Implementation Tutorial: Image Generation and Image-to-Image Translation using GAN

Wu Hyun Shin

MLAI, KAIST

Overview

This tutorial is twofold as follows:

1. **Image Generation** using GAN (***DCGAN***) – 90 min.

2. **Image Translation** using GAN (***CycleGAN***) – 90 min.

*The codes are referring to one of the most-starred public GitHub repositories.
*Both the codes and the dataset for this tutorial will be provided by the instructor.
*The provided version may have been *slightly* modified from the original codes.

Environments

Prerequisites

- Linux or macOS
- Python 3
- NVIDA GPU + CUDA CuDNN
- PyTorch 1.0

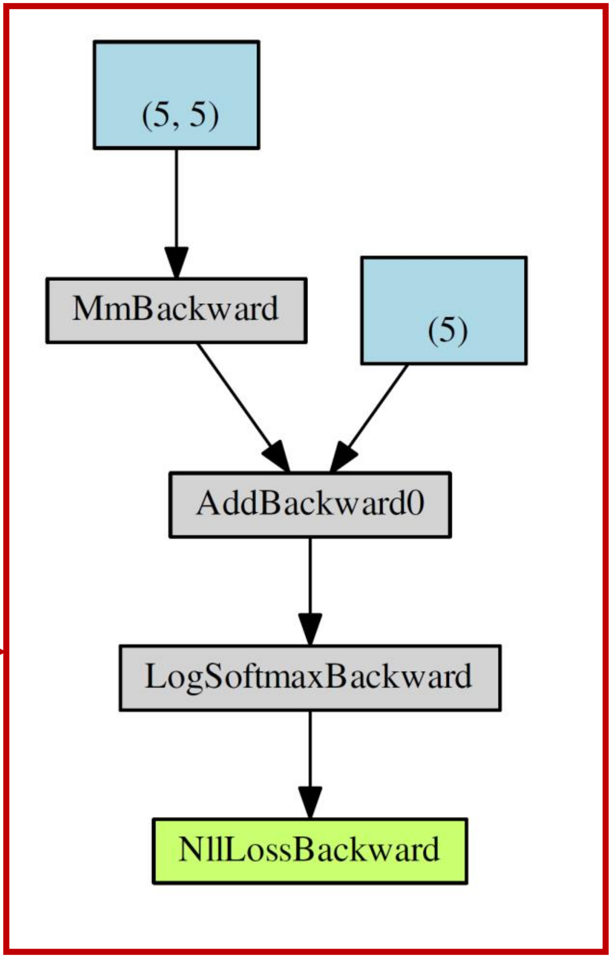
Github Repositories

- DCGAN: https://github.com/pytorch/examples/tree/master/dcgan
- CycleGAN: https://github.com/aitorzip/PyTorch-CycleGAN

Part 0:
Pytorch Autograd

Pytorch Autograd

```
import numpy as np
import torch
import torch.nn.functional as F
import torch.optim
from torchviz import make_dot

X = torch.tensor(np.random.randn(10, 5))  # batch size 10, feature dim 5
W = torch.tensor(np.random.randn(5, 5), requires_grad=True)  # weight
b = torch.tensor(np.random.randn(5), requires_grad=True)  # bias
# torch.randn()

y = X.matmul(W) + b
# torch.add(torch.matmul(X, W), b)

y_label = torch.tensor(np.random.randint(0, 5, 10))  # label for each datapoint
y = F.cross_entropy(y, y_label)

make_dot(y).render()

y.backward()
print(W.grad)
print(b.grad)

lr = 1.0
W.data.sub_(W.grad.data * lr)  # W = W - W.grad * lr
b.data.sub_(b.grad.data * lr)  # W = W - W.grad * lr
```



Pytorch Autograd

```python
import numpy as np
import torch
import torch.nn.functional as F
import torch.optim
from torchviz import make_dot

X = torch.tensor(np.random.randn(10, 5))  # batch size 10, feature dim 5
W = torch.tensor(np.random.randn(5, 5), requires_grad=True)  # weight
b = torch.tensor(np.random.randn(5), requires_grad=True)  # bias
# torch.randn()

y = X.matmul(W) + b
# torch.add(torch.matmul(X, W), b)

y_label = torch.tensor(np.random.randint(0, 5, 10))  # label for each datapoint
y = F.cross_entropy(y, y_label)

make_dot(y).render()

y.backward()
print(W.grad)
print(b.grad)

lr = 1.0
W.data.sub_(W.grad.data * lr)  # W = W - W.grad * lr
b.data.sub_(b.grad.data * lr)  # W = W - W.grad * lr
```

```python
import numpy as np
import torch
import torch.nn.functional as F
import torch.optim
import torch.nn as nn
from torchviz import make_dot


class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear = nn.Linear(5, 5)
        self.xent = torch.nn.CrossEntropyLoss()

    def forward(self, x, y_label):
        y = self.linear(x)
        loss = self.xent(y, y_label)
        return loss


x = torch.tensor(np.random.randn(10, 5)).float()
y_label = torch.tensor(np.random.randint(0, 5, 10)).long()

model = Model()
optim = torch.optim.SGD(model.parameters(), lr=1.0)

loss = model(x, y_label)
model.zero_grad()
loss.backward()
optim.step()
```

Part I:

Image Generation using GAN

(*DCGAN*)

Part I: Image Generation using GAN (DCGAN)

Get ready for the codes!

- https://github.com/pytorch/examples/tree/master/dcgan

```
# Clone with HTTPS.
# Suppose working path is:/st1/whshin/workspace/

whshin@ai2:/st1/whshin/workspace/$ git clone https://github.com/pytorch/examples.git

Cloning into 'examples'...
Username for 'https://github.com': ricoshin
Password for 'https://ricoshin@github.com': *****
remote: Enumerating objects: 1835, done.
remote: Total 1835 (delta 0), reused 0 (delta 0), pack-reused 1835
Receiving objects: 100% (1835/1835), 38.87 MiB | 3.89 MiB/s, done.
Resolving deltas: 100% (960/960), done.
Checking connectivity... done.

whshin@ai2:/st1/whshin/workspace/$ cd examples/dcgan
```

Or you can just download without signing in by using this button: Clone or download

Part I: Image Generation using GAN (DCGAN)

Get ready for the dataset!

- **CelebA** dataset (aligned version) : This is the one we will use today!

```
whshin@ai2:/st1/whshin/workspace/dcgan$ python datasets/download.py
```

- **LSUN** – bedroom (optional)
  - **NOTE**: For Python 3 compatibility, you should modify ***lsun/download.py***!

```
10: import urllib2

        ...

        ...

19: urllib2.urlopen(url)
```

```
try:
    # For Python 3.0 and later
    from urllib.request import urlopen
except ImportError:
    # Fall back to Python 2's urllib2
    from urllib2 import urlopen


urlopen(url)
```

*downloading will take about more than 7 hours..

Part I: Image Generation using GAN (DCGAN)

This is what you can see when you are ready.

```
# Code Explorer
-tutorial_gan/
 └dcgan/
   └datasets/
     └celebA/ # CelebA dataset
       └Img/
         └img_align_celeba/
           └000001.jpg
           └000002.jpg
           └   ...
     └download.py
   └output/
     └real_samples.png
     └fake_samples_epoch_000.png
     └         ...
     └netG_epoch_0.pth
     └netD_epoch_0.pth
     └         ...
   └main.py
 └cyclegan/
```

Part I: Image Generation using GAN (DCGAN)

How to run

You can run the program with this command:

```
whshin@ai2:/st1/whshin/workspace/dcgan$ python main.py --dataset folder --dataroot datasets/celebA/Img
--outf my_output --cuda
```

You can see the results that have generated beforehand:

```
-tutorial_gan/
 └dcgan/
   └output/
     └real_samples.png
     └fake_samples_epoch_000.png
     └          ...
     └netG_epoch_0.pth
     └netD_epoch_0.pth
     └          ...
```

Part I: Image Generation using GAN (DCGAN)

We only need to look no further than just 1 file: ***main.py***

- Let's briefly scan it!

```
import modules
parser.add_argument()
dataset, dataloader

def weights_init(m):

class Generator(nn.Module):
netG = Generator(ngpu).to(device)
netG.apply(weights_init)

class Discriminator(nn.Module):
netD = Discriminator(ngpu).to(device)
netD.apply(weights_init)

optimizerD, optimizerG

for epoch in range(opt.niter):
  for i, data in enumerate(dataloader, 0):
    # train!
```

Part I: Image Generation using GAN (DCGAN)

Module Import

```python
from __future__ import print_function
import argparse
import os
import random

import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data

import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
```

Note that you also have to install ***torchvision*** apart from the core torch package.

Part I: Image Generation using GAN (DCGAN)

Parsers – 17 arguments

- 15 optional arguments
- 2 required arguments: ***dataset***, ***dataroot***

```
parser = argparse.ArgumentParser()
parser.add_argument('--dataset', required=True,
                    help='cifar10 | lsun | mnist |imagenet | folder | lfw | fake')
parser.add_argument('--dataroot', required=True, help='path to dataset')
parser.add_argument('--workers', type=int, help='number of data loading workers', default=2)
parser.add_argument('--batchSize', type=int, default=64, help='input batch size')
parser.add_argument('--imageSize', type=int, default=64, # architecture must be changed to vary this.
                    help='the height / width of the input image to network')
parser.add_argument('--nz', type=int, default=100, help='size of the latent z vector')
parser.add_argument('--ngf', type=int, default=64) # Depth of feature maps carried through the G
parser.add_argument('--ndf', type=int, default=64) # Depth of feature maps propagated through the D
parser.add_argument('--niter', type=int, default=25, help='number of epochs to train for')
...
```

Part I: Image Generation using GAN (DCGAN)

Parsers – 17 arguments

- 15 optional arguments
- 2 required arguments: ***dataset***, ***dataroot***

```
...
parser.add_argument('--lr', type=float, default=0.0002, help='learning rate, default=0.0002')
parser.add_argument('--beta1', type=float, default=0.5, help='beta1 for adam. default=0.5')
parser.add_argument('--cuda', action='store_true', help='enables cuda')
parser.add_argument('--ngpu', type=int, default=1, help='number of GPUs to use') # If 0, use CPU
parser.add_argument('--netG', default='', help="path to netG (to continue training)")
parser.add_argument('--netD', default='', help="path to netD (to continue training)")
parser.add_argument('--outf', default=‘output', help='folder to output images and model checkpoints')
parser.add_argument('--manualSeed', type=int, help='manual seed')
```

Now, we can understand what this command meant:

```
whshin@ai2:/st1/whshin/workspace/dcgan$ python main.py --dataset folder --dataroot datasets/celebA/Img
--outf my_output --cuda
```

Part I: Image Generation using GAN (DCGAN)

Reproducibility

```python
# parser.add_argument('--manualSeed', type=int, help='manual seed')

if opt.manualSeed is None:
  opt.manualSeed = random.randint(1, 10000)
print("Random Seed: ", opt.manualSeed)
random.seed(opt.manualSeed)
torch.manual_seed(opt.manualSeed)
```

Reproducibility can be more crucial when it comes to GAN frameworks due to the instability in convergence.

By ***manually*** setting the random seed, we can guarantee ***reproducible*** results.

- But it is true only on the same platform and PyTorch release.
- e.g. reproducibility between CPU and GPU execution need not be guaranteed.

Part I: Image Generation using GAN (DCGAN)

Reproducibility (Supplementary)

According to the PyTorch Docs, when running on the CuDNN backend, you have to make the model deterministic as follows:

```
# When running on the CuDNN backend, two further options must be set.
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False # this is set to True in the code, which is contradictory.
```

- **NOTE**: Deterministic mode can have ***performance impact***, depending on your model.

PyTorch

What does torch.backends.cudnn.benchmark do?

fmassa Francisco Massa Aug '17

It enables benchmark mode in cudnn.
benchmark mode is good whenever your input sizes for your network do not vary. This way, cudnn will look for the optimal set of algorithms for that particular configuration (which takes some time). This usually leads to faster runtime.
But if your input sizes changes at each iteration, then cudnn will benchmark every time a new size appears, possibly leading to worse runtime performances.

*https://discuss.pytorch.org/t/what-does-torch-backends-cudnn-benchmark-do/5936

Part I: Image Generation using GAN (DCGAN)

Reproducibility (Supplementary)

And if you are using Numpy, you should do this as well:

```
# Plus, if you (or any libraries you're using) rely on Numpy:
numpy.random.seed(opt.manualSeed)
```

There are a set of funcs that can manually set the seed for different device scopes:

```
# This is the one you can see in the code.
torch.manual_seed(opt.manualSeed) # for all devices (both CPU and CUDA)

# These commands will be silently ignored when we are not using CUDA.
torch.cuda.manual_seed(opt.manualSeed) # for the current GPU. (Silently ignored when not using GPU)
torch.cuda.manual_seed_all(opt.manualSeed) # for all the GPUs. (Silently ignored when not using GPU)
```

Part I: Image Generation using GAN (DCGAN)

Data Loading and Processing

Next, we are going to gear up for the data.

```
if opt.dataset in ['imagenet', 'folder', 'lfw’]:
  dataset = ...
  nc = 3
elif opt.dataset == 'lsun’:
  dataset = ...
  nc = 3
elif opt.dataset == 'cifar10’:
  dataset = ...
  nc = 3
elif opt.dataset == 'mnist’:
  dataset = ...
  nc = 1
elif opt.dataset == 'fake’:
  dataset = ...
  nc = 3
```

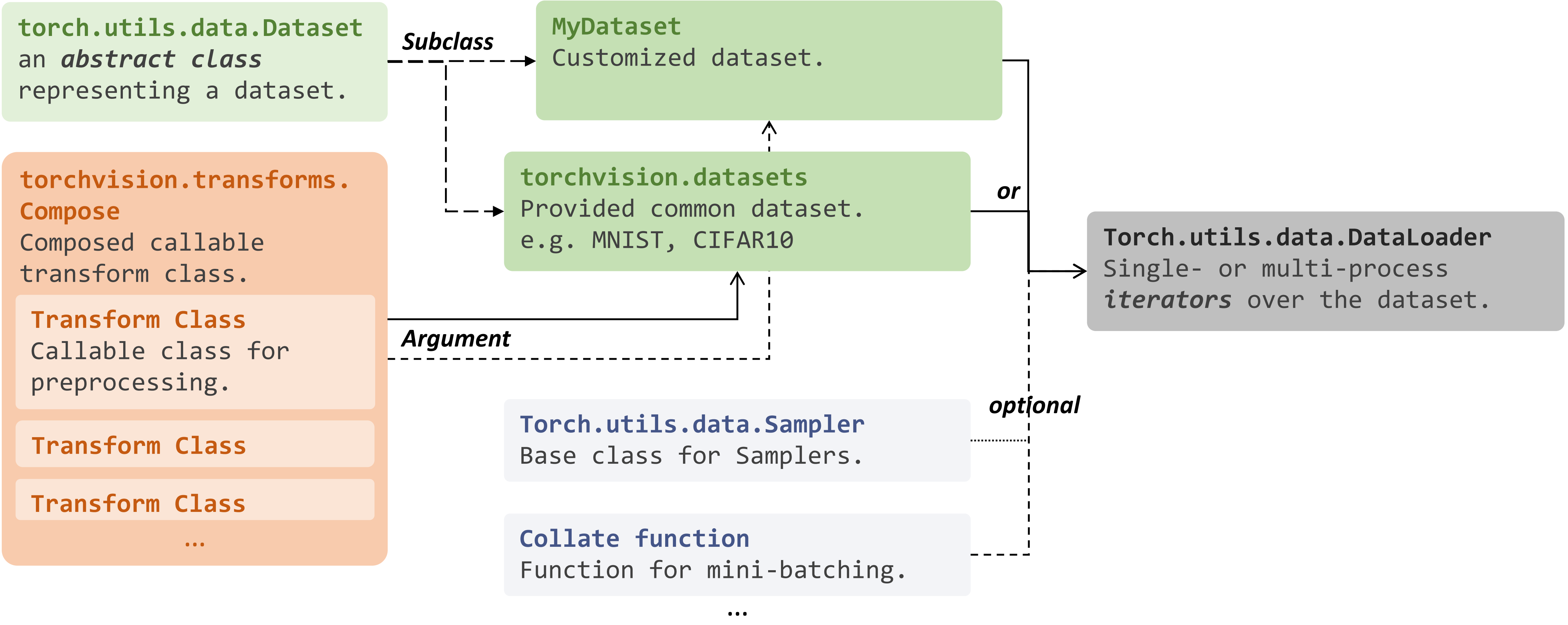
And they we would like to treat them differently according to the name.

So the next question is, “how can we handle those data more efficiently?”

Part I: Image Generation using GAN (DCGAN)

Data Loading and Processing

Let's see how to **load** and **preprocess** data using the tools PyTorch package provides.



Part I: Image Generation using GAN (DCGAN)

Data Loading and Processing: *Dataset*

- We can handle our data more easily by using ***torch.utils.data.Dataset***.

```
torch.utils.data.Dataset is an abstract class representing a dataset.
Your custom dataset should inherit Dataset and override the following methods:
    • __len__ so that len(dataset) returns the size of the dataset.
    • __getitem__ to support the indexing such that dataset[i] can be used to get i-th sample
```

- We can make our custom dataset by subclassing this abstract class:

```python
from torch.utils.data import Dataset
class MyDataset(Dataset):
   ...
dataset = MyDataset(...)
```

- Or, you can just use ***torchvision*** package providing some common datasets:

```python
import torchvision.datasets
# All datasets in this package are subclasses of torch.utils.data.Dataset
dataset = datasets.MNIST(root=opt.dataroot)
```

Part I: Image Generation using GAN (DCGAN)

Data Loading and Processing: *Transforms*

- It is recommendable to use a callable transform class to preprocess the data.

```
# Make custom class
class Transform(Dataset):
  def __init__(self, *args, **kwargs):
    ...
  def __call__(self, sample):
    ...
transform = Transform(*init_args_list)
transformed_sample = transform(sample)
```

or

```
# Load transforms on PIL Image
import torchvision.transforms as transforms

transform = transforms.CenterCrop(*args)
# transform = transforms.ColorJitter(*args)
# transform = transforms.Grayscale(*args)

transformed_sample = transform(sample)
```

- These callable transforms can be merged into a single transform as follows:

```
import torchvision.transforms as transforms

composed_transform = transforms.Compose(
  [Transform_1(*init_arg_1), Transform_2(*init_arg_2), ... , Transform_n(*init_arg_n)])

transformed_sample = composed_transform(sample)
```

- After all, a (composed) transform class can be passed as an argument for:

```
dataset = datasets.MNIST(root=opt.dataroot, transform=composed_transform)
```

Part I: Image Generation using GAN (DCGAN)

Data Loading and Processing: *DataLoader*

- Now, you can iterate through the processed dataset with simple for loop.

```
dataset = datasets.MNIST(root=opt.dataroot, transform=composed_transform)

for i in range(len(dataset)):
  sample = dataset[i]
  do_something(sample)
        ...
```

- The more sophisticated way of doing that is to use ***Torch.utils.data.DataLoader***, which is an iterator that can help with ***batching***, ***shuffling***, and ***multiprocessing***.

```
from torch.utils.data import DataLoader

dataset = datasets.MNIST(root=opt.dataroot, transform=composed_transform)
dataloader = DataLoader(dataset, batch_size=opt.batchSize, shuffle=True, num_workers=int(opt.workers))

for i, batch in enumerate(dataloader):
  do_something(batch)
        ...
```

Part I: Image Generation using GAN (DCGAN)

Data Loading and Processing: ***ImageFolder***

Let's see what we are going to do with our Celeb-A dataset.

- We will use ***torchvision.datasets.ImageFolder***, which is a generic data loader where the images are arranged in subdiretories.

```
import torchvision.datasets

dataset = datasets.ImageFolder(root=dataroot, transform=composed_transforms)
```

CLASS `torchvision.datasets.ImageFolder(root, transform=None, target_transform=None, loader=<function default_loader>) -> Returns (sample, target)`

A generic data loader where the images are arranged in this way:

```
root/class_x/001.ext
root/class_x/002.ext
        ...

root/class_y/aaa.ext
root/class_y/bbb.ext
        ...
```

Part I: Image Generation using GAN (DCGAN)

Module implementation: *Sequential*

```
class Generator(nn.Module):
  def __init__(self, ngpu):
    super(Generator, self).__init__()
    self.ngpu = ngpu
    self.main = nn.Sequential(
      #   nn.ConvTranspose2d(...),
      #   nn.BatchNorm2d(...),
      #   ...
    )
  def forward(self, input):
    return self.main(input)
```

```
class Discriminator(nn.Module):
  def __init__(self, ngpu):
    super(Discriminator, self).__init__()
    self.ngpu = ngpu
    self.main = nn.Sequential(
      #   nn.Conv2d(...),
      #   nn.LeakyReLU(...),
      #   ...
    )
  def forward(self, input):
    return self.main(input)
```

CLASS `torch.nn.Sequential(*args)`

A sequential container. Modules will be added to it in the order they are passed in the constructor. Alternatively, an ordered dict of modules can also be passed in.

Part I: Image Generation using GAN (DCGAN)

Module implementation: *Data Parallelism*

```python
class Generator(nn.Module):
  def __init__(self, ngpu):
    ...
  def forward(self, input):
    if input.is_cuda and self.ngpu > 1:
      output = nn.parallel.data_parallel(
        self.main, input, range(self.ngpu))
    else:
      output = self.main(input)
    return output
```

```python
class Discriminator(nn.Module):
  def __init__(self, ngpu):
    ...
  def forward(self, input):
    if input.is_cuda and self.ngpu > 1:
      output = nn.parallel.data_parallel(
        self.main, input, range(self.ngpu))
    else:
      output = self.main(input)
    return output.view(-1, 1).squeeze(1)
```

CLASS `torch.nn.parallel.`**`data_parallel`**`(module, inputs, device_ids=None, output_device=None, ... )`

Functional version of ***torch.nn.DataParallel***.

CLASS `torch.nn.DataParallel(module, device_ids=None, output_device=None, ... )`

1. Split the input across the specified devices by chunking in batch dimension
2. In the forward pass, the module is replicated on each device, and each replica handles a portion of the input.
3. During the backwards pass, gradients from each replica are summed into the original modules.

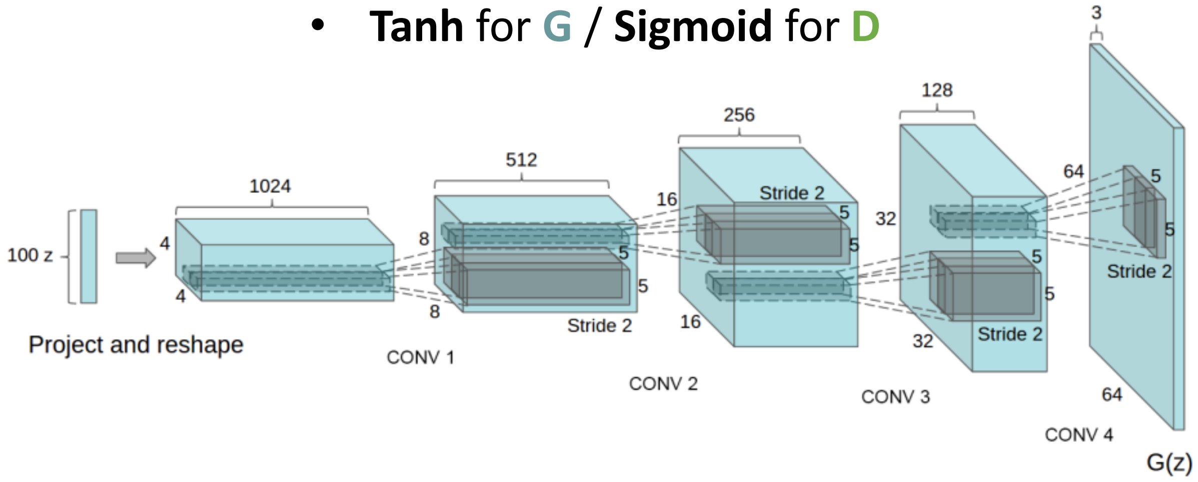
Part I: Image Generation using GAN (DCGAN)

Module implementation: *Generator*

```
# input is Z, going into a convolution
nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
nn.BatchNorm2d(ngf * 8),
nn.ReLU(True),
# state size. (ngf*8) x 4 x 4
nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
nn.BatchNorm2d(ngf * 4),
nn.ReLU(True),
# state size. (ngf*4) x 8 x 8
nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
nn.BatchNorm2d(ngf * 2),
nn.ReLU(True),
# state size. (ngf*2) x 16 x 16
nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
nn.BatchNorm2d(ngf),
nn.ReLU(True),
# state size. (ngf) x 32 x 32
nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
nn.Tanh()
# state size. (nc) x 64 x 64
```

Checklist

- No **pooling** & **FC** layers
- **BatchNorm** for all layers except the output(G) or input(D) layer
- **ReLU** for G / **LeakyReLU** for D
- **Tanh** for G / **Sigmoid** for D




```
CLASS torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=1, , ... , bias=True)
CLASS torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
CLASS torch.nn.ReLU(inplace=False)
```

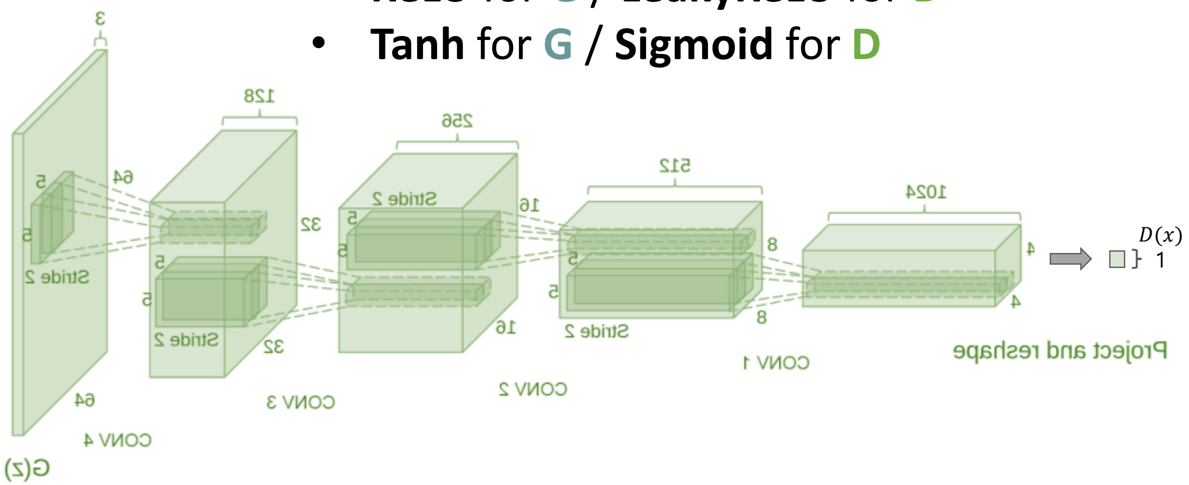
Part I: Image Generation using GAN (DCGAN)

Module implementation: *Discriminator*

```
# input is (nc) x 64 x 64
nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
nn.LeakyReLU(0.2, inplace=True),
# state size. (ndf) x 32 x 32
nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
nn.BatchNorm2d(ndf * 2),
nn.LeakyReLU(0.2, inplace=True),
# state size. (ndf*2) x 16 x 16
nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
nn.BatchNorm2d(ndf * 4),
nn.LeakyReLU(0.2, inplace=True),
# state size. (ndf*4) x 8 x 8
nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
nn.BatchNorm2d(ndf * 8),
nn.LeakyReLU(0.2, inplace=True),
# state size. (ndf*8) x 4 x 4
nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
nn.Sigmoid()
```

Checklist

- No **pooling** & **FC** layers
- **BatchNorm** for all layers except the output(G) or input(D) layer
- **ReLU** for G / **LeakyReLU** for D
- **Tanh** for G / **Sigmoid** for D




```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, , ... , bias=True)
CLASS torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
CLASS torch.nn.ReLU(negative_slope=0.01, inplace=False)
```

Part I: Image Generation using GAN (DCGAN)

Weight initialization

```
# custom weights initialization called on netG and netD
def weights_init(m):
  classname = m.__class__.__name__
  if classname.find('Conv') != -1:
    m.weight.data.normal_(0.0, 0.02)
  elif classname.find('BatchNorm') != -1:
    m.weight.data.normal_(1.0, 0.02)
    m.bias.data.fill_(0)

netG = Generator(ngpu).to(device)
netG.apply(weights_init)

netD = Discriminator(ngpu).to(device)
netD.apply(weights_init)
```

"All weights were initialized from a **zero-centered** Normal distribution with **standard deviation 0.02**."

Batch Normalization

weight

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

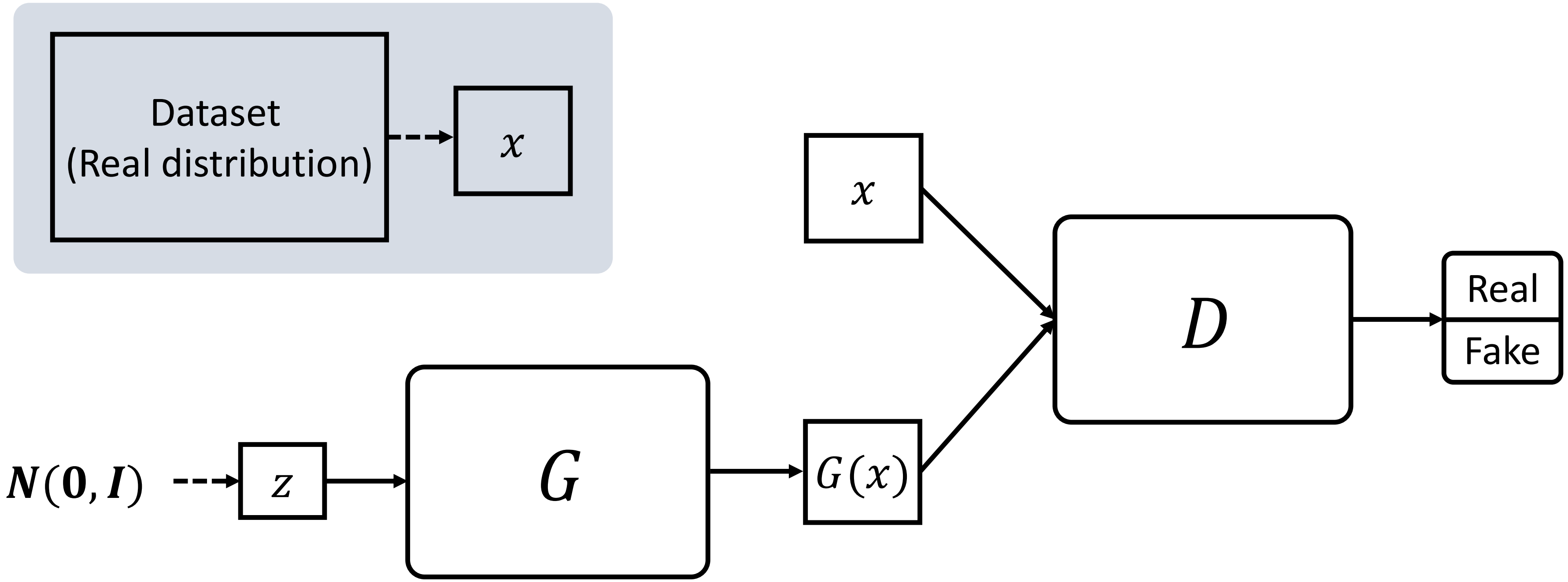
bias

CLASS `torch.nn.Module.apply(fn)`

Applies **fn** recursively to every submodule (as returned by .children()) as well as self.
Typical use includes initializing the parameters of a model (see also torch.nn.init).

Part I: Image Generation using GAN (DCGAN)

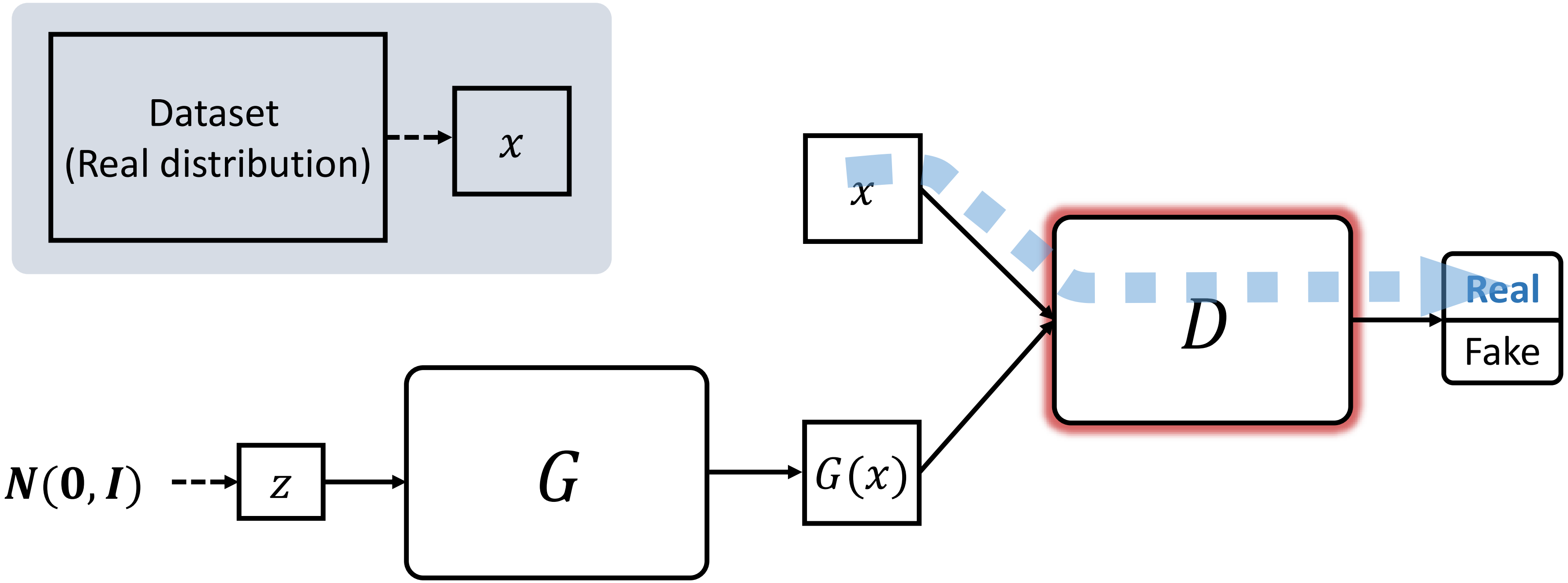
Recap: Overview



$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Part I: Image Generation using GAN (DCGAN)

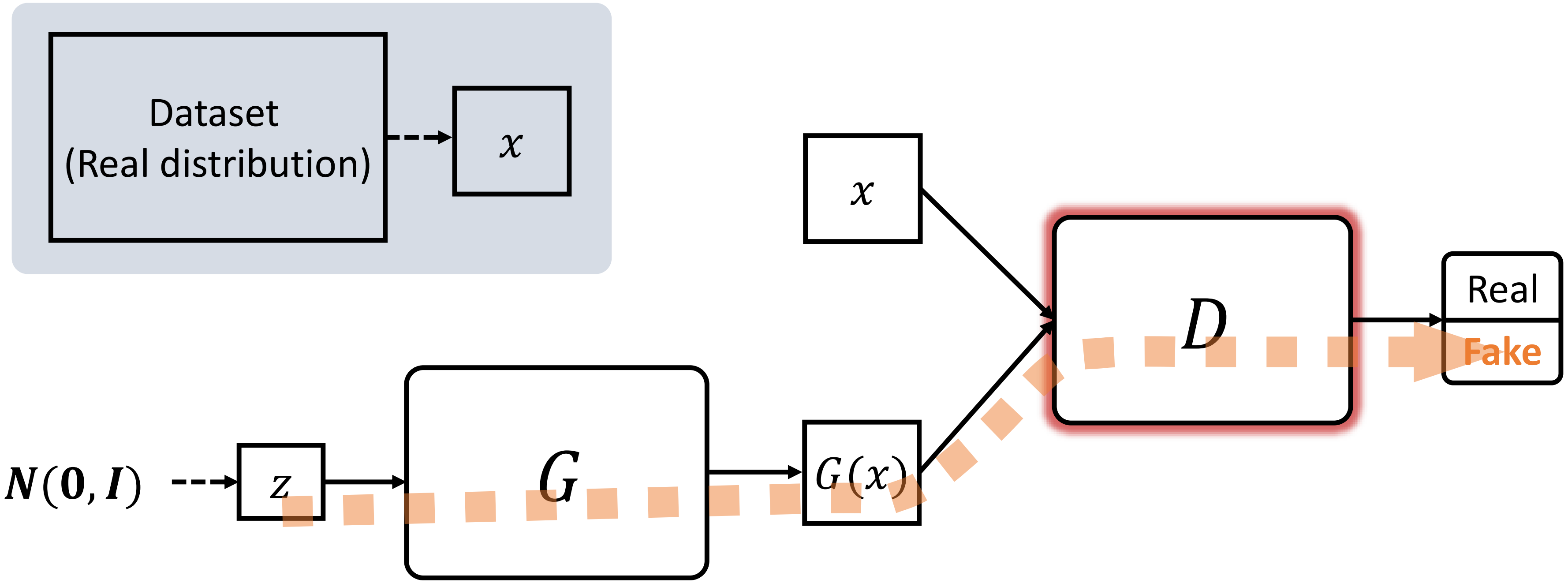
Recap: Training of *Discriminator* *(Real → Real)*



$$\min_{G} \boxed{\max_{D}} V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Part I: Image Generation using GAN (DCGAN)

Recap: Training of *Discriminator* *(Fake → Fake)*



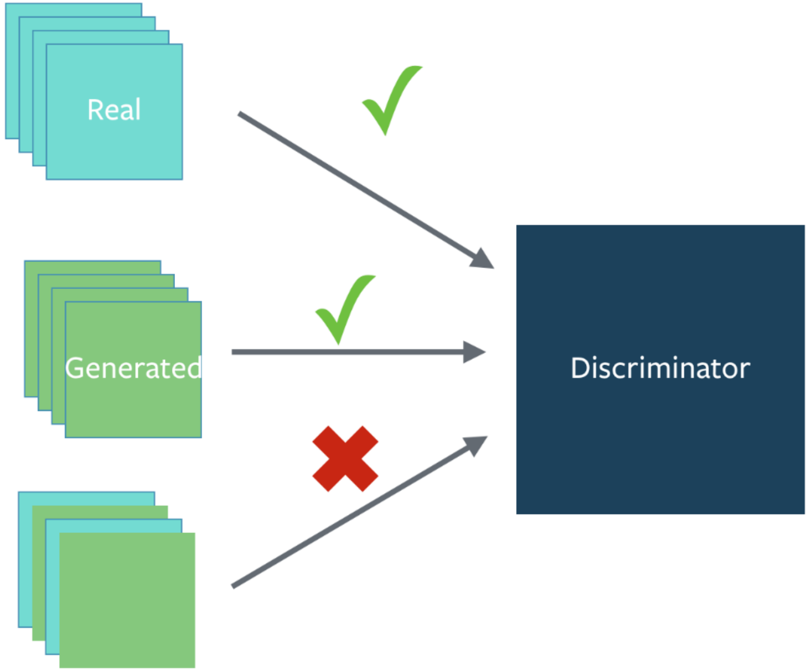
$$\min_G \boxed{\max_D} V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Part I: Image Generation using GAN (DCGAN)

Recap: Training of *Discriminator* *(Fake → Fake)*

We will abide by some of the best practice shown in:

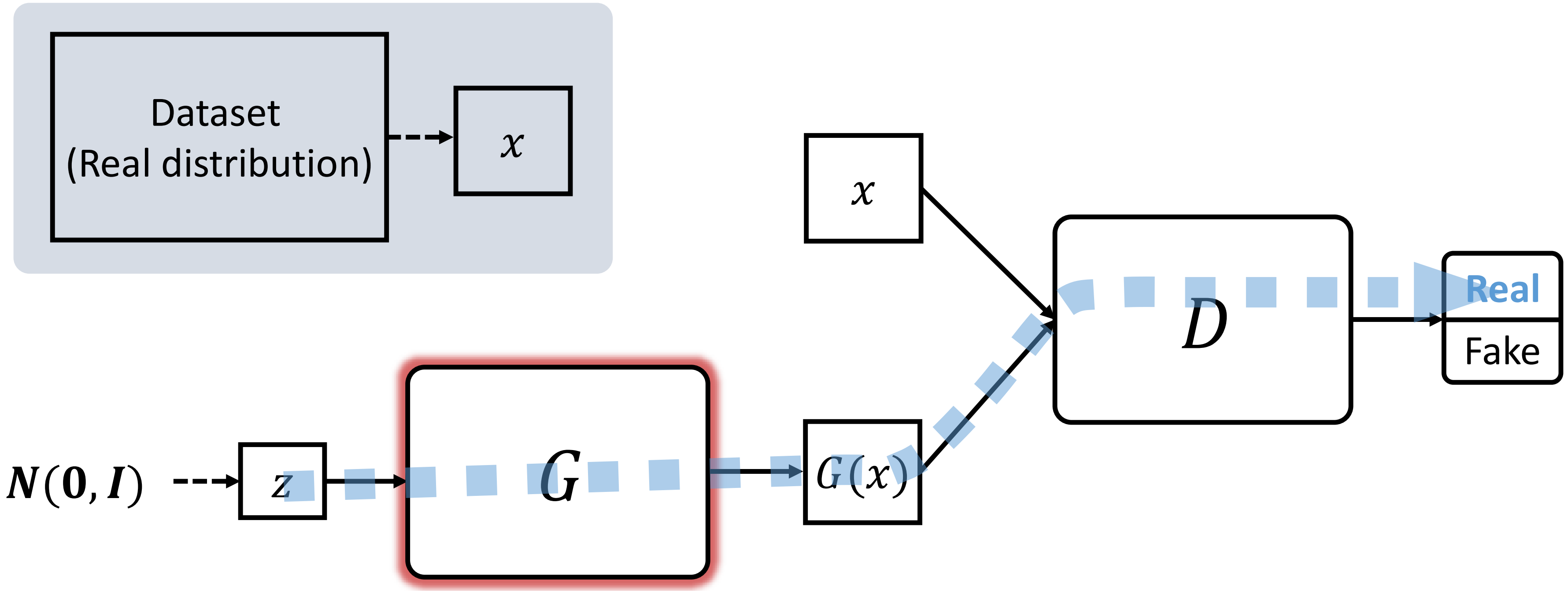
https://github.com/soumith/ganhacks



Constructing different mini-batches for real and fake is known as better practice.

Part I: Image Generation using GAN (DCGAN)

Recap: Training of *Generator* *(Fake → Real)*



$$\boxed{\min_G} \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Part I: Image Generation using GAN (DCGAN)

Loss function and Optimization

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Discriminator

prob. of D predicting that real data is genuine.

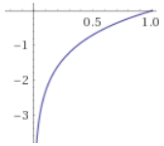
*prob. of D predicting that fake data is **NOT** genuine.*

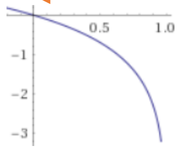
Maximize : $\log(D_1(x)) + \log(1 - D_2(G(z)))$

max

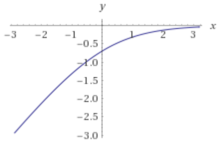
max

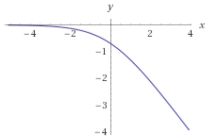
$\log(x)$





$\log(sgm(x))$



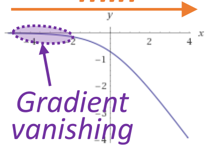


Generator

*prob. of D predicting that fake data is **NOT** genuine.*

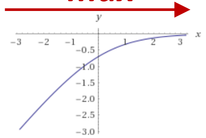
Minimize : $\log(1 - D_2(G(z)))$

min



Maximize : $\log(D_2(G(z)))$

max



prob. of D predicting that fake data is genuine.

: D rejecting G(x) with high confidence.

Part I: Image Generation using GAN (DCGAN)

Loss function and Optimization

We will use the Binary Cross Entropy loss function(***torch.nn.BCELoss***).

CLASS `torch.nn.BCELoss(weight=None, size_average=None, reduce=None, reduction='mean')`

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -\left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)\right]$$

We can specify which part of the equation to use with the label y.

Real label: y = 1
Fake label: y = 0

$$l_n = -\left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)\right]$$

*We use this part, when label is **real*** (for $\log x_n$)

*We use this part, when label is **fake*** (for $\log(1 - x_n)$)

Discriminator

Maximize: $\log(D_1(x)) + \log(1 - D_2(G(z)))$

```
criterion = nn.BCELoss()
errD_real = criterion(real_input, real_label)
errD_fake = criterion(fake_input, fake_label)
errD = errD_real + errD_fake
```

Generator

Maximize: $\log(D_2(G(z)))$

```
criterion = nn.BCELoss()
errG = criterion(fake_input, real_label)
# errG = (-1) * criterion(fake_input, fake_label)
```

Part I: Image Generation using GAN (DCGAN)

Loss function

```
netD = Discriminator(ngpu).to(device)
netG = Generator(ngpu).to(device)
netD.apply(weights_init)
netG.apply(weights_init)
optimD = optim.Adam(netD.parameters(), ... )
optimG = optim.Adam(netG.parameters(), ... )
```

```
criterion = nn.BCELoss()
real_label = torch.full((input.size(0),), 1)
fake_label = torch.full((input.size(0),), 0)
for epoch in range(opt.niter):
  for i, data in enumerate(dataloader, 0):
    # training loop
```

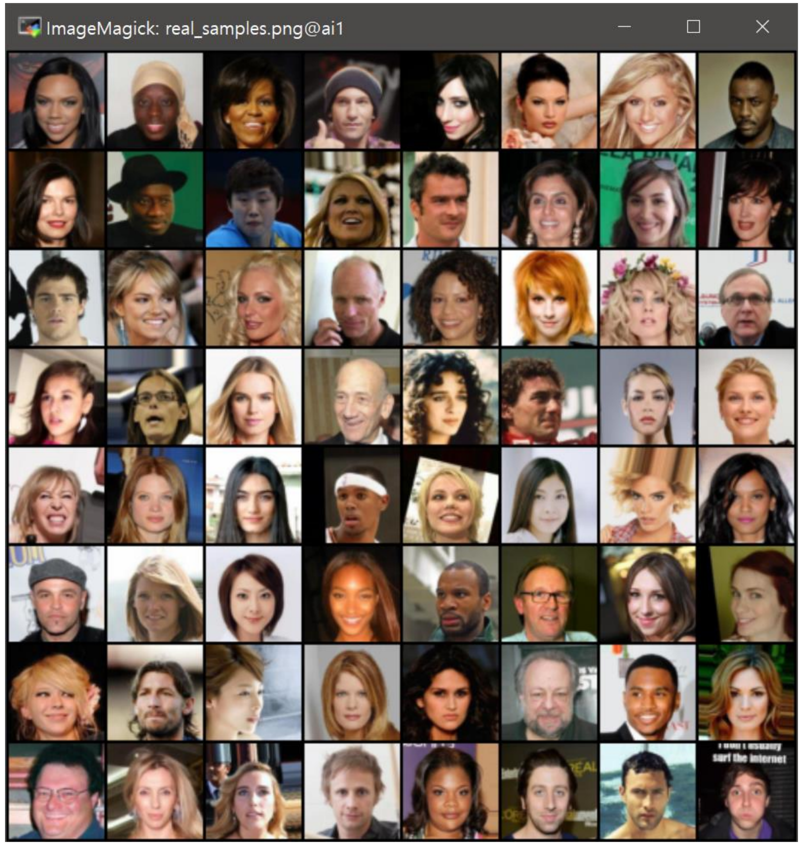
```
    # train D: max log(D(x)) + log(1 - D(G(z)))
    #############################################
    # train with real
    output = netD(input)
    errD_real = criterion(output, real_label)
    errD_real.backward()
    # train with fake
    fake = netG(torch.randn(batch_size, nz, 1, 1)
    output = netD(fake.detach())
    errD_fake = criterion(output, fake_label)
    errD_fake.backward()
    # update network
    optimD.step()
```

```
    # train G: maximize log(D(G(z)))
    #############################################
    fake = netG(torch.randn(batch_size, nz, 1, 1))
    output = netD(fake)
    errG = criterion(output, real_label)
    errG.backward()
    # update network
    optimG.step()
```

Part I: Image Generation using GAN (DCGAN)

Result

We can check the qualitative results under the path: ***./output*** (default)



Real samples

Fake samples(25 epochs)

Part II:
Image Translation using GAN
(*CycleGAN*)

Part II: Image Translation using GAN (CycleGAN)

Get ready for the codes,

- https://github.com/aitorzip/PyTorch-CycleGAN

```
# Clone with HTTPS.
# Suppose working path is:/st1/whshin/workspace/

whshin@ai2:/st1/whshin/workspace/$ git clone https://github.com/aitorzip/PyTorch-CycleGAN.git
Cloning into ' PyTorch-CycleGAN '...
**omitted**
Checking connectivity... done.

# Just for brevity.
whshin@ai2:/st1/whshin/workspace/$ mv PyTorch-CycleGAN/ cyclegan

whshin@ai2:/st1/whshin/workspace/$ cd cyclegan
```

and the dataset!

- **horse2zebra** : This is the one we will use today!

```
whshin@ai2:/st1/whshin/workspace/cyclegan$ ./download_dataset horse2zebra
```

Part II: Image Translation using GAN (CycleGAN)

This is what you can see when you are ready.

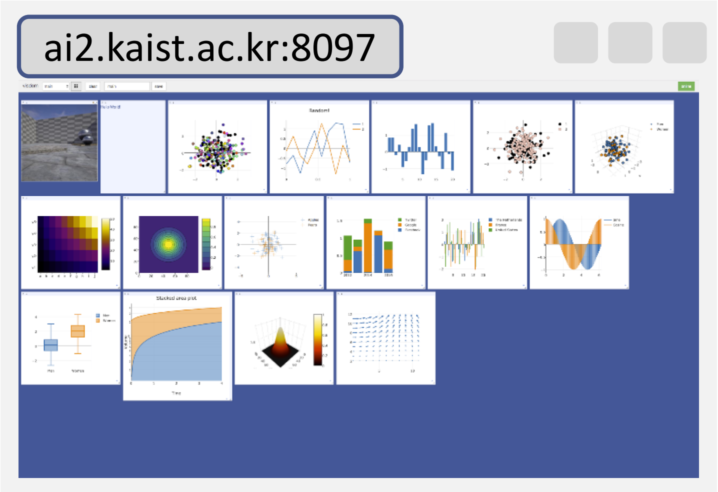
```
# Code Explorer
-tutorial_gan/
  └dcgan/
  └cyclegan/
    └datasets/
      └horse2zebra/
        └test/
          └A/ n02381460_1000.jpg, ...
          └B/ n02391049_1000.jpg, ...
        └train/
          └A/ n02381460_1001.jpg, ...
          └B/ n02391049_10007.jpg, ...
    └output/ fake_A.png, fake_B.png, loss_D.png, ...
      └ABA/ A_0001.png, AB_0001.png, ABA_0001.png, ...
      └BAB/ B_0001.png, BA_0001.png, BAB_0001.png, ...
    └datasets.py
    └models.py # Residual block, Generator, Discriminator
    └test.py
    └train.py
    └utils.py # Logger, ReplayBuffer, LambdaLR, weight initializer
```

Part I: Image Generation using GAN (DCGAN)

How to run

To plot loss graphs and draw images in a web browser view:

```
whshin@ai2:/st1/whshin/workspace/cyclegan$ pip3 install visdom
whshin@ai2:/st1/whshin/workspace/cyclegan$ python -m visdom.server
```



Visdom

https://github.com/facebookresearch/visdom

A flexible tool for creating, organizing, and sharing visualizations of live, rich data.

Supports Torch and Numpy.

You can run the program with this command:

```
whshin@ai2:/st1/whshin/workspace/cyclegan$ python train.py --dataroot datasets/horse2zebra --outf my_output --cuda
```

Part II: Image Translation using GAN (CycleGAN)

Let's focus primarily on the **train.py**,

and check the other modules whenever they are actually called from the this code.

```
import modules
parser.add_argument()

# Networks (from model.py)
netG_A2B, netG_B2A, netD_A, netD_B

# Lossess
criterion_GAN, criterion_cycle, criterion_identity

# Optimizers & Dataset loader
optimizer_G, optimizer_D_A, optimizer_D_B

# LR schedulers & replay buffer (from utils.py)
lr_scheduler_G, lr_scheduler_D_A, lr_scheduler_D_B
fake_A_buffer, fake_B_buffer

# Dataset loader
dataloader

# Logger (from utils.py)
logger
```

```
for epoch in range(opt.epoch, opt.n_epochs):
  for i, batch in enumerate(dataloader):
    ###### Generators A2B and B2A ######
    # 1. Identity loss
    # 2. GAN loss
    # 3. Cycle loss
    # 4. Total loss

    ###### Discriminator A #####
    # 1. Real loss
    # 2. Fake loss
    # 3. Total loss

    ###### Discriminator A #####
    # 1. Real loss
    # 2. Fake loss
    # 3. Total loss
  # Update learning rate
  # Save models checkpoints
```

Part II: Image Translation using GAN (CycleGAN)

Module Import(train.py)

```python
import argparse
import itertools

import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torch.autograd import Variable
from PIL import Image
import torch

from models import Generator
from models import Discriminator
from utils import ReplayBuffer
from utils import LambdaLR
from utils import Logger
from utils import weights_init_normal
from datasets import ImageDataset
```

Part II: Image Translation using GAN (CycleGAN)

Parsers(train.py) – 11 arguments

```
parser = argparse.ArgumentParser()
parser.add_argument('--epoch', type=int, default=0, help='starting epoch')
parser.add_argument('--n_epochs', type=int, default=200, help='number of epochs of training')
parser.add_argument('--batchSize', type=int, default=1, help='size of the batches')
parser.add_argument('--dataroot', type=str, default='horse2zebra/’,
                    help='root directory of the dataset')
parser.add_argument('--lr', type=float, default=0.0002, help='initial learning rate')
parser.add_argument('--decay_epoch', type=int, default=100,
                   help='epoch to start linearly decaying the learning rate to 0')
parser.add_argument('--size', type=int, default=256, help='size of the data crop (squared assumed)')
parser.add_argument('--input_nc', type=int, default=3, help='number of channels of input data')
parser.add_argument('--output_nc', type=int, default=3, help='number of channels of output data')
parser.add_argument('--cuda', action='store_true', help='use GPU computation')
parser.add_argument('--n_cpu', type=int, default=8,
                   help='number of cpu threads to use during batch generation’)
parser.add_argument('--outf', default='output', help='folder to output images and model checkpoints')
```

Part II: Image Translation using GAN (CycleGAN)

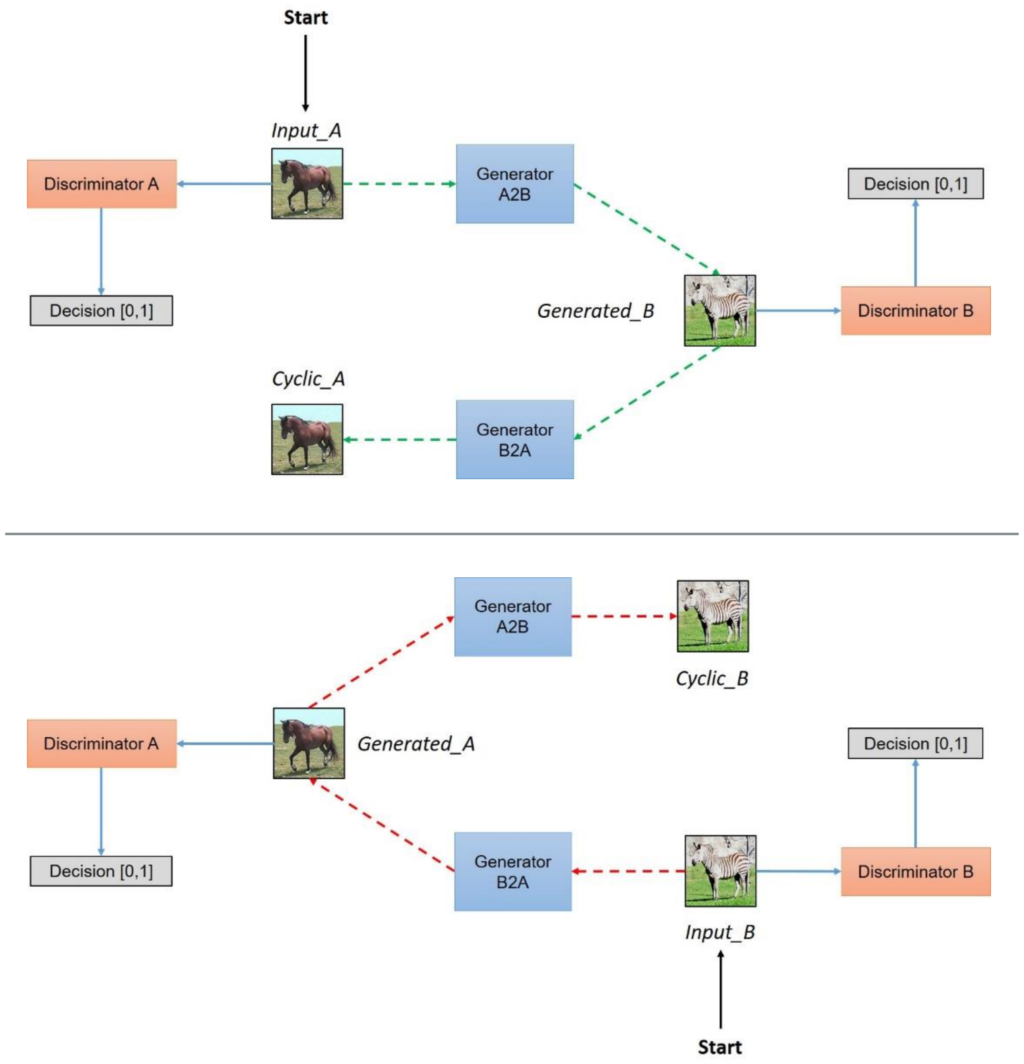
Networks (train.py)

```
# Networks
netG_A2B = Generator(opt.input_nc, opt.output_nc)
netG_B2A = Generator(opt.output_nc, opt.input_nc)
netD_A = Discriminator(opt.input_nc)
netD_B = Discriminator(opt.output_nc)

if opt.cuda:
  netG_A2B.cuda()
  netG_B2A.cuda()
  netD_A.cuda()
  netD_B.cuda()

netG_A2B.apply(weights_init_normal)
netG_B2A.apply(weights_init_normal)
netD_A.apply(weights_init_normal)
netD_B.apply(weights_init_normal)
```

weigths_init_normal() is the same function as the one we used for **DCGAN**.

Part II: Image Translation using GAN (CycleGAN)

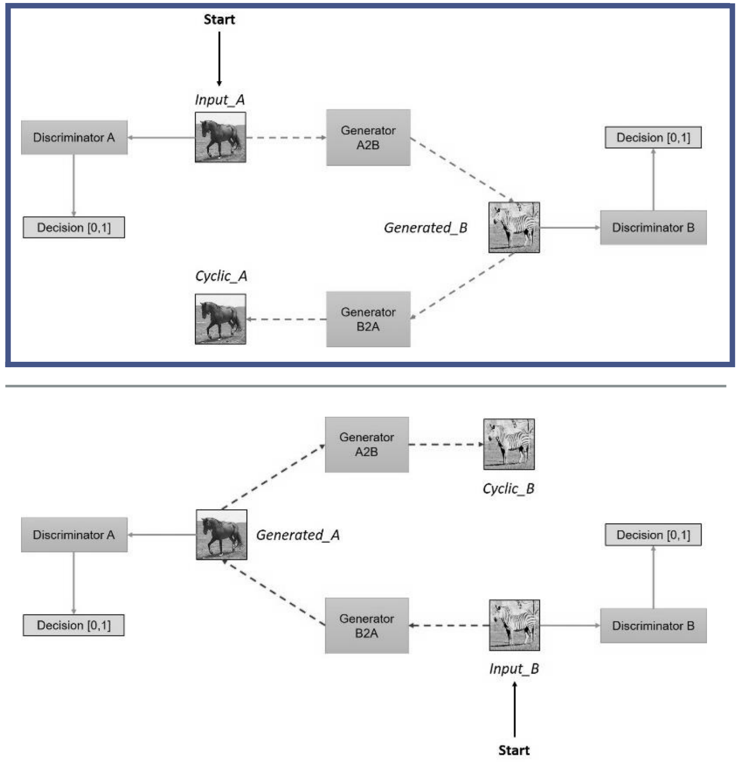
Networks Overview

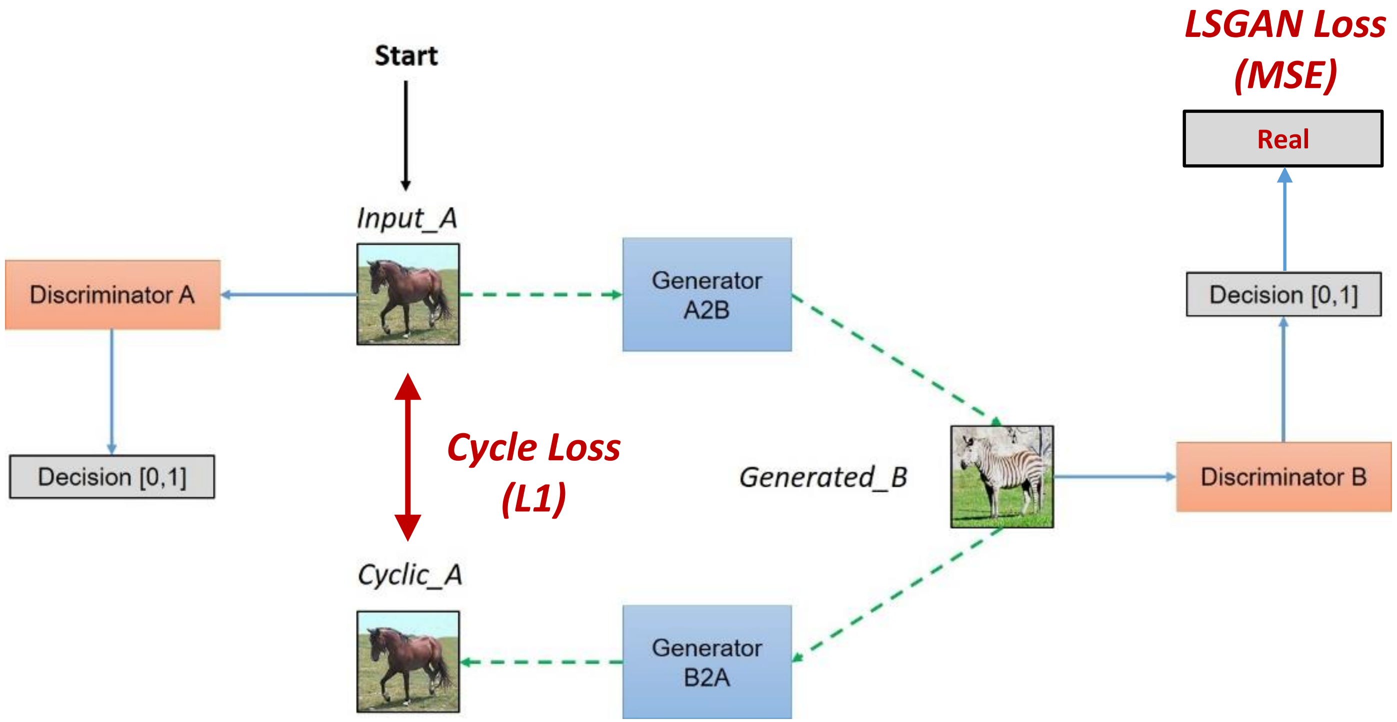


Part II: Image Translation using GAN (CycleGAN)

Networks Overview

: Generators(**A2B** & **B2A**) training

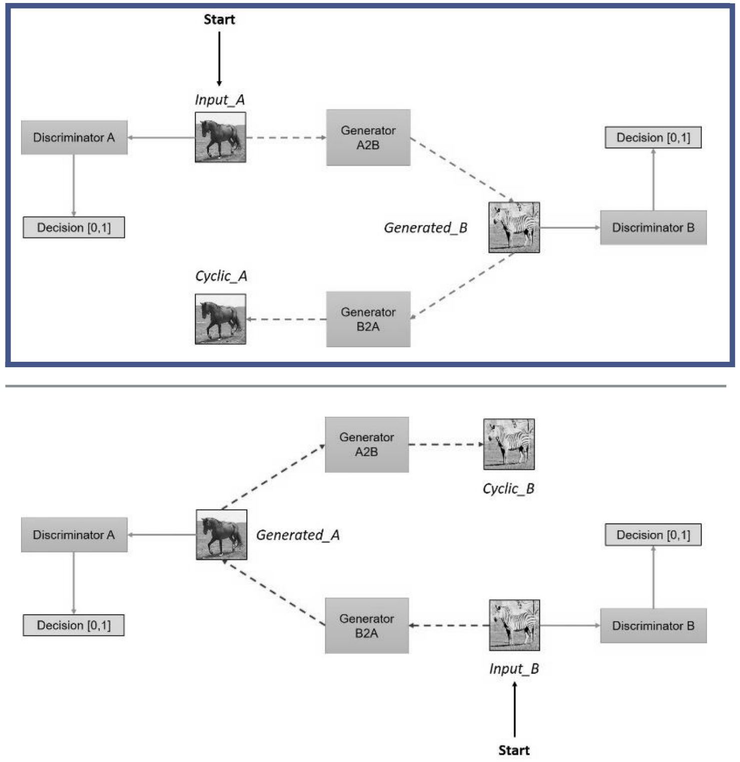


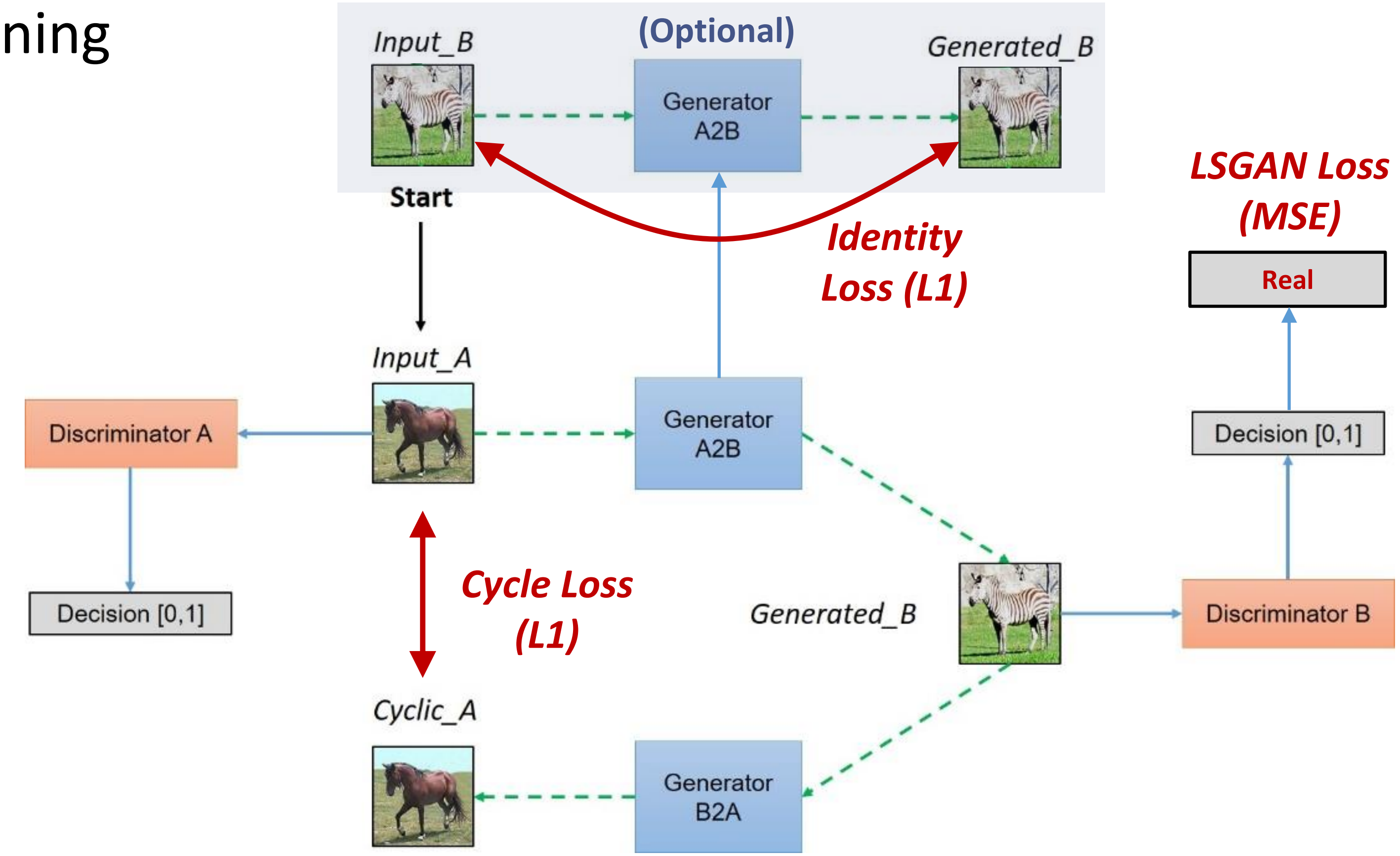


Part II: Image Translation using GAN (CycleGAN)

Networks Overview

: Generators(**A2B** & **B2A**) training

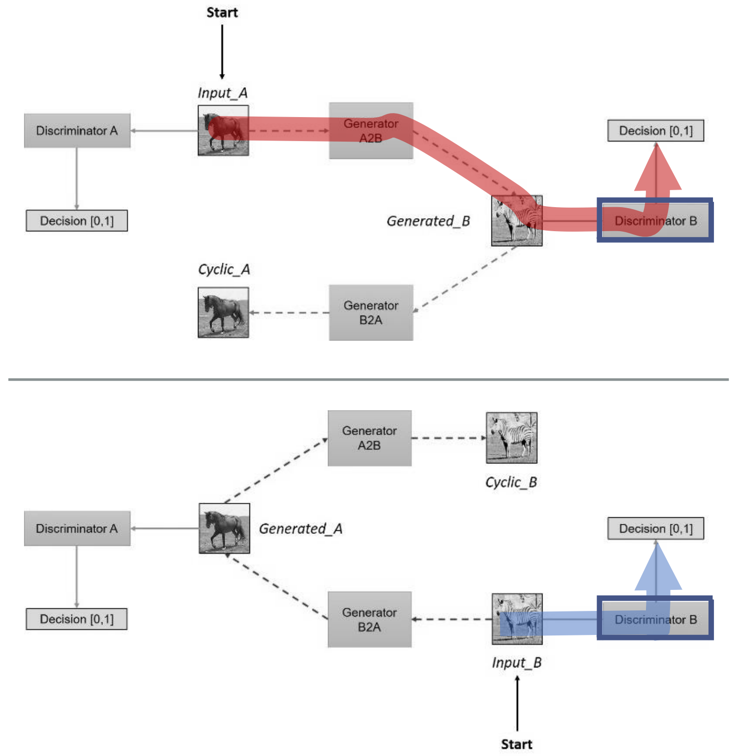


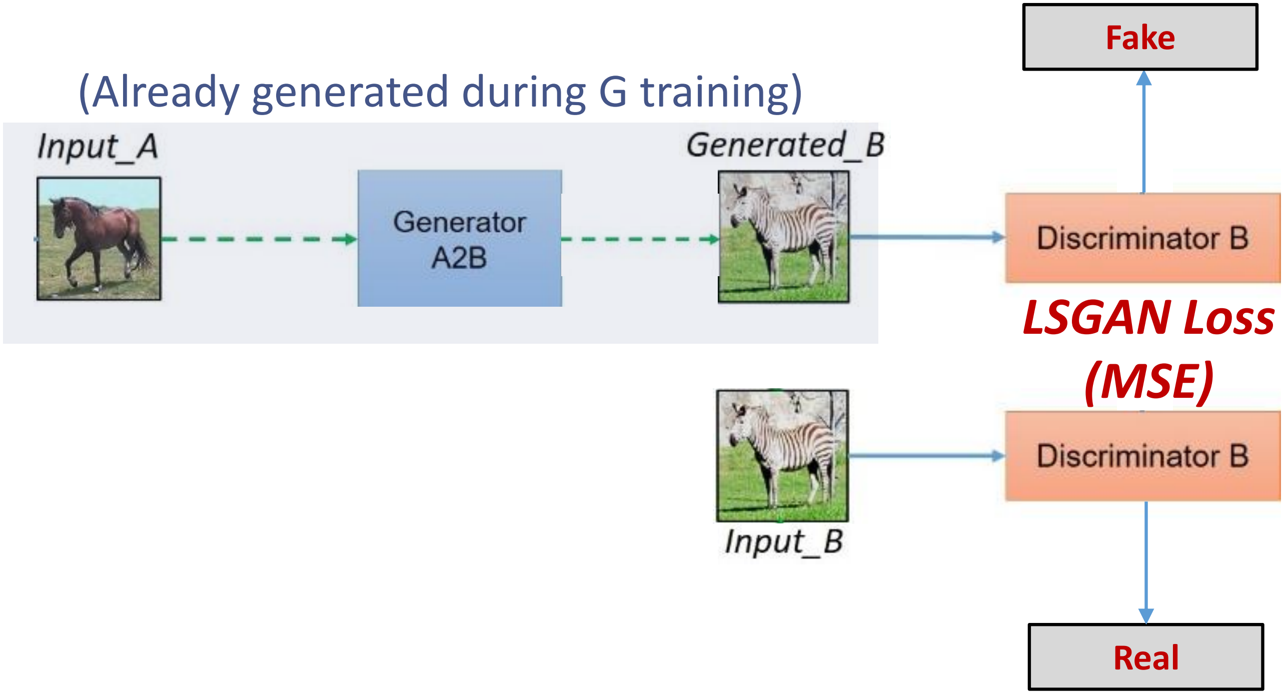


Part II: Image Translation using GAN (CycleGAN)

Networks Overview

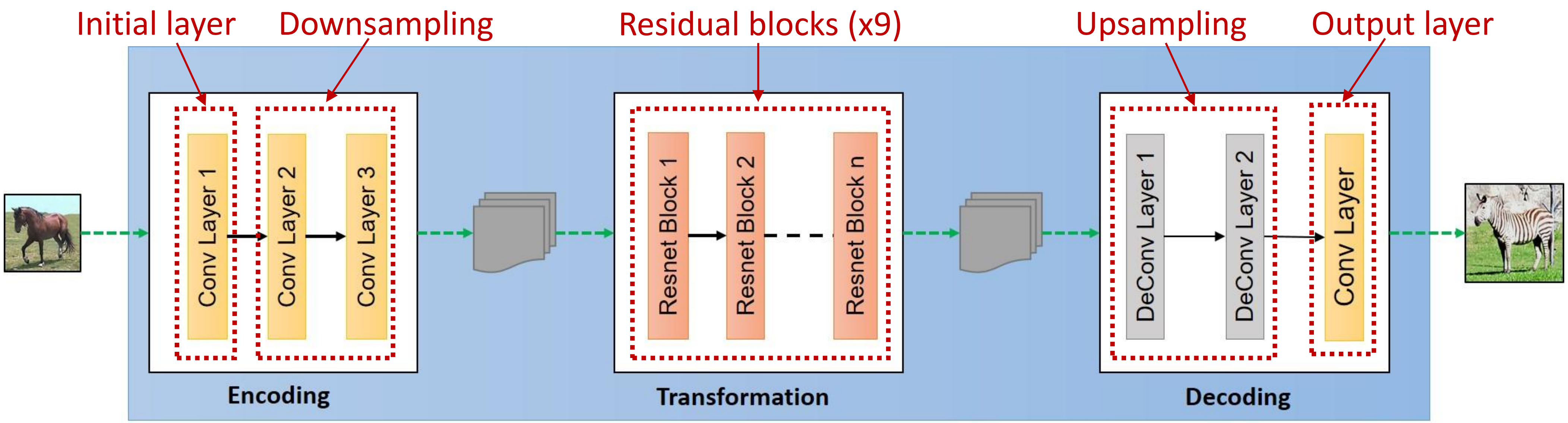
: Discriminator(**B**) training

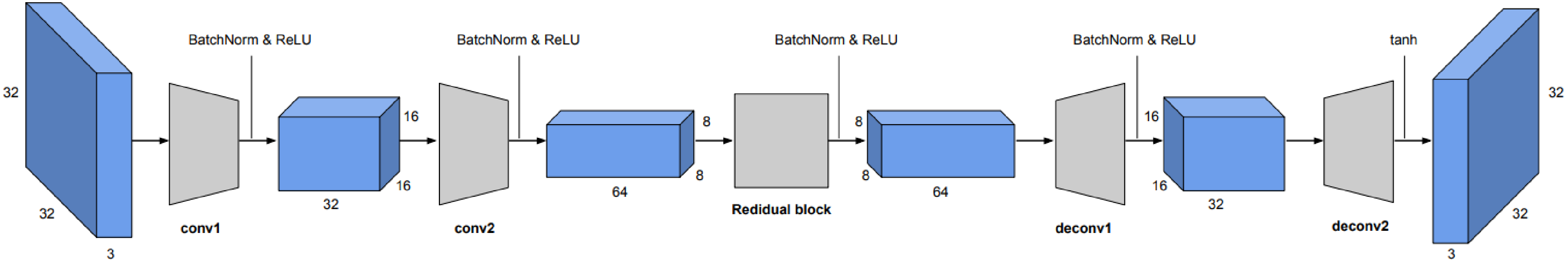




Part II: Image Translation using GAN (CycleGAN)

Networks - Generator





Part II: Image Translation using GAN (CycleGAN)

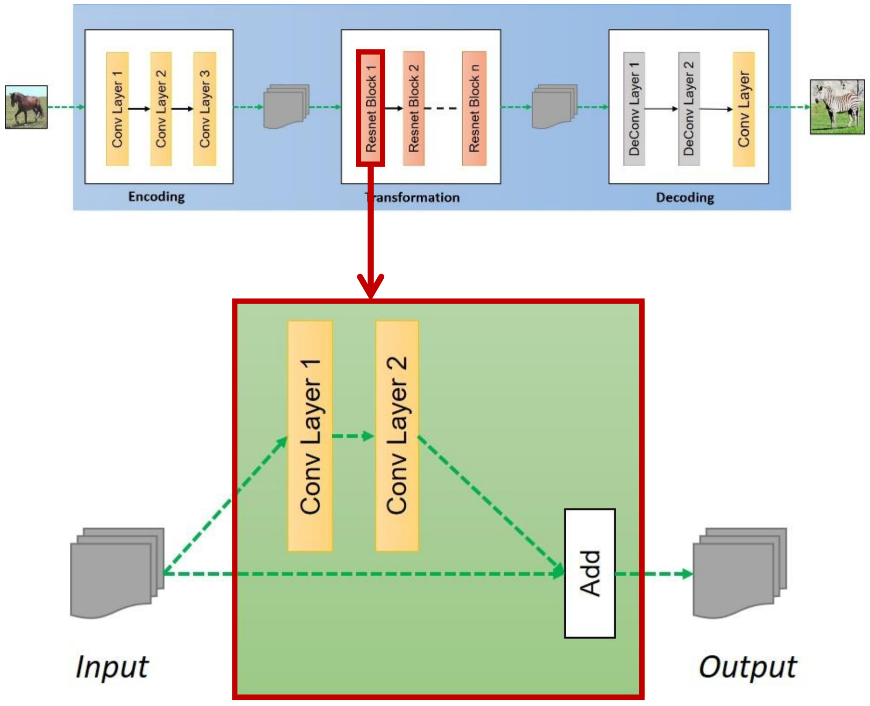
Networks (model.py) - Generator

```
class Generator(nn.Module):
  def __init__(
    self, input_nc, output_nc, n_residual_blocks=9):
    super(Generator, self).__init__()

    # Initial convolution block
    model = [ nn.ReflectionPad2d(3),
              nn.Conv2d(input_nc, 64, 7),
              nn.InstanceNorm2d(64),
              nn.ReLU(inplace=True) ]

    # Downsampling
    in_features = 64
    out_features = in_features*2

    for _ in range(2):
      model += [ nn.Conv2d(in_features, out_features,
                           kernel_size=3, stride=2,
                           padding=1),
                 nn.InstanceNorm2d(out_features),
                 nn.ReLU(inplace=True) ]

    in_features = out_features
    out_features = in_features*2

    # Residual blocks
    for _ in range(n_residual_blocks):
      model += [ResidualBlock(in_features)]

    # Upsampling
    out_features = in_features//2
    for _ in range(2):
      model += [ nn.ConvTranspose2d(in_features, out_features,
                                    kernel_size=3, stride=2,
                                    padding=1, output_padding=1),
                 nn.InstanceNorm2d(out_features),
                 nn.ReLU(inplace=True) ]
    in_features = out_features
    out_features = in_features//2

    # Output layer
    model += [ nn.ReflectionPad2d(3),
               nn.Conv2d(64, output_nc, 7),
               nn.Tanh() ]

    self.model = nn.Sequential(*model)

  def forward(self, x):
    return self.model(x)
```

Part II: Image Translation using GAN (CycleGAN)

Networks (model.py) - Generator (Residual Block)

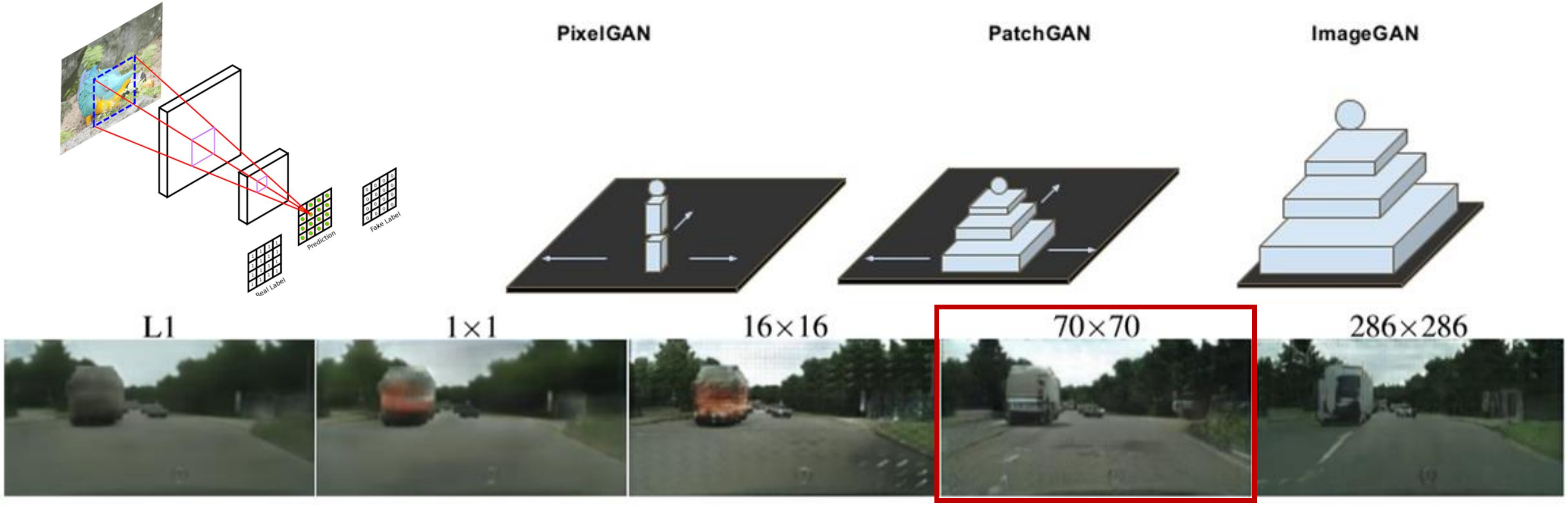
```python
class ResidualBlock(nn.Module):
  def __init__(self, in_features):
    super(ResidualBlock, self).__init__()

    conv_block = [ nn.ReflectionPad2d(1),
                   nn.Conv2d(in_features, in_features,
                             kernel_size=3),
                   nn.InstanceNorm2d(in_features),
                   nn.ReLU(inplace=True),
                   nn.ReflectionPad2d(1),
                   nn.Conv2d(in_features, in_features,
                             kernel_size=3),
                   nn.InstanceNorm2d(in_features) ]

    self.conv_block = nn.Sequential(*conv_block)

  def forward(self, x):
    return x + self.conv_block(x)
```



Resnet Block

Part II: Image Translation using GAN (CycleGAN)

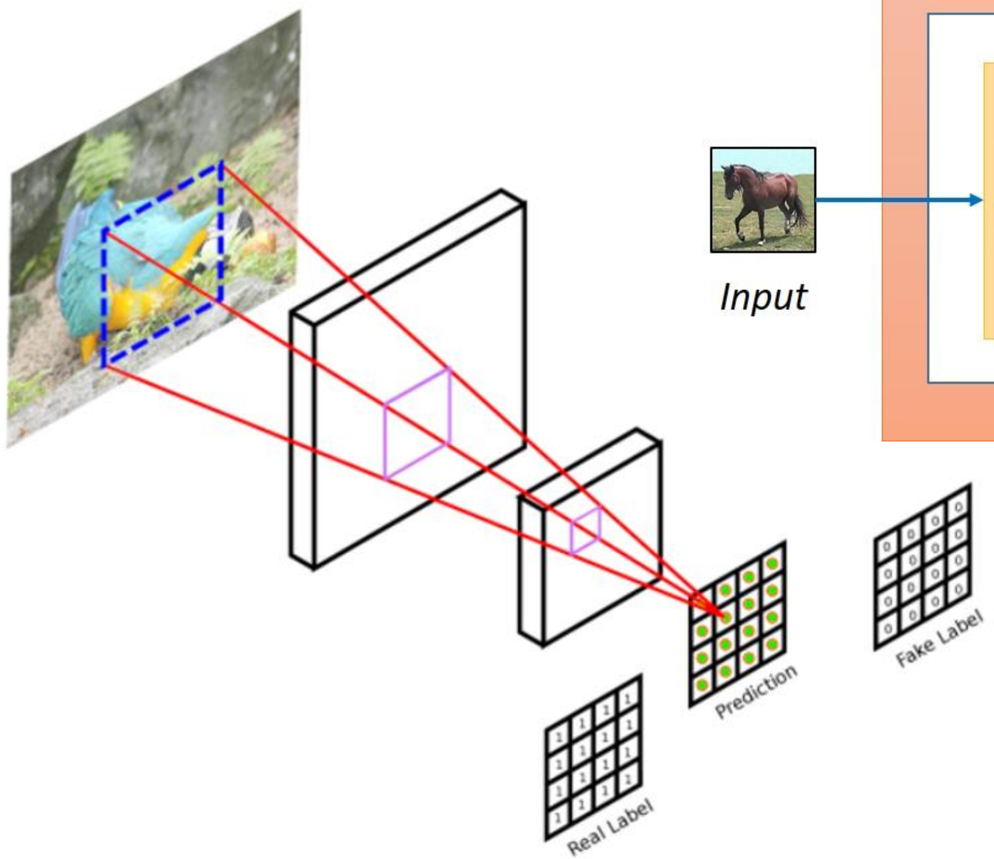
Networks - Discriminator (from PatchGAN)

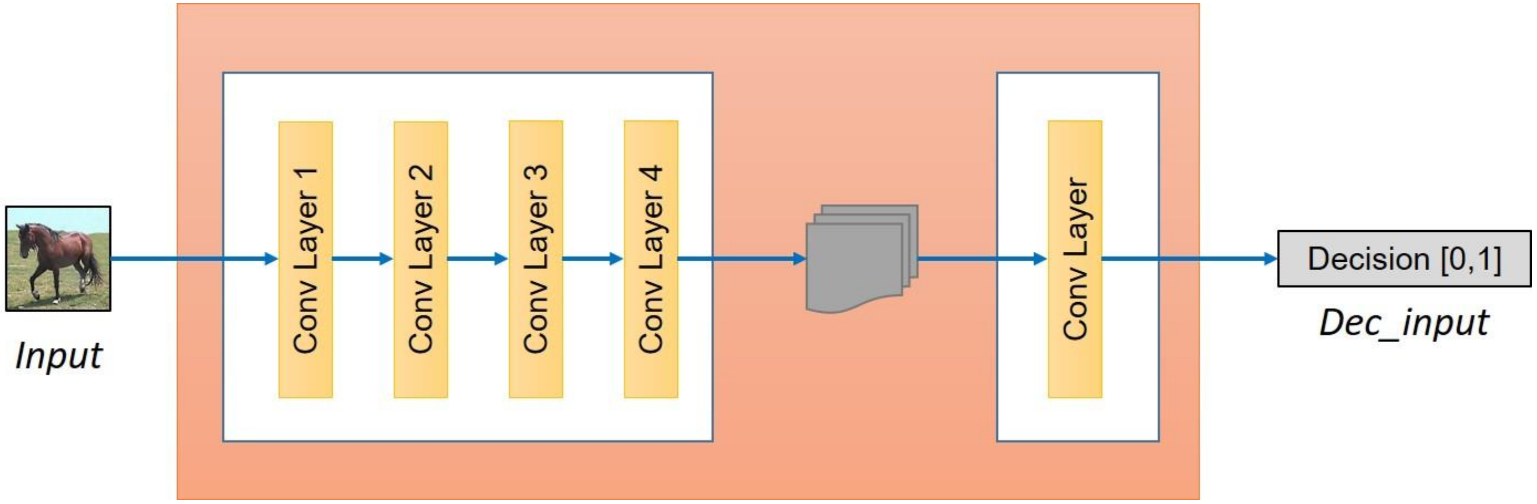


Correlation between the pixels decreases proportional to the distance.
PatchGAN discriminate the real and the fake in overlapping patches of a certain size rather than considering the whole image.

Part II: Image Translation using GAN (CycleGAN)

Networks - Discriminator (from PatchGAN)





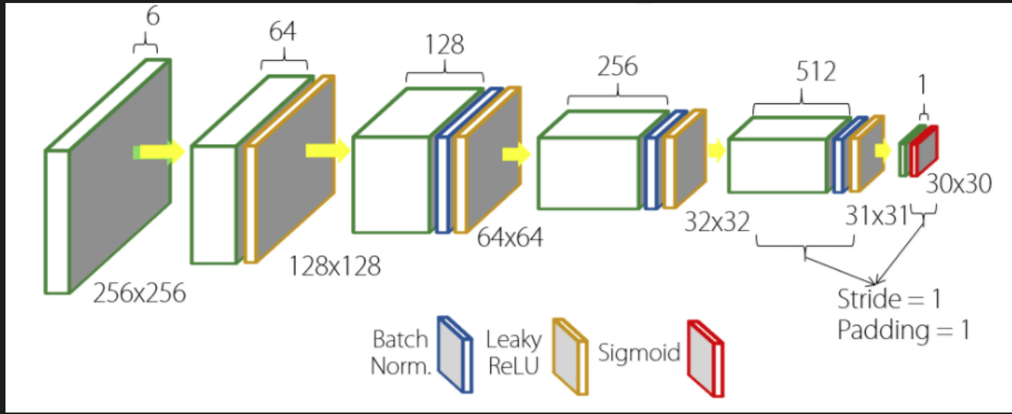
Patch-level discriminator has..

1. Less number of parameters.
2. Smaller computational cost.
3. Flexible input size.

Part II: Image Translation using GAN (CycleGAN)

Networks (model.py) - Discriminator (from PatchGAN)

```
class Discriminator(nn.Module):
  def __init__(self, input_nc):
    super(Discriminator, self).__init__()

    # A bunch of convolutions one after another
    model = [ nn.Conv2d(input_nc, 64, 4,
                        stride=2, padding=1),
              nn.LeakyReLU(0.2, inplace=True) ]

    model += [ nn.Conv2d(64, 128, 4,
                         stride=2, padding=1),
               nn.InstanceNorm2d(128),
               nn.LeakyReLU(0.2, inplace=True) ]

    model += [ nn.Conv2d(128, 256, 4, stride=2, padding=1),
               nn.InstanceNorm2d(256),
               nn.LeakyReLU(0.2, inplace=True) ]

    model += [ nn.Conv2d(256, 512, 4, padding=1),
               nn.InstanceNorm2d(512),
               nn.LeakyReLU(0.2, inplace=True) ]

    # FCN classification layer
    model += [nn.Conv2d(512, 1, 4, padding=1)]

    self.model = nn.Sequential(*model)

  def forward(self, x):
    x = self.model(x)
    # Average pooling and flatten
    return F.avg_pool2d(x, x.size()[2:]).view(x.size()[0], -1)
```



(1) **LSGAN** does NOT need ***sigmoid function*** at the last layer.

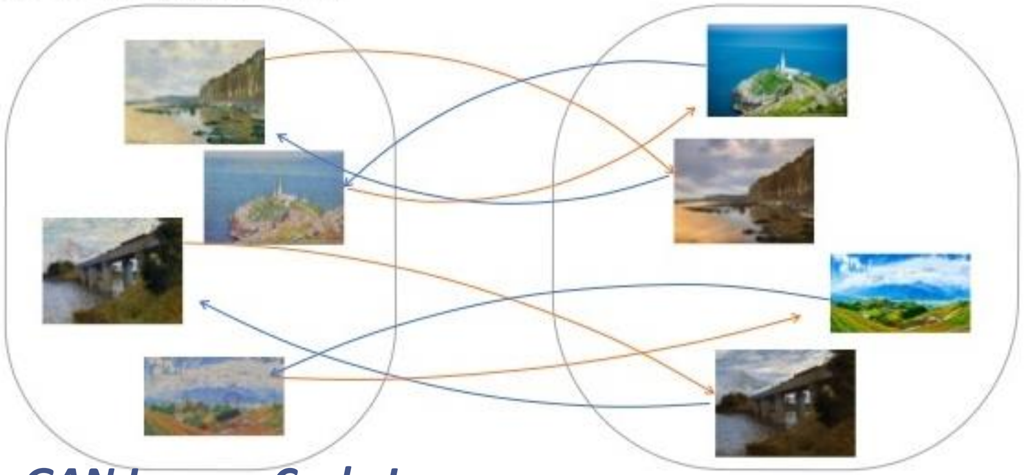
(2) The author applies ***average pooling*** resulting in 1-D label for each instance.

Part II: Image Translation using GAN (CycleGAN)

Losses (train.py)

```
# Lossess
criterion_GAN = torch.nn.MSELoss() # We will use LSGAN loss
criterion_cycle = torch.nn.L1Loss()
criterion_identity = torch.nn.L1Loss()
```

CycleGAN (Zhu et al. 2017)



GAN Loss *Cycle Loss*

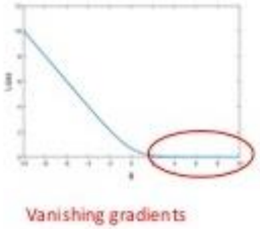
$$L_{GAN}(G(x), y) + \|F(G(x)) - x\|_1 \quad + \quad L_{GAN}(F(y), x) + \|G(F(y)) - y\|_1$$

+ ***Identity Loss (Optional):*** $\|G(y) - y\|_1$

Training Details: Objective

- GANs with cross-entropy loss

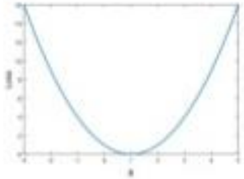
$$\mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{\text{data}}(y)}[\log D_Y(y)] + \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log(1 - D_Y(G(x)))],$$



- Least square GANs [Mao et al. 2016]

  Stable training + better results

$$\mathcal{L}_{\text{LSGAN}}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{\text{data}}(y)}[(D_Y(y) - 1)^2] + \mathbb{E}_{x \sim p_{\text{data}}(x)}[D_Y(G(x))^2],$$



Part II: Image Translation using GAN (CycleGAN)

Optimizers (train.py)

```
# Optimizers
optimizer_G = torch.optim.Adam(itertools.chain(netG_A2B.parameters(), netG_B2A.parameters()),
                               lr=opt.lr, betas=(0.5, 0.999))
optimizer_D_A = torch.optim.Adam(netD_A.parameters(), lr=opt.lr, betas=(0.5, 0.999))
optimizer_D_B = torch.optim.Adam(netD_B.parameters(), lr=opt.lr, betas=(0.5, 0.999))

# LR schedulers
lr_scheduler_G = torch.optim.lr_scheduler.LambdaLR(
  optimizer_G, lr_lambda=LambdaLR(opt.n_epochs, opt.epoch, opt.decay_epoch).step)
lr_scheduler_D_A = torch.optim.lr_scheduler.LambdaLR(
  optimizer_D_A, lr_lambda=LambdaLR(opt.n_epochs, opt.epoch, opt.decay_epoch).step)
lr_scheduler_D_B = torch.optim.lr_scheduler.LambdaLR(
  optimizer_D_B, lr_lambda=LambdaLR(opt.n_epochs, opt.epoch, opt.decay_epoch).step)
```

CLASS `torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda, last_epoch=-1)`

- **optimizer** (Optimizer) – Wrapped optimizer.
- **lr_lambda** (function or list) – A function which computes a multiplicative factor given an integer parameter epoch, or a list of such functions, one for each group in optimizer.param_groups. E.g. lr_lambda = **lambda** epoch: 0.95 ** epoch
- **last_epoch** (int) – The index of last epoch. Default: -1.

Part II: Image Translation using GAN (CycleGAN)

Optimizers - Customized Scheduler (utils.py)

```
# LR schedulers (train.py)
lr_scheduler_G = torch.optim.lr_scheduler.LambdaLR(
  optimizer_G, lr_lambda=LambdaLR(opt.n_epochs, opt.epoch, opt.decay_epoch).step)

# LR Lambda (utils.py)
class LambdaLR():
  def __init__(self, n_epochs, offset, decay_start_epoch):
    assert ((n_epochs - decay_start_epoch) > 0), ""
      "Decay must start before the training session ends!"
    self.n_epochs = n_epochs
    self.offset = offset
    self.decay_start_epoch = decay_start_epoch

  def step(self, epoch):
    return 1.0 - max(0, epoch + self.offset - self.decay_start_epoch) \
      / (self.n_epochs - self.decay_start_epoch)
```

*It decays the initial learning rate linearly to zero through the entire epochs.

Part II: Image Translation using GAN (CycleGAN)

Replay Buffer(train.py and utils.py): Update D using a history of generated images.

```
# train.py
fake_A_buffer = ReplayBuffer()
fake_B_buffer = ReplayBuffer()

# utils.py
class ReplayBuffer():
  def __init__(self, max_size=50):
    assert (max_size > 0), 'Empty buffer '
      'or trying to create a black hole. ‘
      'Be careful.’
    self.max_size = max_size
    self.data = []
  def push_and_pop(self, data):
    to_return = []
```

```
    for element in data.data:
      element = torch.unsqueeze(element, 0)
      if len(self.data) < self.max_size:
        self.data.append(element)
        to_return.append(element)
      else:
        if random.uniform(0,1) > 0.5:
          i = random.randint(0, self.max_size-1)
          to_return.append(self.data[i].clone())
          self.data[i] = element
        else:
          to_return.append(element)
    return Variable(torch.cat(to_return))
```

If the buffer is **NOT** full; keep inserting current images to the buffer.

If the buffer is full; **(1)** By **50%** chance, the buffer will return a **previously stored image**, and insert the current image into the buffer.

(2) By another **50%** chance, the buffer will return **current image.**

Part II: Image Translation using GAN (CycleGAN)

Dataloader(train.py)

```
# Dataset loader (train.py)
transforms_ = [ transforms.Resize(int(opt.size*1.12), Image.BICUBIC),
                transforms.RandomCrop(opt.size),
                transforms.RandomHorizontalFlip(),
                transforms.ToTensor(),
                transforms.Normalize((0.5,0.5,0.5), (0.5,0.5,0.5)) ]

dataset = ImageDataset(opt.dataroot, transforms_=transforms_, unaligned=True)
dataloader = DataLoader(dataset, batch_size=opt.batchSize, shuffle=True, num_workers=opt.n_cpu)
```

Part II: Image Translation using GAN (CycleGAN)

Dataloader(datasets.py) - Customized Dataset

```python
# Dataset loader (datasets.py)
import glob
import random
import os

from torch.utils.data import Dataset
from PIL import Image
import torchvision.transforms as transforms

class ImageDataset(Dataset):
  def __init__(self, root, transforms_=None,
    unaligned=False, mode='train’):
    self.transform = transforms.Compose(transforms_)
    self.unaligned = unaligned

    self.files_A = sorted(glob.glob(os.path.join(
      root, '%s/A' % mode) + '/*.*’))
    self.files_B = sorted(glob.glob(os.path.join(
      root, '%s/B' % mode) + '/*.*'))
```

```python
def __getitem__(self, index):
  item_A = self.transform(Image.open(
    self.files_A[index % len(self.files_A)]))

  if self.unaligned:
    item_B = self.transform(Image.open(
      self.files_B[random.randint(
        0, len(self.files_B) - 1)]))
  else:
    item_B = self.transform(Image.open(
      self.files_B[index % len(self.files_B)]))

  return {'A': item_A, 'B': item_B}

def __len__(self):
  return max(len(self.files_A), len(self.files_B))
```

Part II: Image Translation using GAN (CycleGAN)

Training (train.py)

```
for epoch in range(opt.epoch, opt.n_epochs):
  for i, batch in enumerate(dataloader):
    # Set model input

    ###### Generators A2B and B2A ######
    # Identity loss
    # GAN loss
    # Cycle loss

    ###### Discriminator A ######
    # Real loss
    # Fake loss

    ###### Discriminator B ######
    # Real loss
    # Fake loss

    # logger.log

  # Update learning rate
  # Save models checkpoints
```

CycleGAN (Zhu et al. 2017)

GAN Loss ***Cycle Loss***

$$L_{GAN}(G(x),y)+\|F(G(x))-x\|_1 \quad + \quad L_{GAN}(F(y),x)+\|G(F(y))-y\|_1$$

+ ***Identity Loss:*** **BtoA**(A) = A

Part II: Image Translation using GAN (CycleGAN)

Training (train.py) – Generators A2B & B2A

```
###### Generators A2B and B2A ######
optimizer_G.zero_grad()
loss_G = 0.0

###### 1. Identity loss (L1 Loss) ######
# G_A2B(B) should equal B
same_B = netG_A2B(real_B)
loss_G += criterion_identity(same_B, real_B) * 5.0
# G_B2A(A) should equal A
same_A = netG_B2A(real_A)
loss_G += criterion_identity(same_A, real_A) * 5.0

###### 2. LSGAN loss (MSE Loss) ######
# G_A2B(A) should fool D_B
fake_B = netG_A2B(real_A)
pred_fake = netD_B(fake_B)
loss_G += criterion_GAN(pred_fake, target_real)
```

```
# G_B2A(B) should fool D_A
fake_A = netG_B2A(real_B)
pred_fake = netD_A(fake_A)
loss_G += criterion_GAN(pred_fake, target_real)

###### 3. Cycle loss (L1 Loss) ######
# G_B2A(G_A2B(A)) should equal A
recon_A = netG_B2A(fake_B)
loss_G += criterion_cycle(recon_A, real_A) * 10.0
# G_A2B(G_B2A(B)) should equal B
recon_B = netG_A2B(fake_A)
loss_G = criterion_cycle(recon_B, real_B) * 10.0

###### Update both Gs ######
loss_G.backward()
optimizer_G.step()
```

Part II: Image Translation using GAN (CycleGAN)

Training (train.py) – Discriminator A & B

```
###### Discriminator A ######
optimizer_D_A.zero_grad()

# Real loss (MSE Loss)
pred_real = netD_A(real_A)
loss_D_real = criterion_GAN(pred_real, target_real)

# Fake loss (MSE Loss)
fake_A = fake_A_buffer.push_and_pop(fake_A)
pred_fake = netD_A(fake_A.detach())
loss_D_fake = criterion_GAN(pred_fake, target_fake)

# Total loss
loss_D_A = (loss_D_real + loss_D_fake) * 0.5
loss_D_A.backward()

optimizer_D_A.step()
```

```
###### Discriminator B ######
optimizer_D_B.zero_grad()

# Real loss (MSE Loss)
pred_real = netD_B(real_B)
loss_D_real = criterion_GAN(pred_real, target_real)

# Fake loss (MSE Loss)
fake_B = fake_B_buffer.push_and_pop(fake_B)
pred_fake = netD_B(fake_B.detach())
loss_D_fake = criterion_GAN(pred_fake, target_fake)

# Total loss
loss_D_B = (loss_D_real + loss_D_fake) * 0.5
loss_D_B.backward()

optimizer_D_B.step()
```

Part II: Image Translation using GAN (CycleGAN)

Update learning rate & save models checkpoints

```
# Update learning rates
lr_scheduler_G.step()
lr_scheduler_D_A.step()
lr_scheduler_D_B.step()

# Save models checkpoints
torch.save(netG_A2B.state_dict(), '%s/netG_A2B.pth' % (opt.opt.outf))
torch.save(netG_B2A.state_dict(), '%s/netG_B2A.pth' % (opt.opt.outf))
torch.save(netD_A.state_dict(), '%s/netD_A.pth' % (opt.opt.outf))
torch.save(netD_B.state_dict(), '%s/netD_B.pth' % (opt.opt.outf))
```

Part II: Image Translation using GAN (CycleGAN)

Testing (test.py)

You can run the test code with this command:

```
whshin@ai2:/st1/whshin/workspace/cyclegan$ python test.py --dataroot datasets/horse2zebra --outf my_output --cuda
```

You can load the model parameters trained for about 30 epochs (--outf output), and result will be saved under the same path. (ABA, BAB)

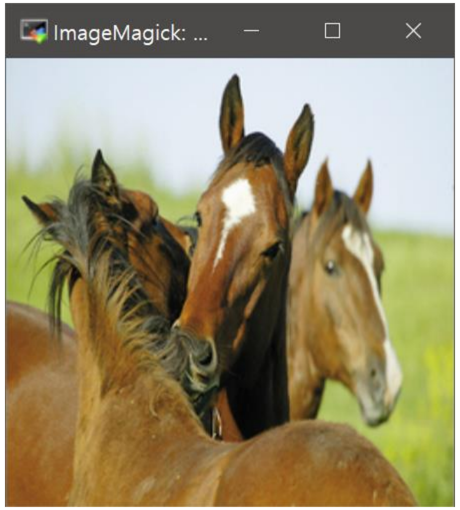
```
-tutorial_gan/
└cyclegan/
  └output/ netG_A2B.path, netG_B2A.pth, ... # Pretrained models
    # Test result will be saved under <outf>/ABA and <outf>/BAB
    └ABA/ A_0001.png, AB_0001.png, ABA_0001.png, ...
    └BAB/ B_0001.png, BA_0001.png, BAB_0001.png, ...
```

Part II: Image Translation using GAN (CycleGAN)

Testing (test.py)

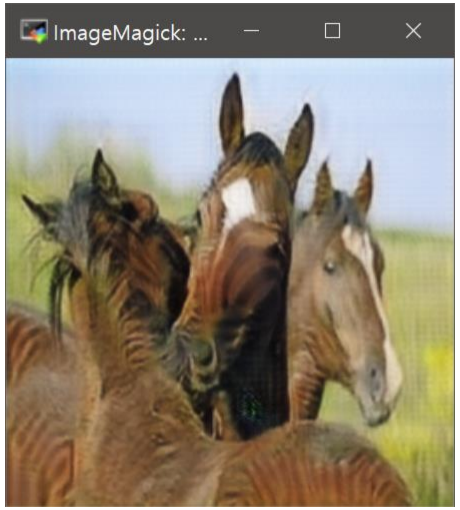
*Generator trained for about 30 epochs. (can be improved if trained longer)

./output/ABA



A_0005.png

AB_0005.png



ABA_0005.png

Part II: Image Translation using GAN (CycleGAN)

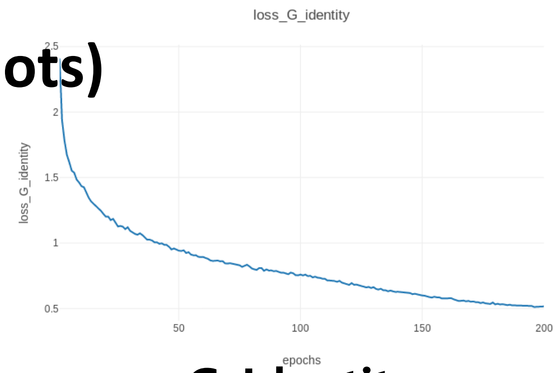
Results (Images)

A → G_AB(A)

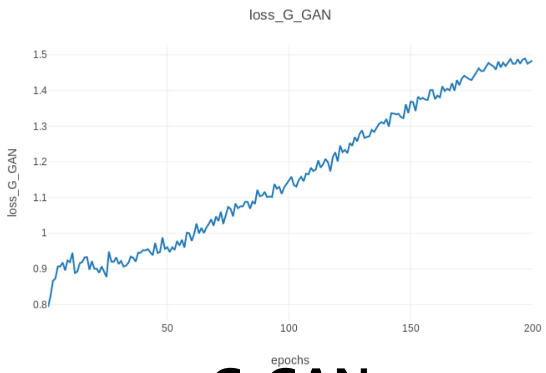
B → G_BA(B)

Part II: Image Translation using GAN (CycleGAN)

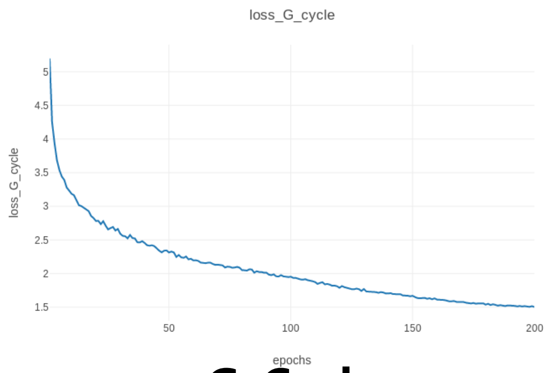
Results (Plots)



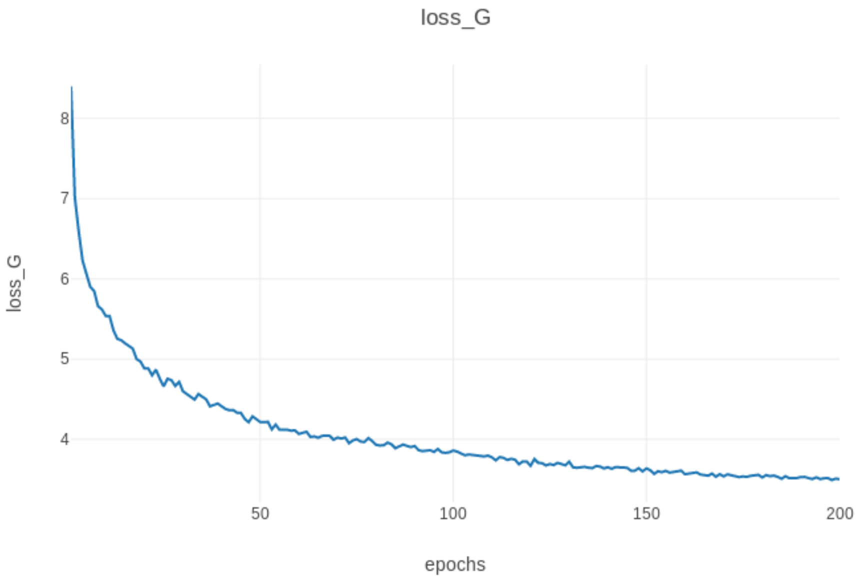
G_Identity



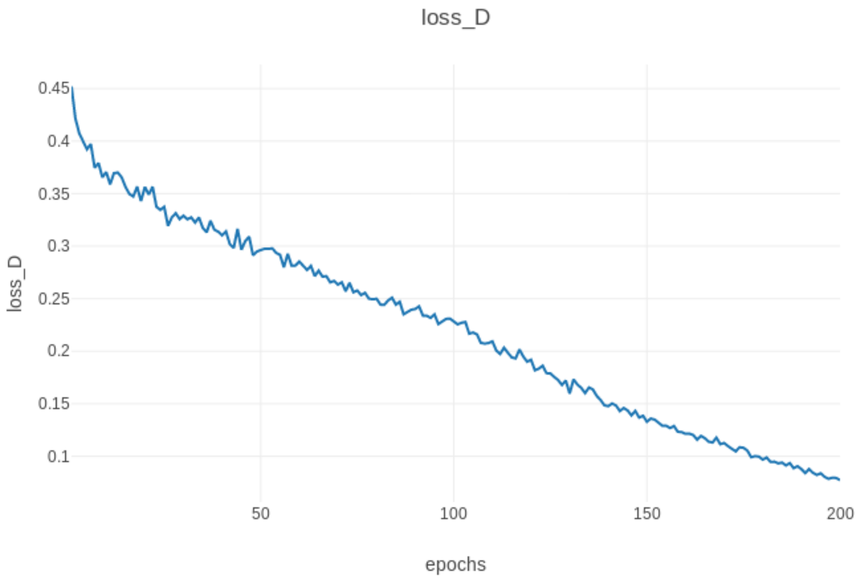
G_GAN



G_Cycle



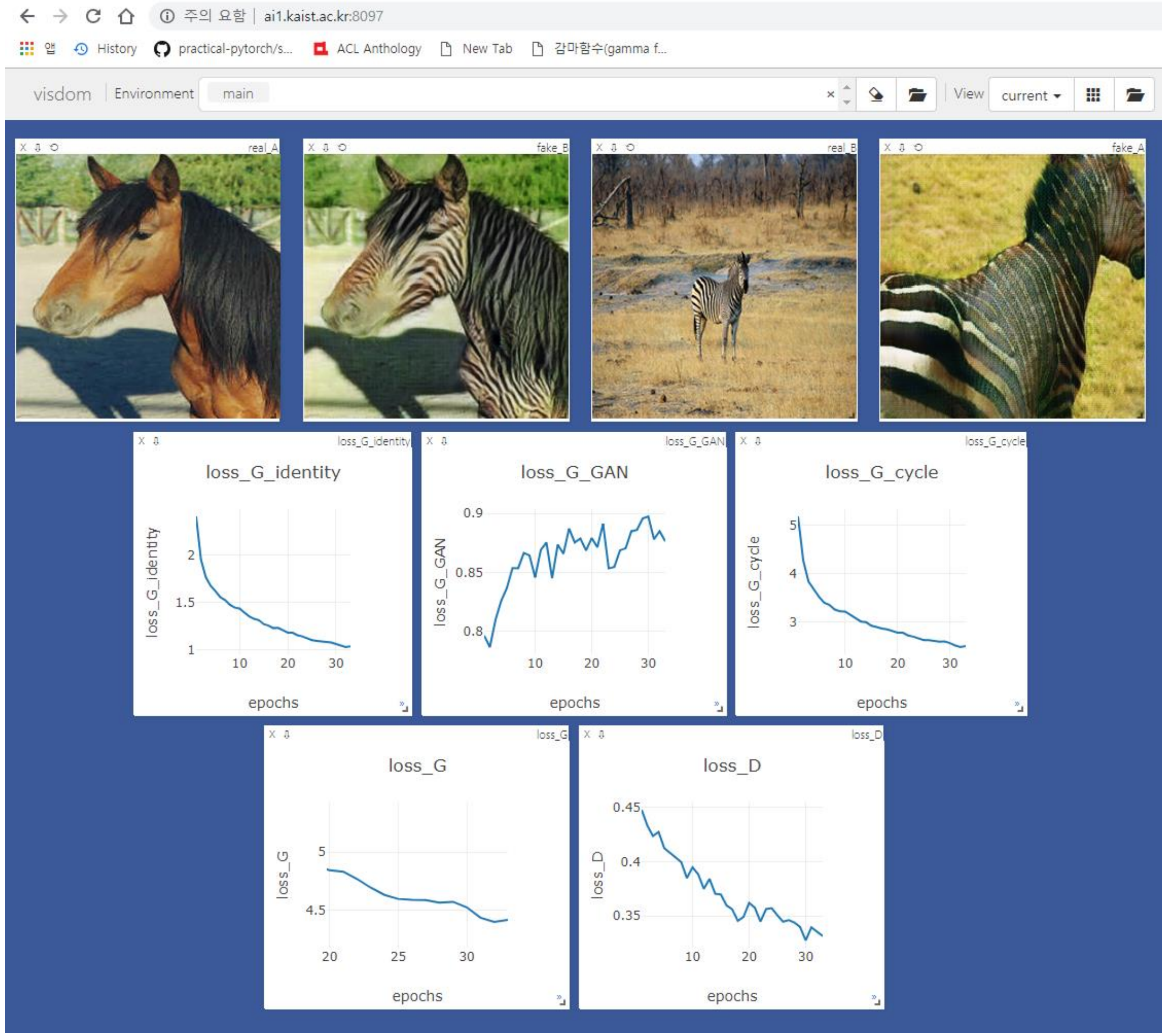
Generator



Discriminator

Part II: Image Translation using GAN (CycleGAN)

Results (Visdom)



Any questions?