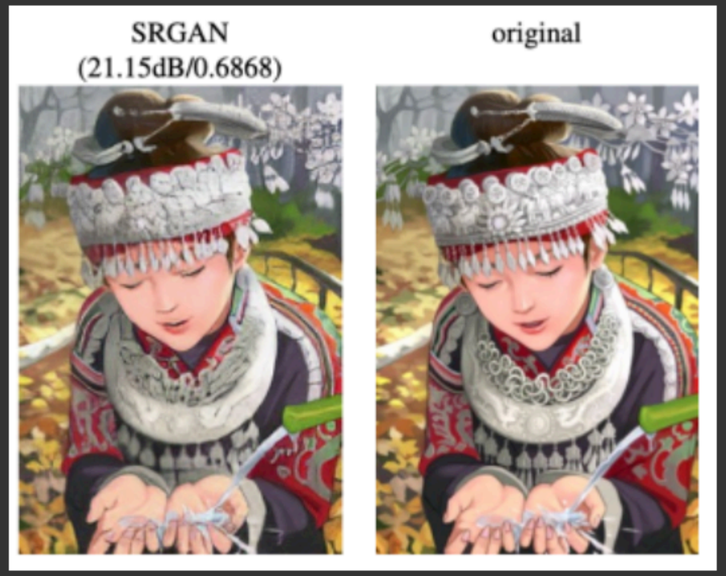
	Method	CelebA-HQ	FFHQ
A	Baseline Progressive GAN [30]	7.79	8.04
B	+ Tuning (incl. bilinear up/down)	6.11	5.25
C	+ Add mapping and styles	5.34	4.85
D	+ Remove traditional input	5.07	4.88
E	+ Add noise inputs	**5.06**	4.42
F	+ Mixing regularization	5.17	**4.40**

Table 1. Fréchet inception distance (FID) for various generator designs (lower is better). In this paper we calculate the FIDs using 50,000 images drawn randomly from the training set, and report the lowest distance encountered over the course of training.

Finally, we provide our generator with a direct means to generate stochastic detail by introducing explicit *noise inputs*. These are single-channel images consisting of uncorrelated Gaussian noise, and we feed a dedicated noise image to each layer of the synthesis network. The noise image is broadcasted to all feature maps using learned per-feature scaling factors and then added to the output of the corresponding convolution, as illustrated in Figure 1b. The implications of adding the noise inputs are discussed in Sec-

SRGAN

- Super-resolution GAN applies a deep network in combination with an adversary network to produce higher resolution images.
- SRGAN is more appealing to a human with more details compared with the similar design without GAN.
- During the training, A high-resolution image (HR) is downsampled to a low-resolution image (LR).
- A GAN generator upsamples LR images to super-resolution images (SR). Click to know more



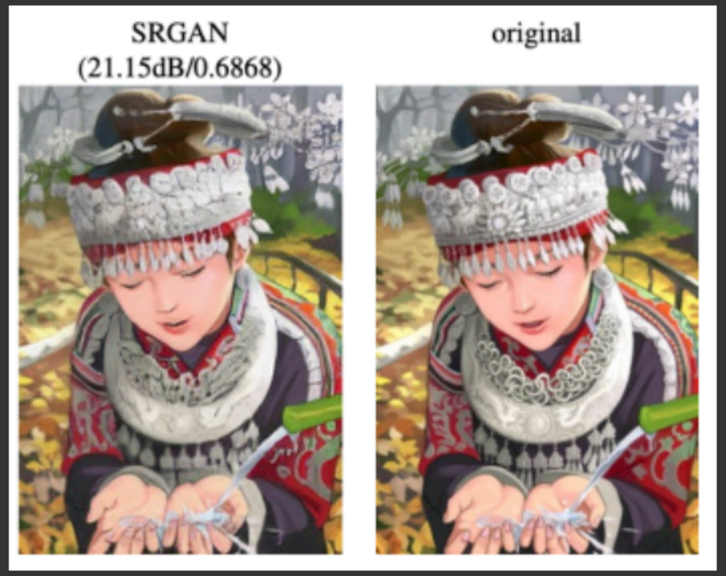
10 MP → 40 MP

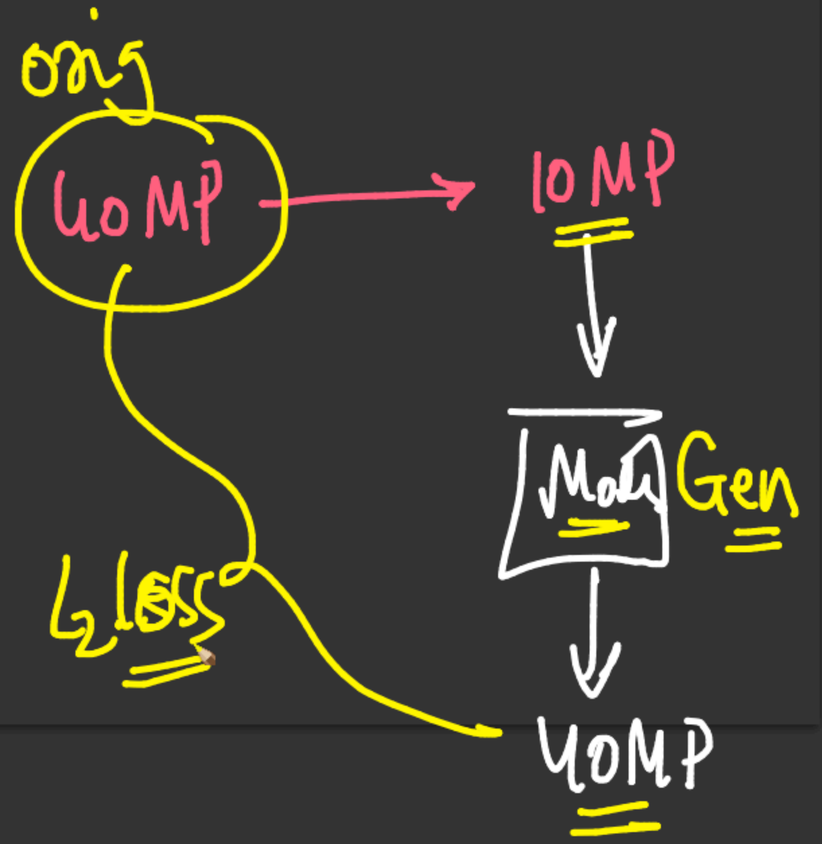
Cycle GAN

- CycleGAN is used to enable training without the need for paired data.
- Cycle GAN is used to transfer characteristic of one image to another or can map the distribution of images to another.

SRGAN

- Super-resolution GAN applies a deep network in combination with an adversary network to produce higher resolution images.
- SRGAN is more appealing to a human with more details compared with the similar design without GAN.
- During the training, A high-resolution image (HR) is downsampled to a low-resolution image (LR).
- A GAN generator upsamples LR images to super-resolution images (SR). Click to know more





Cycle GAN

- CycleGAN is used to enable training without the need for paired data.
- Cycle GAN is used to transfer characteristic of one image to another or can map the distribution of images to another.

$F(y) = x$; $G(x) = y$

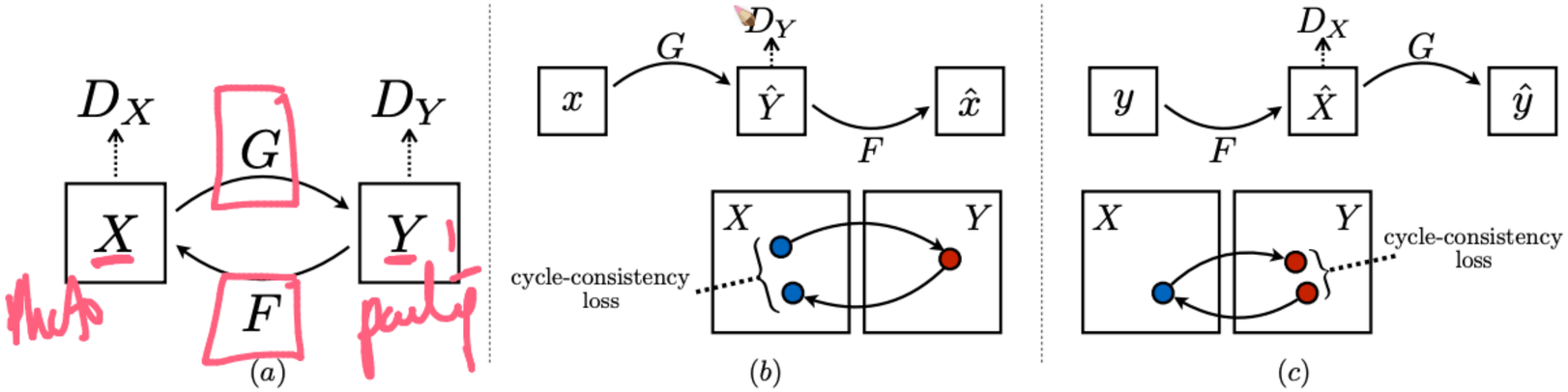


Figure 3: (a) Our model contains two mapping functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and associated adversarial discriminators D_Y and D_X. D_Y encourages G to translate X into outputs indistinguishable from domain Y, and vice versa for D_X and F. To further regularize the mappings, we introduce two *cycle consistency losses* that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$

images cannot be distinguished from images in the target domain.

Image-to-Image Translation The idea of image-to-image translation goes back at least to Hertzmann et al.'s Image Analogies [19], who employ a non-parametric texture model [10] on a single input-output training image pair. More recent approaches use a *dataset* of input-output exam-
ples to learn a parametric
(e.g., [33]). Our approach builds on the "pix2pix" frame-

tween the input and output, nor do we assume that the input and output have to lie in the same low-dimensional embedding space. This makes our method a general-purpose solution for many vision and graphics tasks. We directly compare against several prior and contemporary approaches in Section 5.1.

Cycle Consistency The idea of using transitivity as a
a long history. In
visual tracking, enforcing simple forward-backward con-

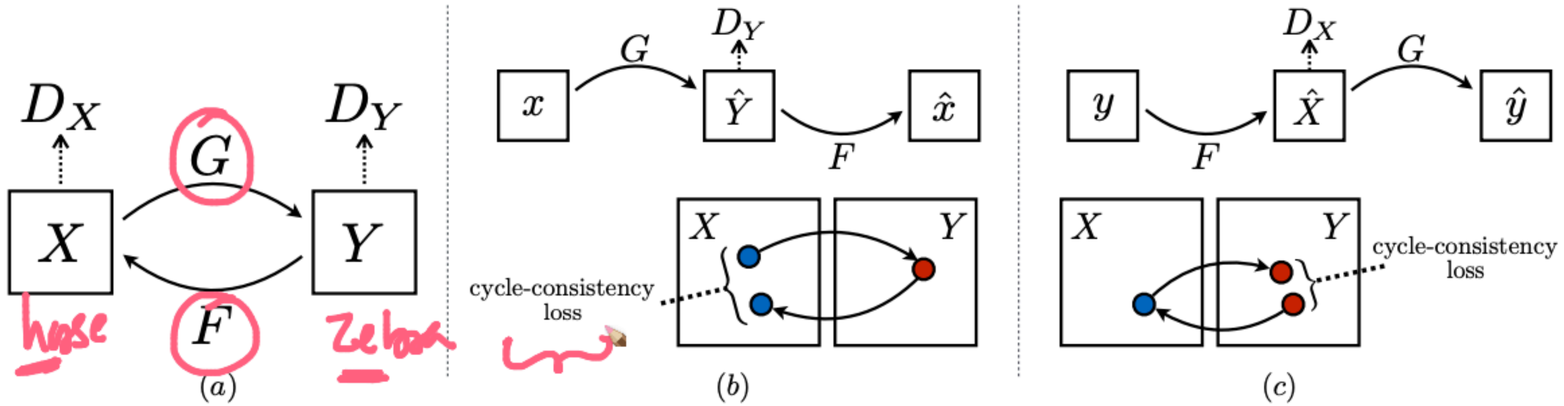


Figure 3: (a) Our model contains two mapping functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and associated adversarial discriminators D_Y and D_X. D_Y encourages G to translate X into outputs indistinguishable from domain Y, and vice versa for D_X and F. To further regularize the mappings, we introduce two *cycle consistency losses* that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$

images cannot be distinguished from images in the target domain.

Image-to-Image Translation The idea of image-to-image translation goes back at least to Hertzmann et al.'s Image Analogies [19], who employ a non-parametric texture model [10] on a single input-output training image pair. More recent approaches use a *dataset* of input-output examples to learn a parametric translation function using CNNs (e.g., [33]). Our approach

work of Isola et al. [22],

tween the input and output, nor do we assume that the input and output have to lie in the same low-dimensional embedding space. This makes our method a general-purpose solution for many vision and graphics tasks. We directly compare against several prior and contemporary approaches in Section 5.1.

Cycle Consistency The idea of using transitivity as a way to regularize structured data has a long history. In

ard-backward consistency has been a standard trick for decades [24, 48],

50 / 50

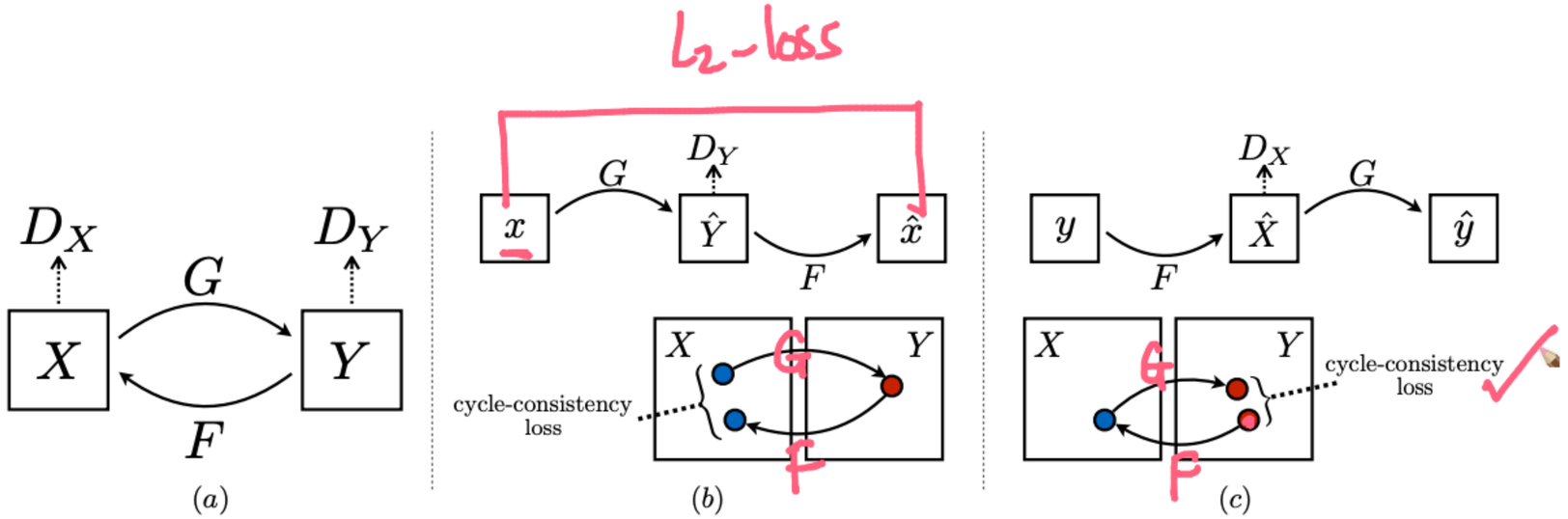


Figure 3: (a) Our model contains two mapping functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and associated adversarial discriminators D_Y and D_X. D_Y encourages G to translate X into outputs indistinguishable from domain Y, and vice versa for D_X and F. To further regularize the mappings, we introduce two *cycle consistency losses* that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$

images cannot be distinguished from images in the target domain.

Image-to-Image Translation The idea of image-to-image translation goes back at least to Hertzmann et al.'s Image Analogies [19], who employ a non-parametric texture model [10] on a single input-output training image pair. More recent approaches use a *dataset* of input-output examples to learn a parametric translation function using CNNs (e.g., [33]). Our approach

work of Isola et al. [22],

tween the input and output, nor do we assume that the input and output have to lie in the same low-dimensional embedding space. This makes our method a general-purpose solution for many vision and graphics tasks. We directly compare against several prior and contemporary approaches in Section 5.1.

Cycle Consistency The idea of using transitivity as a way to regularize structured data has a long history. In

ard-backward con-

sistency has been a standard trick for decades [24, 48].

51 / 51

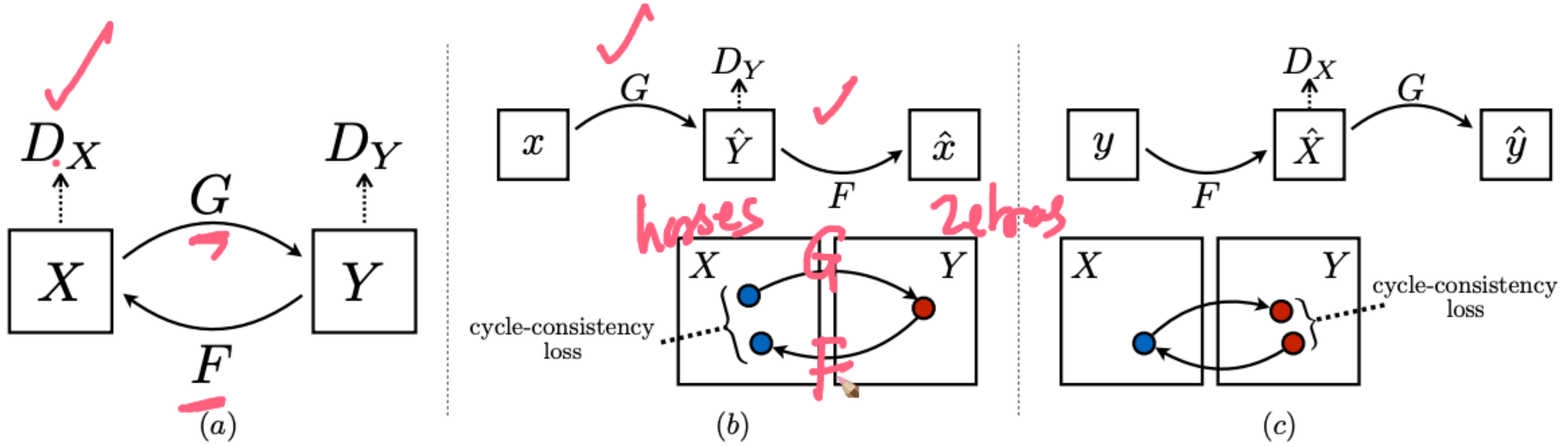


Figure 3: (a) Our model contains two mapping functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and associated adversarial discriminators D_Y and D_X. D_Y encourages G to translate X into outputs indistinguishable from domain Y, and vice versa for D_X and F. To further regularize the mappings, we introduce two *cycle consistency losses* that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$

images cannot be distinguished from images in the target domain.

Image-to-Image Translation The idea of image-to-image translation goes back at least to Hertzmann et al.'s Image Analogies [19], who employ a non-parametric texture model [10] on a single input-output training image pair. More recent approaches use a *dataset* of input-output examples to learn a parametric translation function using CNNs (e.g., [33]). Our approach

work of Isola et al. [22],

tween the input and output, nor do we assume that the input and output have to lie in the same low-dimensional embedding space. This makes our method a general-purpose solution for many vision and graphics tasks. We directly compare against several prior and contemporary approaches in Section 5.1.

Cycle Consistency The idea of using transitivity as a way to regularize structured data has a long history. In

ard-backward con-

sistency has been a standard trick for decades [24, 48].

$$\min \sum_{x_i, y_j} \| F(G(x_i)) - x_i \|_2 + \| G(F(y_j)) - y_j \|_2$$

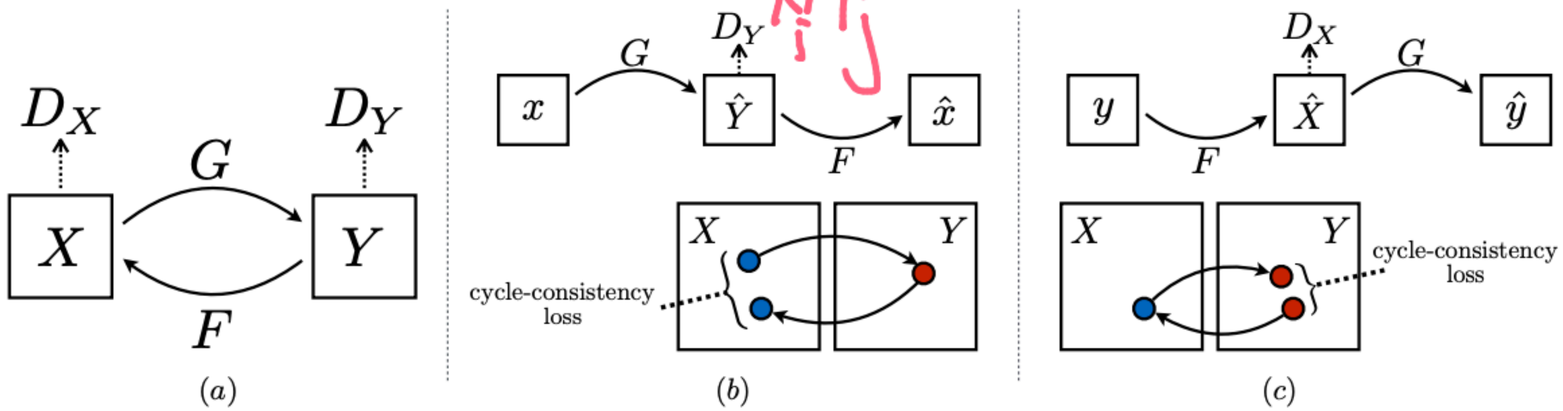


Figure 3: (a) Our model contains two mapping functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and associated adversarial discriminators D_Y and D_X. D_Y encourages G to translate X into outputs indistinguishable from domain Y, and vice versa for D_X and F. To further regularize the mappings, we introduce two *cycle consistency losses* that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$

images cannot be distinguished from images in the target domain.

Image-to-Image Translation The idea of image-to-image translation goes back at least to Hertzmann et al.'s Image Analogies [19], who employ a non-parametric texture model [10] on a single input-output training image pair. More recent approaches use a *dataset* of input-output examples to learn a parametric t
(e.g., [33]). Our approach

tween the input and output, nor do we assume that the input and output have to lie in the same low-dimensional embedding space. This makes our method a general-purpose solution for many vision and graphics tasks. We directly compare against several prior and contemporary approaches in Section 5.1.

Cycle Consistency The idea of using transitivity as a
a long history. In
ard-backward con-

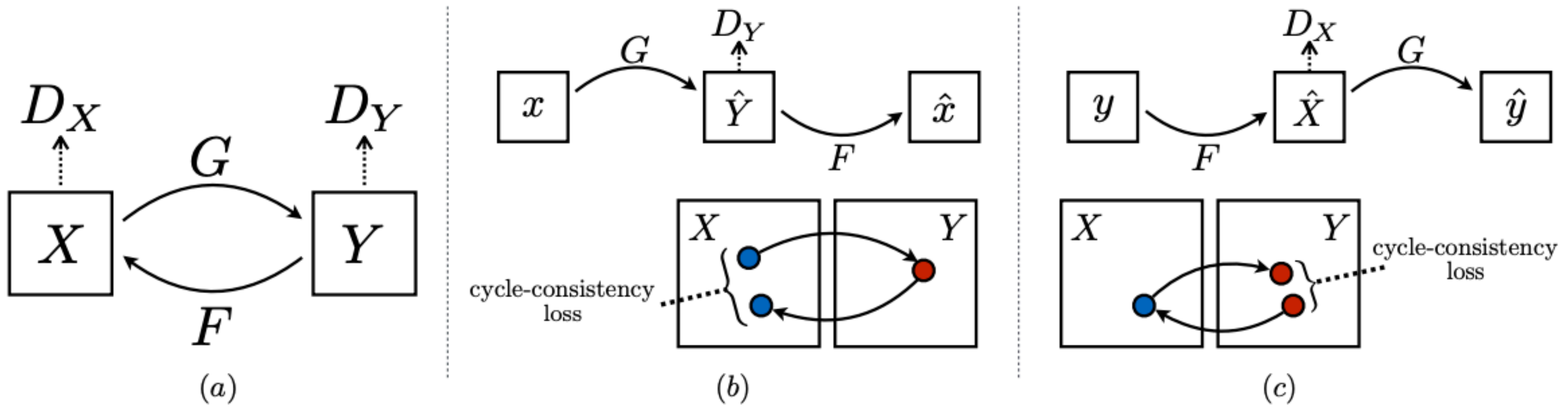


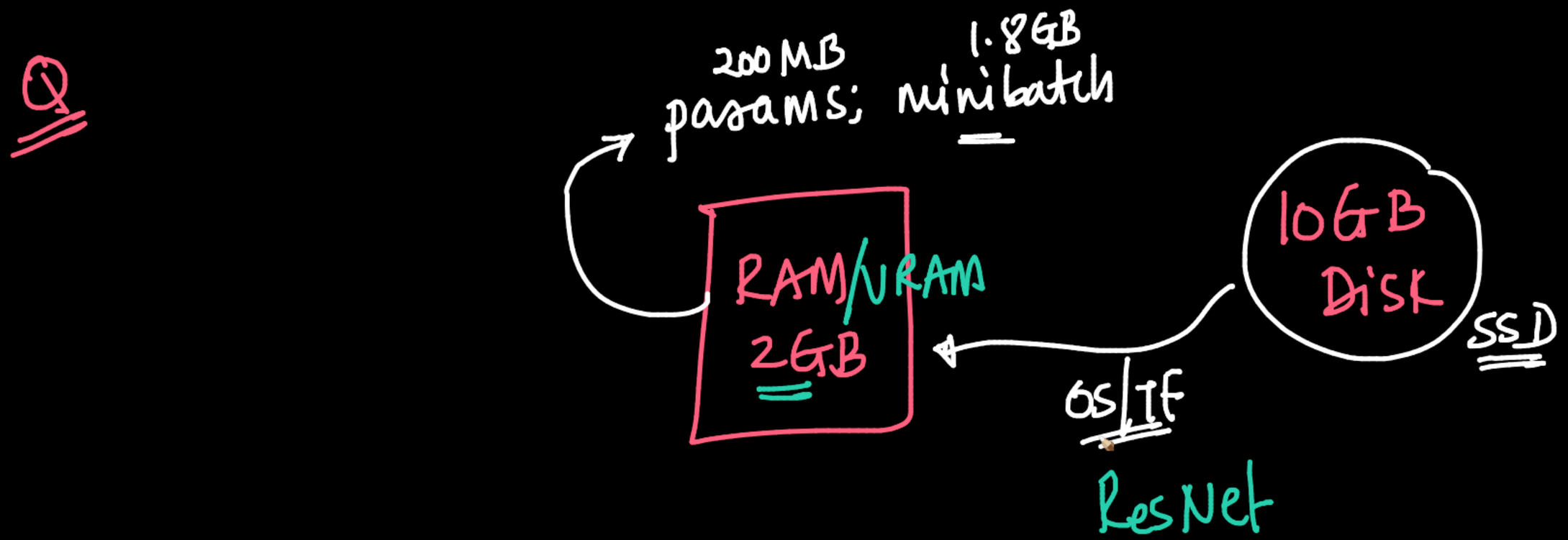
Figure 3: (a) Our model contains two mapping functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and associated adversarial discriminators D_Y and D_X. D_Y encourages G to translate X into outputs indistinguishable from domain Y, and vice versa for D_X and F. To further regularize the mappings, we introduce two *cycle consistency losses* that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$

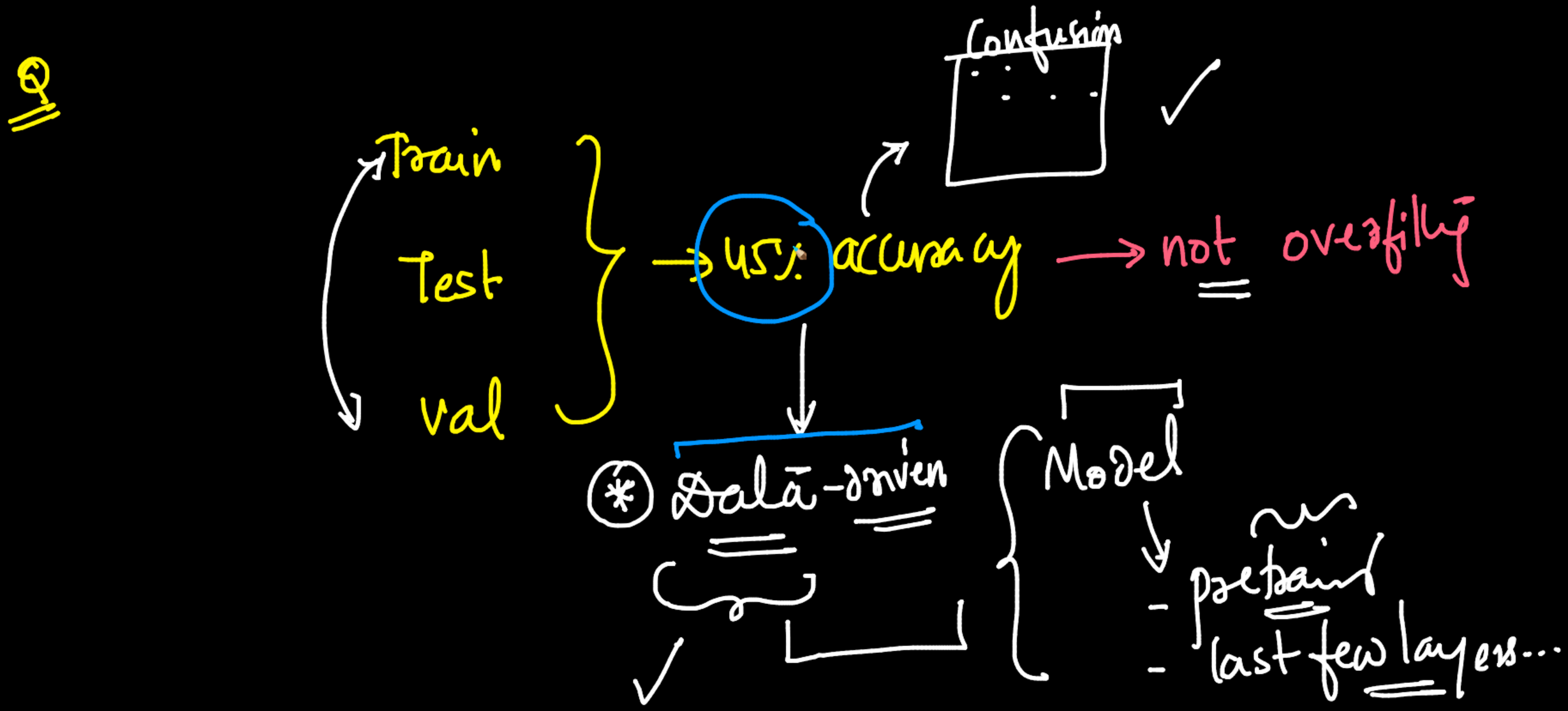
images cannot be distinguished from images in the target domain.

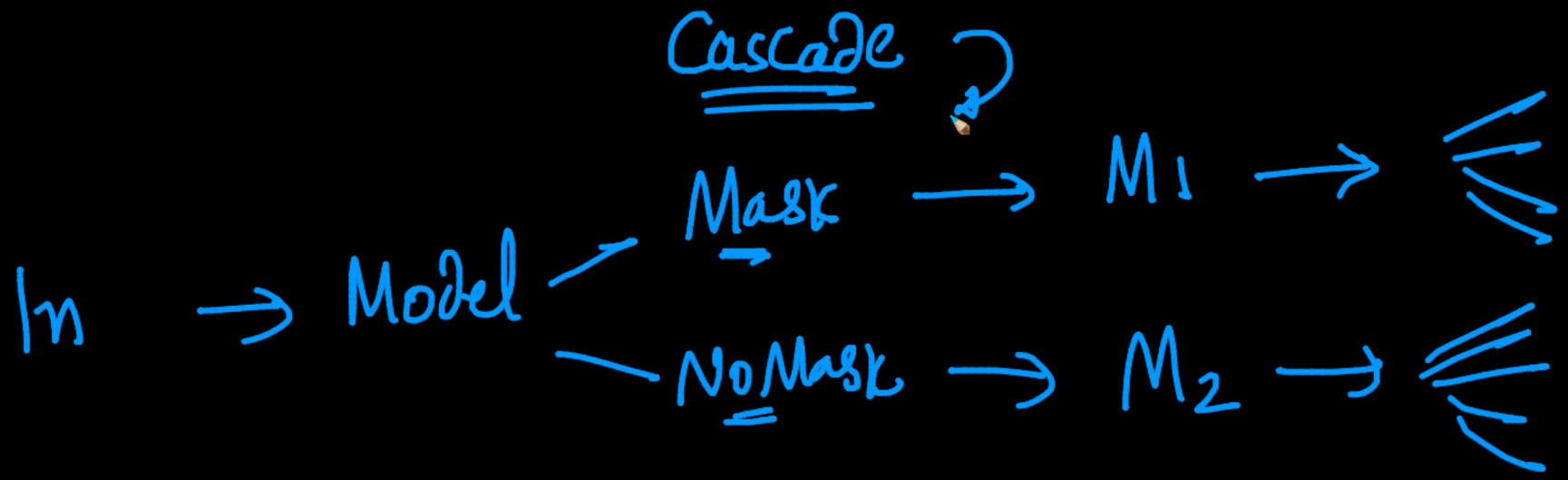
Image-to-Image Translation The idea of image-to-image translation goes back at least to Hertzmann et al.'s Image Analogies [19], who employ a non-parametric texture model [10] on a single input-output training image pair. More recent approaches use a *dataset* of input-output examples to learn a parametric t
(e.g., [33]). Our approach

tween the input and output, nor do we assume that the input and output have to lie in the same low-dimensional embedding space. This makes our method a general-purpose solution for many vision and graphics tasks. We directly compare against several prior and contemporary approaches in Section 5.1.

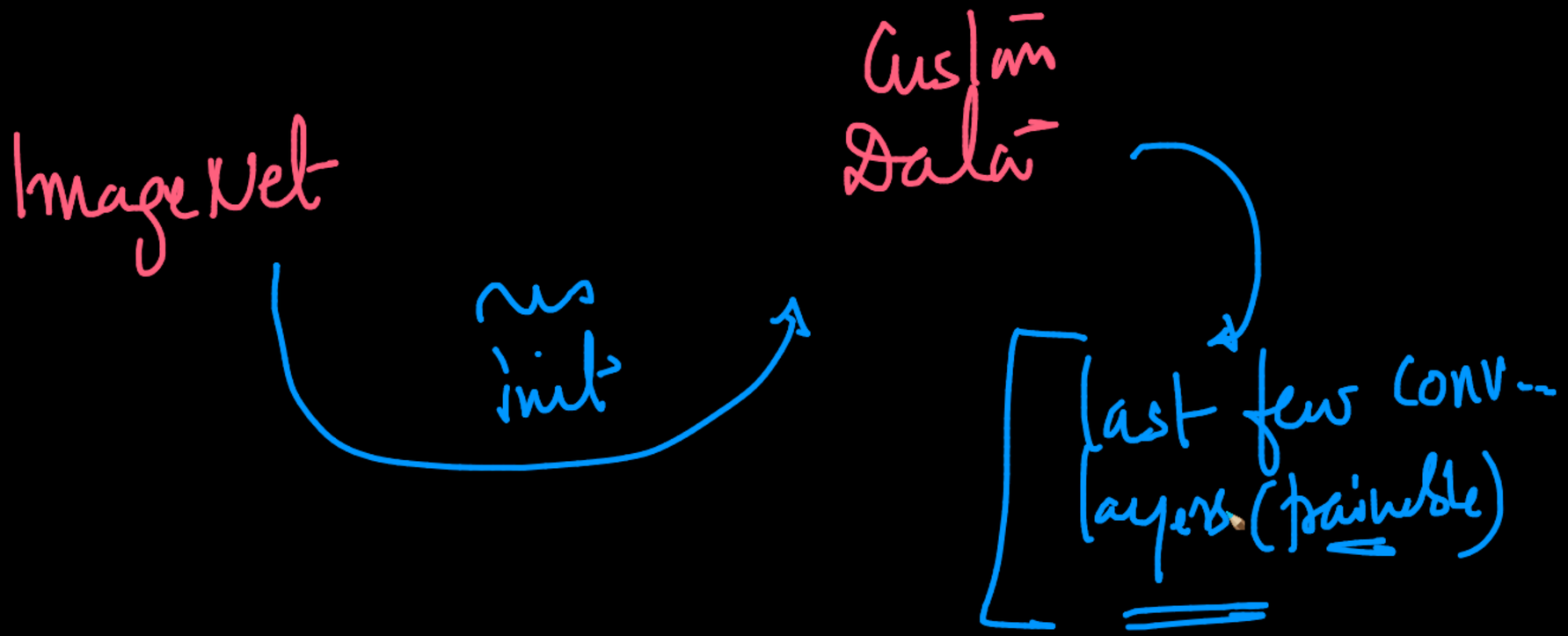
Cycle Consistency The idea of using transitivity as a
a long history. In
ard-backward con-


Q
200 MB
params;
1.8GB
minibatch
RAM/VRAM
2GB
10GB
Disk
SSD
6S/IF
ResNet

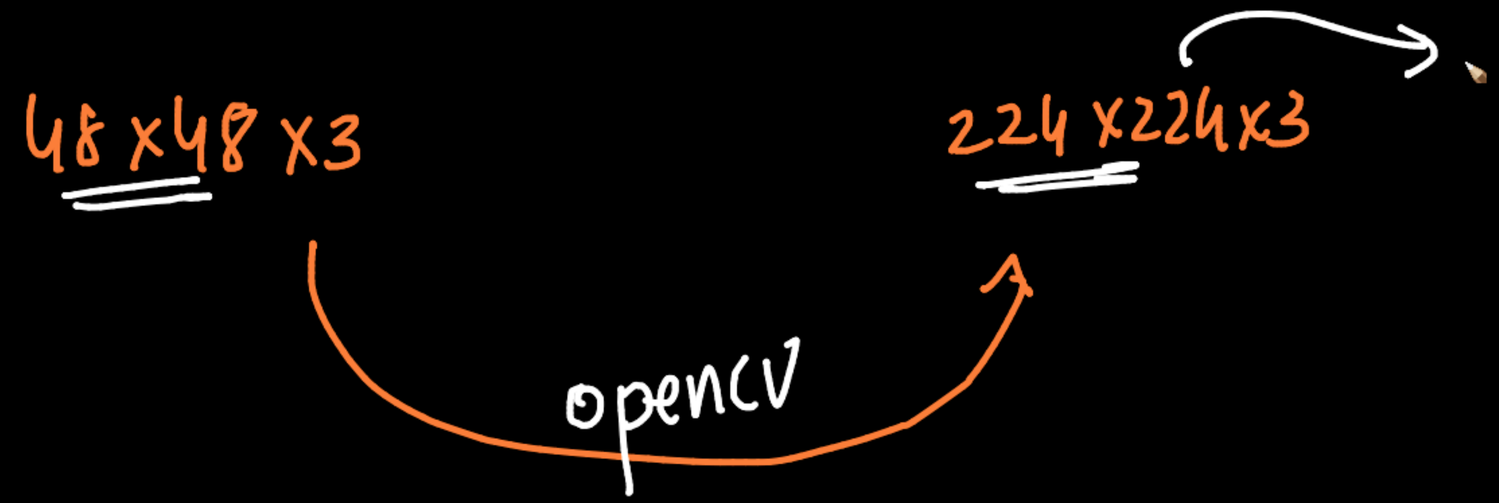



Cascade ?
In
Model
Mask
M1
NoMask
M2

Q



Q



Agenda:

1. Generate new anime characters
2. Understand the architecture of Generative Adverserial Network(GAN)
3. Applications of GAN

Problem Statement:

`You are working as a Data Scientist at Kyoto Animation, a Japanese animation studio`

- In 2020, about 179 new animes were released and total more than 4505 anime have been released.
- Anime industry wants to develop an automated system to generate newer anime characters.