# APS360: Applied Fundamentals of Deep Learning

Week 9: Generative Neural Networks

# Generative Models

# Generative Model vs. Discriminative Model

Suppose we have two tasks on a dataset of tweets:

1. Identify if a tweet is real or fake

# Generative Model vs. Discriminative Model

Suppose we have two tasks on a dataset of tweets:

1. Identify if a tweet is real or fake
    - This task is **supervised** and requires a **discriminative model** .
    - The model learns to approximate **p(y|x)**

# Generative Model vs. Discriminative Model

Suppose we have two tasks on a dataset of tweets:

1. Identify if a tweet is real or fake
    - This task is **supervised** and requires a **discriminative model** .
    - The model learns to approximate **p(y|x)**
2. Generate a new tweet

# Generative Model vs. Discriminative Model

Suppose we have two tasks on a dataset of tweets:

1. Identify if a tweet is real or fake
    - This task is **supervised** and requires a **discriminative model** .
    - The model learns to approximate **p(y|x)**
2. Generate a new tweet
    - This task is **unsupervised** and requires a **generative model** .
    - The model learns to approximate **p(x)**

# Generative Model vs. Discriminative Model

**Input**

**Discriminative** Model
(e.g. Classifier)

**Output**

Label

Encoding

**Generative** Model
(e.g. Variational Autoencoder)

# Generative Learning

Generative learning is an **Unsupervised Learning task** :

- There is a loss function → an auxiliary task that we know the answer to
- There is no ground truth with respect to the actual task that we want to accomplish.
- We are learning the **structure & distribution of data** , rather than labels for data!

# Generative Models

A generative model is used to generate new data, using some input encoding:

## Unconditional Generative Models

- Random noise or a fixed token as input
- No control over what category they generate

## Conditional Generative Models

- One-hot encoding of the target category + random noise, or
- An embedding generated by another model (e.g., from CNN)
- User have a high-level control over what the model will generate

# Generative Models

There are different families of deep generative models:

- Autoregressive Models
- Variational AutoEncoders (VAEs)
- Generative Adversarial Networks (GANs)
- Flow-Based Generative Models
- Diffusion Models

We already covered these

We are covering it today

We won't cover them in this course

# Problem with Autoencoders

Vanilla autoencoders generate blurry images with blurry backgrounds

To minimize the MSE loss, autoencoders predict the average pixel

Can we use a **better loss function** ?

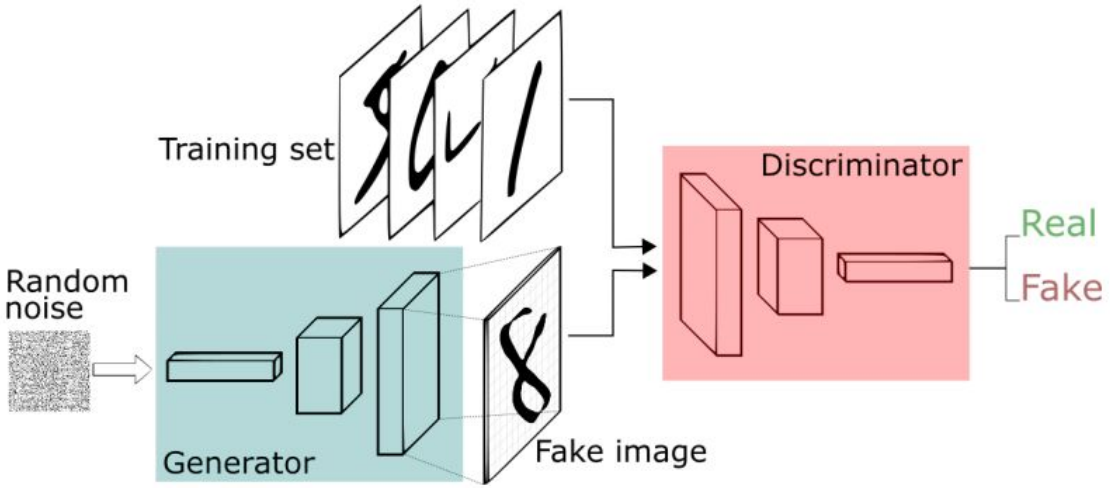
# Generative Adversarial Networks

# Generative Adversarial Networks

Idea → Train two models

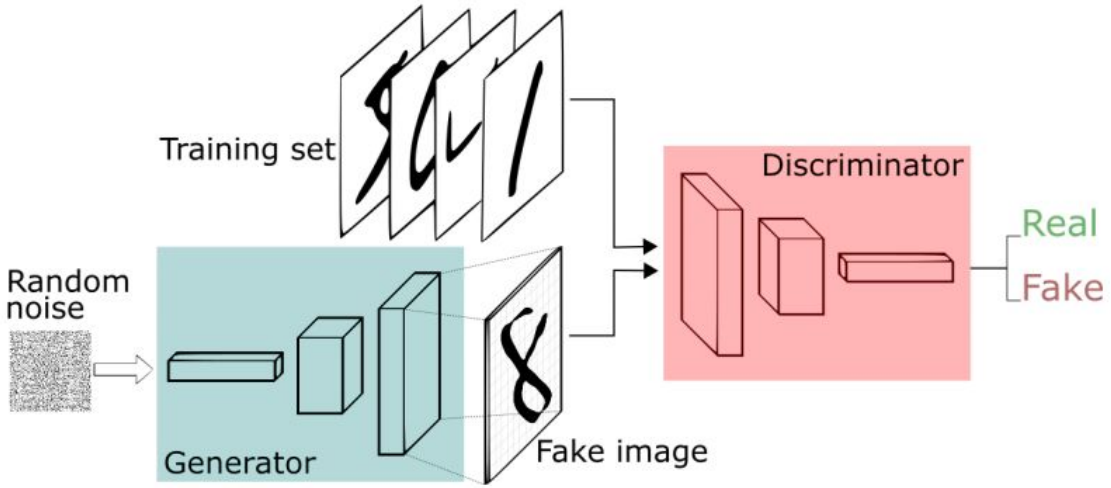
**Generator model** : try to fool the discriminator by generating real-looking images

**Discriminator model** : try to distinguish between real and fake images




**The loss function of the generator is defined by the discriminator!**

# Generative Adversarial Networks




**Generator network**

Input → A noise vector

Output → A generated image

**Discriminator network**

Input → An image

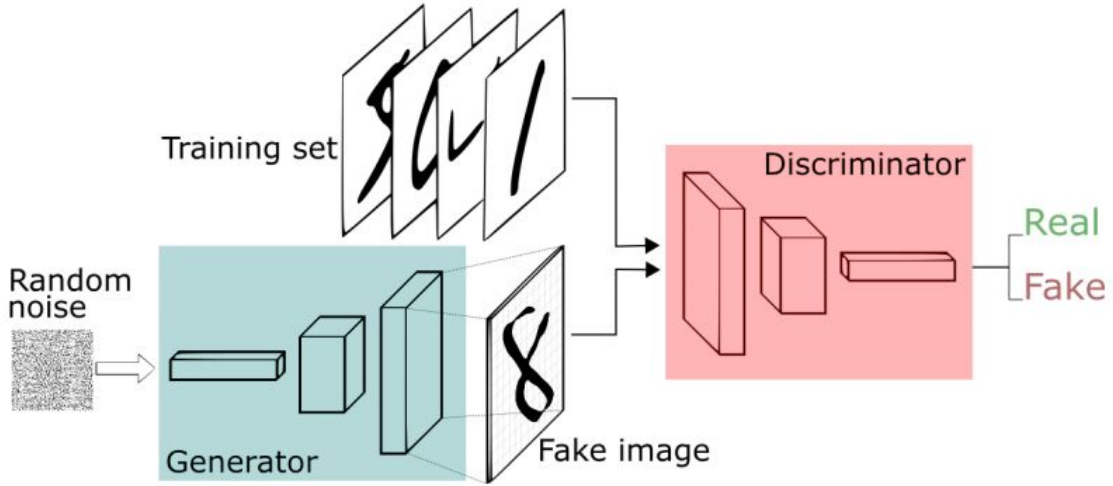
Output → A binary label (real vs fake)

# Generative Adversarial Networks

Play a **minmax** game:

The **discriminator** will try to do the best job it can

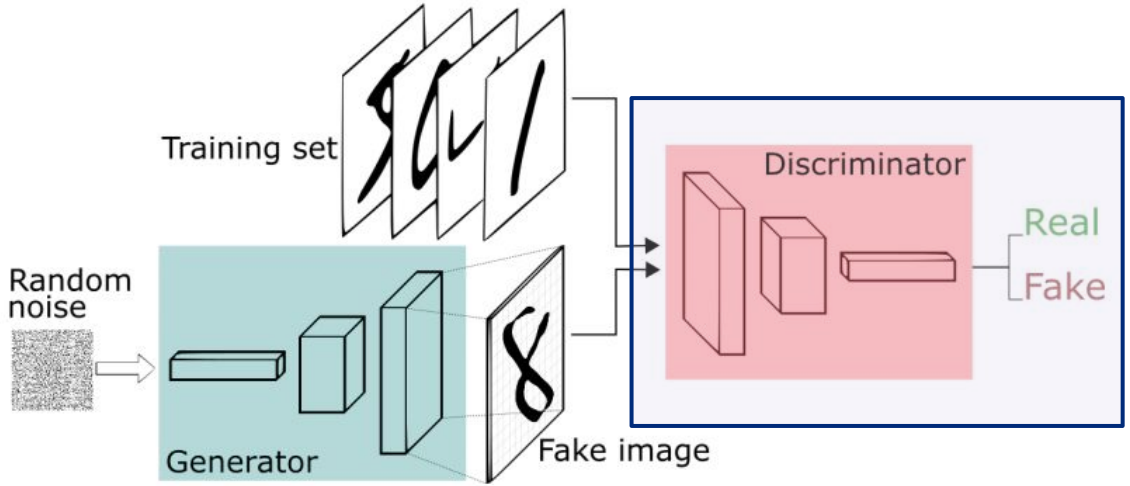
The **generator** is set to make the discriminator as wrong as possible



# Loss Function for MinMax Game

Learn **discriminator** weights to **maximize the probability** that it **labels a real image as real** and a **generated image as fake**

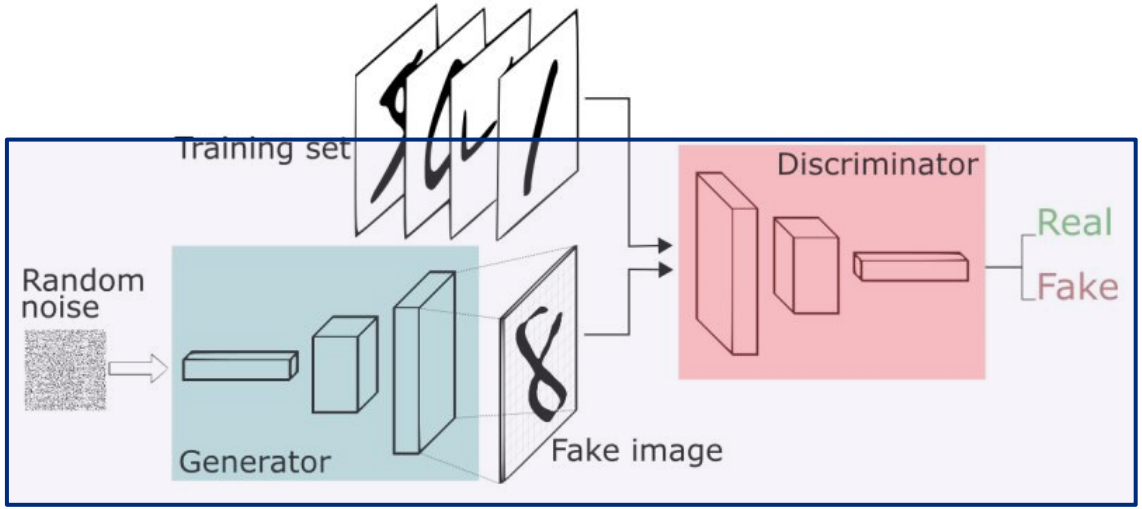
What loss function should we use? **Binary Cross-Entropy**



# Loss Function for MinMax Game

Learn **generator** weights to **maximize the probability** that the **discriminator labels a generated image as real**

What loss function should we use? **Discriminator**



# PyTorch Implementation

# PyTorch: Discriminator

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
          nn.Linear(28*28, 300),
          nn.LeakyReLU(0.2),
          nn.Linear(300, 100),
          nn.LeakyReLU(0.2),
          nn.Linear(100, 1))

    def forward(self, x):
        x = x.view(x.size(0), -1)
        out = self.model(x)
        return out.view(x.size(0))
```

# PyTorch: Generator

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
          nn.Linear(100, 300),
          nn.LeakyReLU(0.2),
          nn.Linear(300, 28*28),
          nn.Sigmoid())

    def forward(self, x):
        out = self.model(x).view(x.size(0), 1, 28, 28)
        return out.view(x.size(0))
```

# PyTorch: Training the Discriminator

```
def train_discrimintor(discriminator, generator, images):
    batch_size = images.size(0)
    noise = torch.randn(batch_size, 100)
    fake_images = generator(noise)
    inputs = torch.cat([images, fake_images])
    labels = torch.cat([torch.zeros(batch_size),    # Real
                        torch.ones(batch_size)])    # Fake
    outputs = discriminator(inputs)
    loss = criterion(outputs, labels)
    return outputs, loss
```

# PyTorch: Training the Generator

```
def train_generator(discriminator, generator, batch_size):
    batch_size = images.size(0)
    noise = torch.randn(batch_size, 100)
    fake_images = generator(noise)
    outputs = discriminator(fake_images)
    # Only looks at fake outputs
    # gets rewarded if we fool the discriminator!
    labels = torch.zeros(batch_size)
    loss = criterion(outputs, labels)
    return fake_images, loss
```

# Problems of Training GANs

# Vanishing Gradients

If the discriminator is too good, then the generator will not learn

Remember that we are using the discriminator as a loss function for the generator

If the discriminator is too good, small changes in the generator weights **won't change the discriminator output**

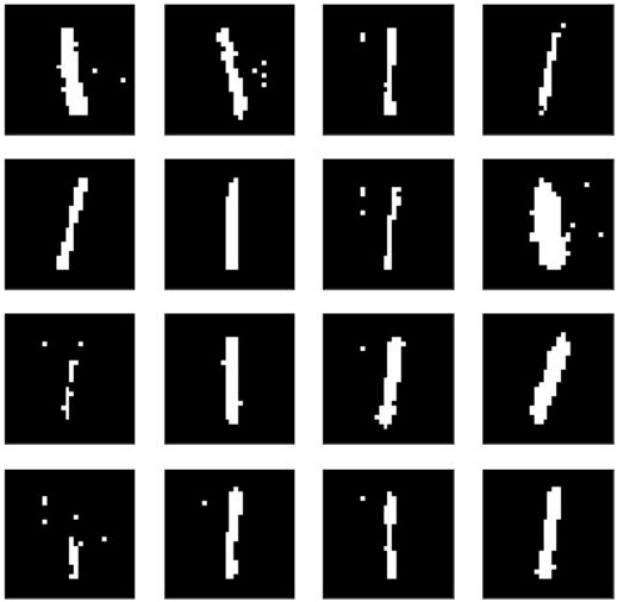
If small changes in generator weights make no difference, then we can't incrementally improve the generator **(no gradients!)**

# Mode Collapse

We want the generator to generate variety of outputs (e.g., all digits within MNIST).

If generator starts producing the **same output (or a small set of outputs)** , the best strategy for the discriminator is to reject that output.

However, if the discriminator is trapped in **local optimum** , it cannot adapt to generator, and the generator can fool it by only generating one type of data (e.g. only digit 1)



# Failing to Converge

Due to the MinMax optimization process, training Vanilla GANs is **very difficult** .

It is difficult to numerically see whether there is progress → Plotting the “training curve” doesn’t help much!

Takes a long time to train (a long time before we see progress)

To train GANs faster, we’ll use:

- LeakyReLU Activations instead of ReLU
- Batch Normalization
- Regularizing discriminator weights & adding noise to discriminator inputs

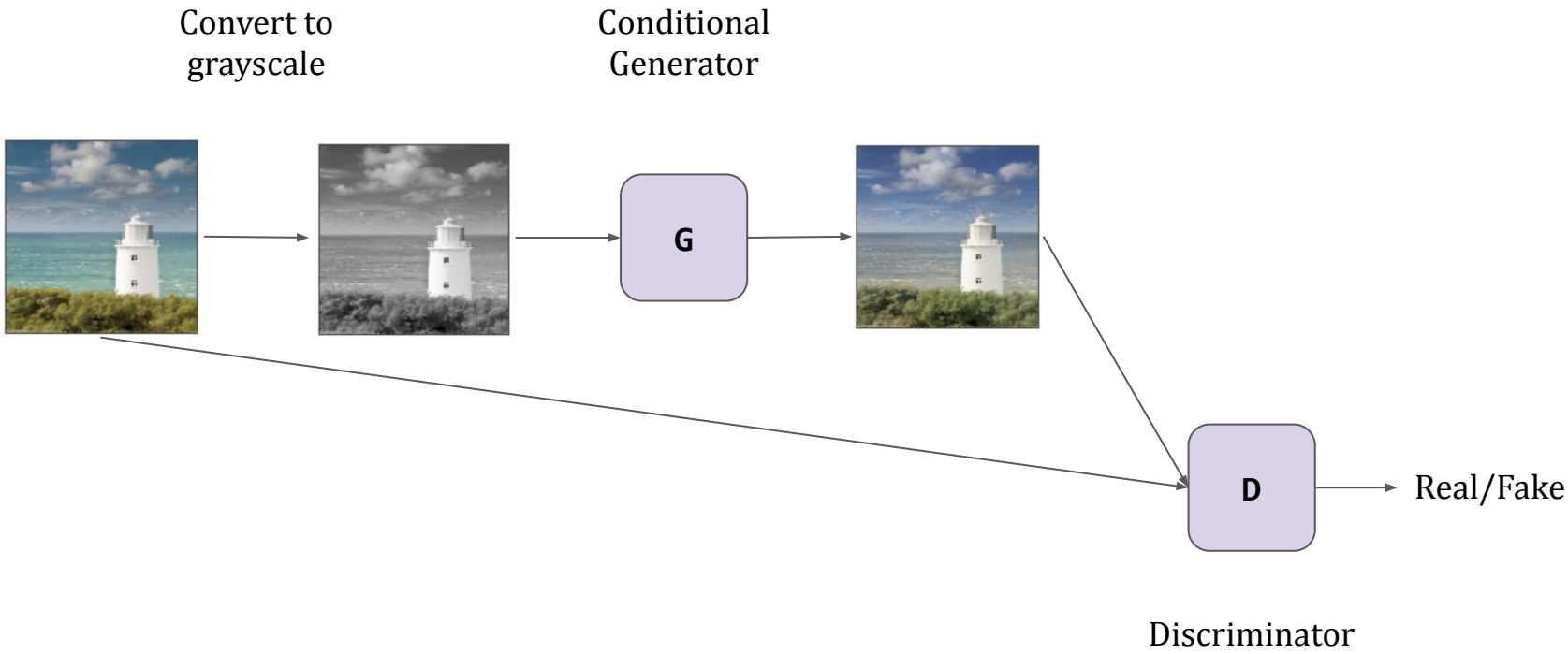
# Applications of GANs

# GANs in 2018 ...

https://arxiv.org/pdf/1812.04948.pdf

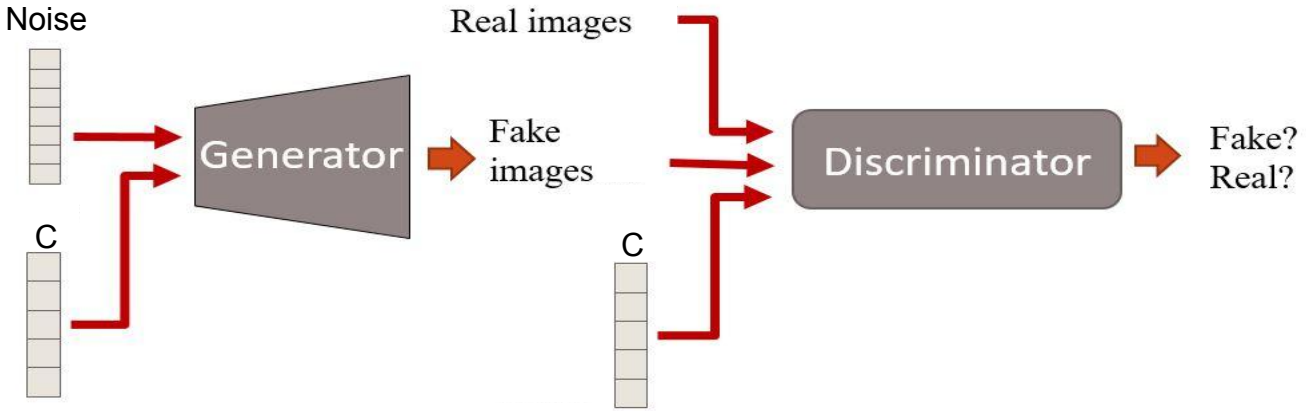
# Grayscale to Color


# Grayscale to Color



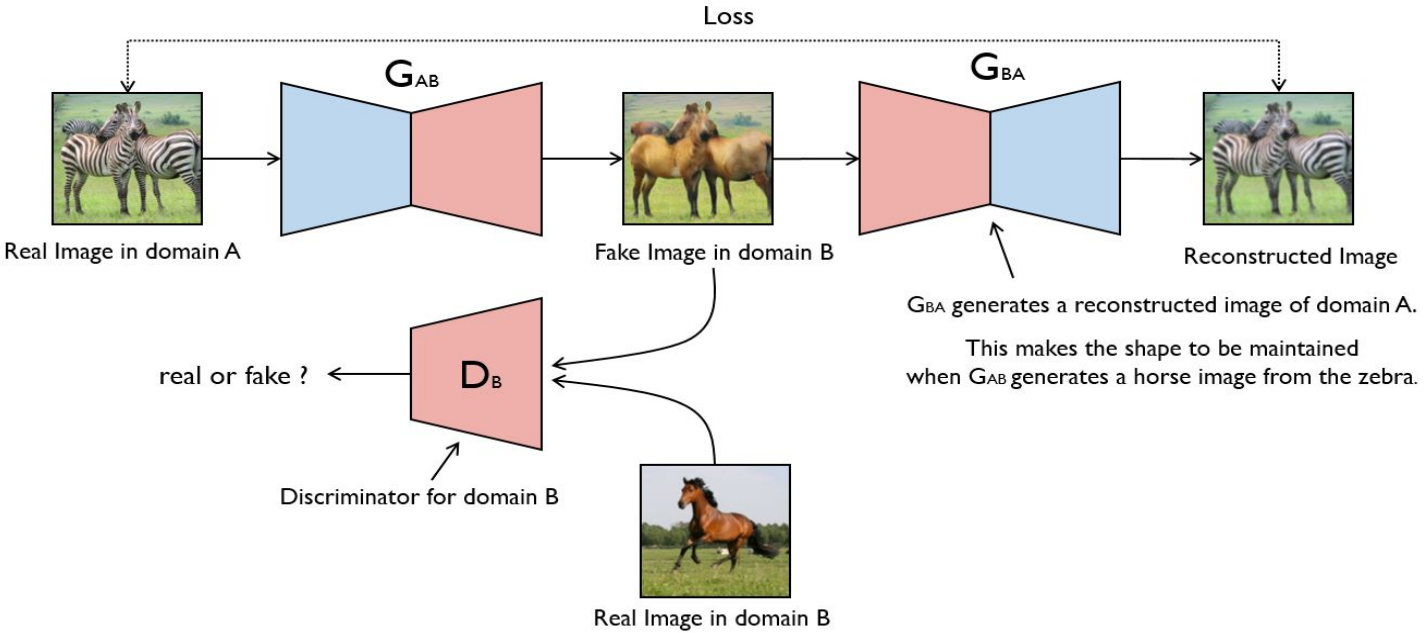
# Conditional Generation

How could we have a GAN trained on MNIST output only specific digits?

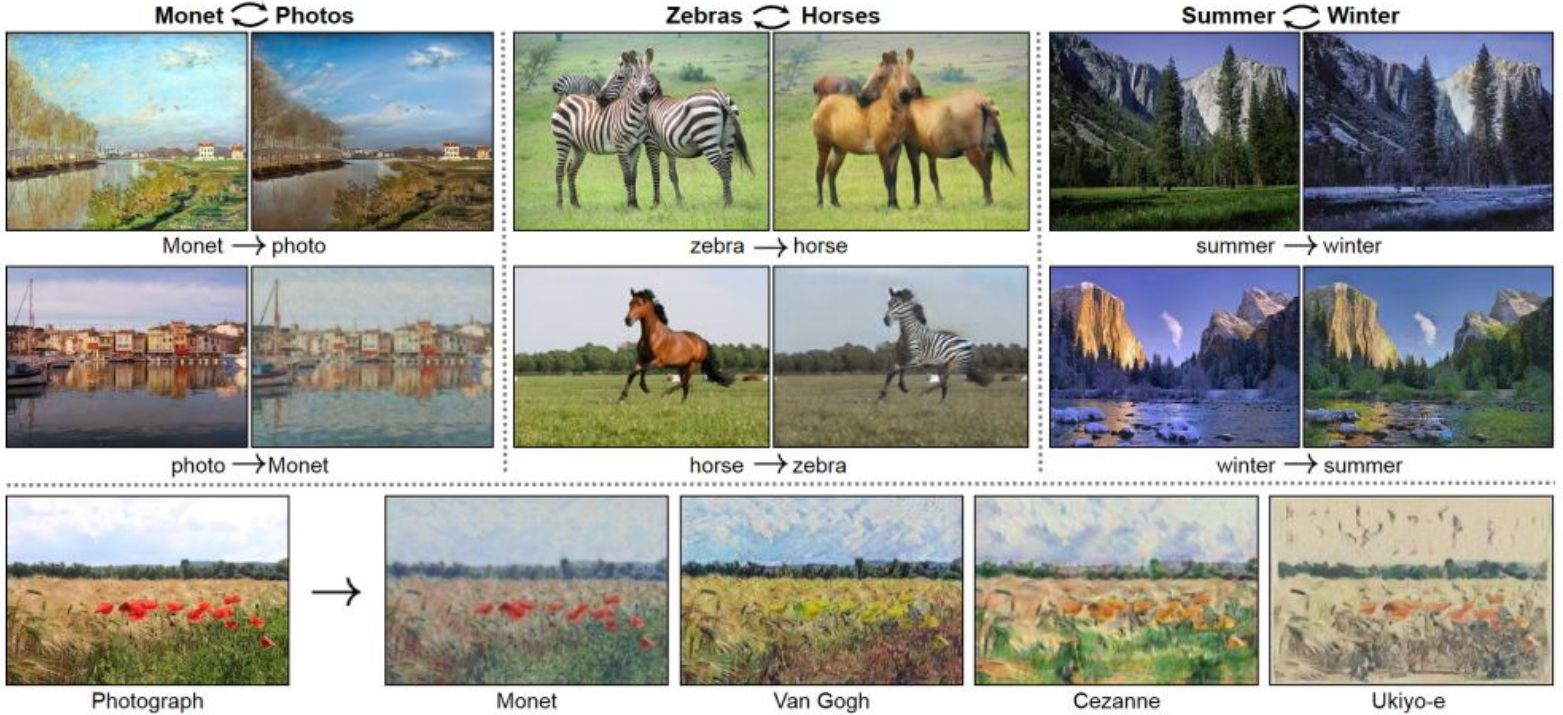


# Style Transfer

**Cycle GAN** : Cycle loss is reconstruction loss between input to cyclegan and output of cyclegan to ensure consistency



# Style Transfer



# Questions?