Chapter 4
Generative Adversarial Network

4. 1 Generative Adversarial Network

Generative Adversarial Networks (GANs for short) have had a huge success since they were introduced in 2014 by Ian J. Goodfellow and co-authors in the article Generative Adversarial Nets.

Suppose that we are interested in generating black and white square images of dogs with a size of n by n pixels. We can reshape each data as a N=nxn dimensional vector (by stacking columns on top of each others) such that an image of dog can then be represented by a vector. However, it doesn't mean that all vectors represent a dog once shaped back to a square! So, we can say that the N dimensional vectors that effectively give something that look like a dog are distributed according to a very specific probability distribution over the entire N dimensional vector space (some points of that space are very likely to represent dogs whereas it is highly unlikely for some others). In the same spirit, there exists, over this N dimensional vector space, probability distributions for images of cats, birds and so on.

Then, the problem of generating a new image of dog is equivalent to the problem of generating a new vector following the "dog probability distribution" over the N dimensional vector space. So we are, in fact, facing a problem of generating a random variable with respect to a specific probability distribution.

At this point, we can mention two important things. First the "dog probability distribution" we mentioned is a very complex distribution over a very large space. Second, even if we can assume the existence of such underlying distribution (there actually exists images that looks like dog and others that doesn't) we obviously don't know how to express explicitly this distribution. Both previous points make the process of generating random variables from this distribution pretty difficult. Let's then try to tackle these two problems in the following.

Our first problem when trying to generate our new image of dog is that the "dog probability distribution" over the N dimensional vector space is a very complex one and we don't know how to directly generate complex random variables. However, as we know pretty well how to generate N uncorrelated uniform random variables, we could make use of the transform method. To do so, we need to express our N dimensional random variable as the result of a very complex function applied to a simple N dimensional random variable!

Here, we can emphasise the fact that finding the transform function is not as straightforward as just taking the closed-form inverse of the Cumulative Distribution Function (that we obviously don't know) as we have done when describing the inverse transform method. The transform function can't be explicitly expressed and, then, we have to learn it from data.
As most of the time in these cases, very complex function naturally implies neural network modelling. Then, the idea is to model the transform function by a neural network that takes as input a simple N dimensional uniform random variable and that returns as output another N dimensional random variable that should follow, after training, the the right "dog probability distribution". Once the architecture of the network has been designed, we still need to train it. In the next two sections, we will discuss two ways to train these generative networks, including the idea of adversarial training behind GANs!

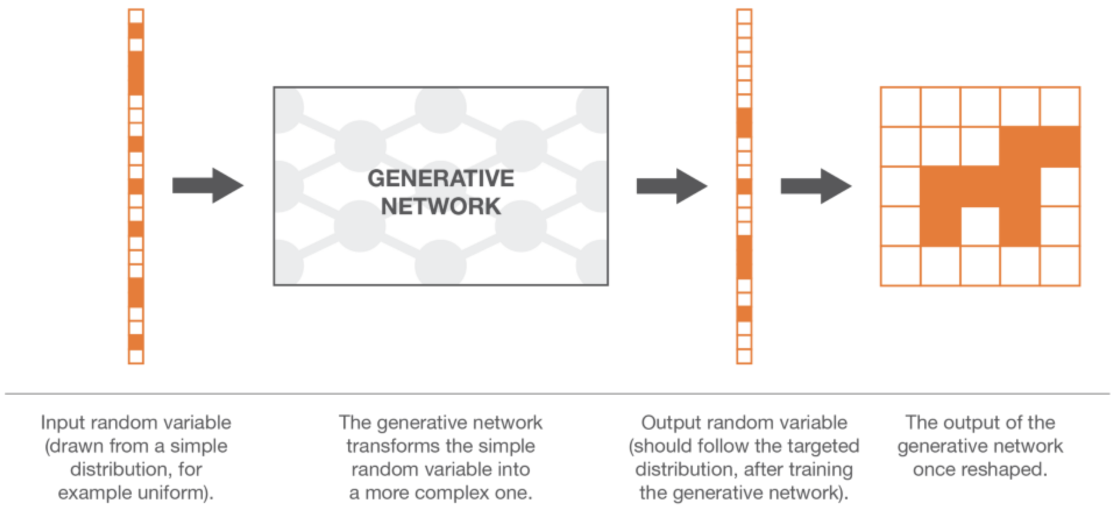
GENERATIVE NETWORK
Input random variable (drawn from a simple distribution, for example uniform).
The generative network transforms the simple random variable into a more complex one.
Output random variable (should follow the targeted distribution, after training the generative network).
The output of the generative network once reshaped.

Fig 4.1 GAN structure

<u>**4.2 Generative vs. Discriminative Algorithms**</u>

To understand GANs, you should know how generative algorithms work, and for that, contrasting them with discriminative algorithms is instructive. Discriminative algorithms try to classify input data; that is, given the features of an instance of data, they predict a label or category to which that data belongs.
For example, given all the words in an email (the data instance), a discriminative algorithm could predict whether the message is spam or not_spam. spam is one of the labels, and the bag of words gathered from the email are the features that constitute the input data. When this problem is expressed mathematically, the label is called y and the features are called x.

The formulation p(y|x) is used to mean “the probability of y given x”, which in this case would translate to “the probability that an email is spam given the words it contains.”
So discriminative algorithms map features to labels. They are concerned solely with that correlation. One way to think about generative algorithms is that they do the opposite. Instead of predicting a label given certain features, they attempt to predict features given a certain label.
The question a generative algorithm tries to answer is: Assuming this email is spam, how likely are these features? While discriminative models care about the relation between y and x, generative models care about “how you get x.” They allow you to capture p(x|y), the probability of x given y, or the probability of features given a label or category. (That said, generative algorithms can also be used as classifiers. It just so happens that they can do more than categorize input data.)

Another way to think about it is to distinguish discriminative from generative like this:

- Discriminative models learn the boundary between classes
- Generative models model the distribution of individual classes

One neural network, called the *generator*, generates new data instances, while the other, the *discriminator*, evaluates them for authenticity; i.e. the discriminator decides whether each instance of data that it reviews belongs to the actual training dataset or not.
Let’s say we’re trying to do something more banal than mimic the Mona Lisa. We’re going to generate hand-written numerals like those found in the MNIST dataset, which is taken from the real world. The goal of the discriminator, when shown an instance from the true MNIST dataset, is to recognize those that are authentic.
Meanwhile, the generator is creating new, synthetic images that it passes to the discriminator. It does so in the hopes that they, too, will be deemed authentic, even though they are fake. The goal of the generator is to generate passable hand-written digits: to lie without being caught. The goal of the discriminator is to identify images coming from the generator as fake.

Here are the steps a GAN takes:

- The generator takes in random numbers and returns an image.
- This generated image is fed into the discriminator alongside a stream of images taken from the actual, ground-truth dataset.
- The discriminator takes in both real and fake images and returns probabilities, a number

between 0 and 1, with 1 representing a prediction of authenticity and 0 representing fake.

So, you have a double feedback loop:

- The discriminator is in a feedback loop with the ground truth of the images, which we know.
- The generator is in a feedback loop with the discriminator.

You can think of a GAN as the opposition of a counterfeiter and a cop in a game of cat and mouse, where the counterfeiter is learning to pass false notes, and the cop is learning to detect them. Both are dynamic; i.e. the cop is in training, too (to extend the analogy, maybe the central bank is flagging bills that slipped through), and each side comes to learn the other's methods in a constant escalation.
For MNIST, the discriminator network is a standard convolutional network that can categorize the images fed to it, a binomial classifier labeling images as real or fake. The generator is an inverse convolutional network, in a sense: While a standard convolutional classifier takes an image and downsamples it to produce a probability, the generator takes a vector of random noise and upsamples it to an image. The first throws away data through downsampling techniques like maxpooling, and the second generates new data.
Both nets are trying to optimize a different and opposing objective function, or loss function, in a zero-zum game. This is essentially an actor-critic model. As the discriminator changes its behavior, so does the generator, and vice versa. Their losses push against each other.

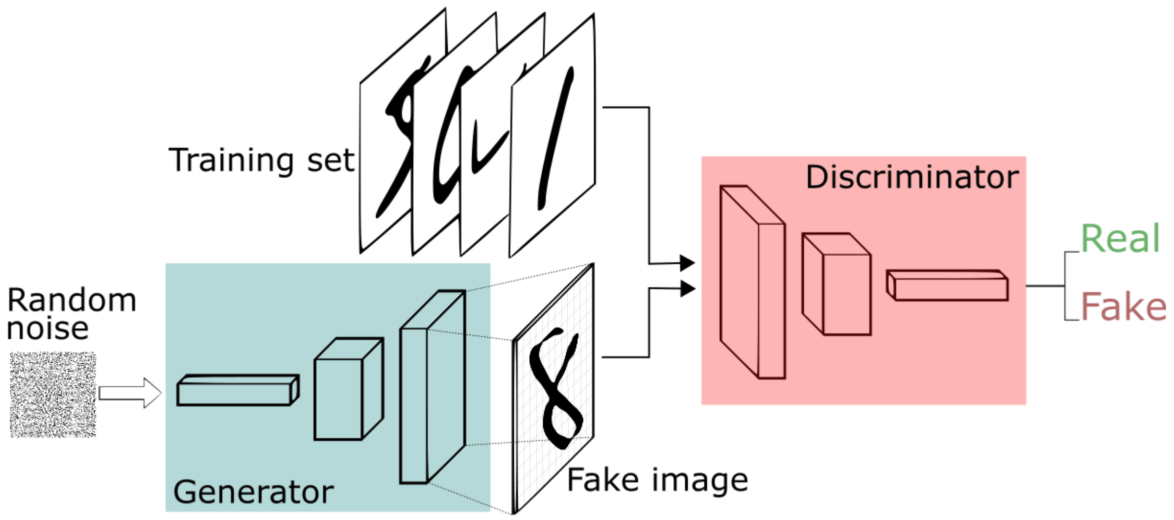

Fig 4.2 Functioning of a GAN

4.3 Cycle GAN

CycleGAN is a Generative Adversarial Network (GAN) that uses *two*generators and *two* discriminators

We call one generator *G*, and have it converted images from the *X* domain to the *Y* domain. The other generator is called *F*, and converts images from *Y* to *X*.

$$G : X \to Y$$
$$F : Y \to X$$

Both G and F are generators that take an image from one domain and translate it to another. G maps from X to Y, whereas F goes in the opposite direction, mapping Y to X.

Each generator has a corresponding discriminator, which attempts to tell apart its synthesized images from real ones.

D_y : Distinguishes y from G(x)

D_x : Distinguishes x from F(y)

One discriminator provides adversarial training for G, and the other does the same for F.

4.3.1 The Objective Function

There are two components to the CycleGAN objective function, an *adversarial loss* and a *cycle consistency loss*. Both are essential to getting good results.

If you are familiar with GANs, the adversarial loss should come as no surprise. Both generators are attempting to "fool" their corresponding discriminator into being less able to distinguish their generated images from the real versions.

$$Loss_{adv}(G, D_y, X) = \frac{1}{m}\sum_{i=1}^{m}(1 - D_y(G(x_i)))^2$$

$$Loss_{adv}(F, D_x, Y) = \frac{1}{m}\sum_{i=1}^{m}(1 - D_x(F(y_i)))^2$$

However, the adversarial loss alone is not sufficient to produce good images, as it leaves the model *under-constrained*. It enforces that the generated output be of the appropriate domain but does *not* enforce that the input and output are recognizably the same. For example, a generator that output an image *y* that was an excellent example of that domain, but looked

nothing like *x*, would do well by the standard of the adversarial loss, despite not giving us what we really want.
The *cycle consistency loss* addresses this issue. It relies on the expectation that if you convert an image to the other domain and back again, by successively feeding it through both generators, you should get back something similar to what you put in. It enforces that $F(G(x)) \approx x$ and $G(F(y)) \approx y$.

$$Loss_{cyc}(G, F, X, Y) = \frac{1}{m}\sum_{i=1}^{m}[F(G(x_i)) - x_i] + [G(F(y_i)) - y_i]$$

We can create the full objective function by putting these loss terms together, and weighting the cycle consistency loss by a hyperparameter λ. We suggest setting λ = 10.

$$Loss_{full} = Loss_{adv} + \lambda Loss_{cyc}$$

4.4 Cycle GAN architecture

The CycleGAN works without paired examples of transformation from source to target domain. Recent methods such as Pix2Pix depend on the availability of training examples where the same data is available in both domains. The power of CycleGAN lies in being able to learn such transformations without one-to-one mapping between training data in source and target domains. The need for a paired image in the target domain is eliminated by making a two-step transformation of source domain image - first by trying to map it to target domain and then back to the original image. Mapping the image to target domain is done using a generator network and the quality of this generated image is improved by pitching the generator against a discriminator (as described below)
We have a generator network and discriminator network playing against each other. The generator tries to produce samples from the desired distribution and the discriminator tries to predict if the sample is from the actual distribution or produced by the generator. The generator and discriminator are trained jointly. The effect this has is that eventually the generator learns to approximate the underlying distribution completely and the discriminator is left guessing randomly.

The above adversarial method of training has a problem though. Quoting the authors of the original paper:

Adversarial training can, in theory, learn mappings GG and FF that produce outputs identically distributed as target domains YY and XX respectively. However, with large enough capacity, a network can map the same set of input images to any random permutation of images in the target domain, where any of the learned mappings can induce an output distribution that matches the target distribution. Thus, an adversarial loss alone cannot guarantee that the learned function can map an individual input xixi to a desired output yiyi.

To regularize the model, the authors introduce the constraint of cycle-consistency - if we transform from source distribution to target and then back again to source distribution, we should get samples from our source distribution.

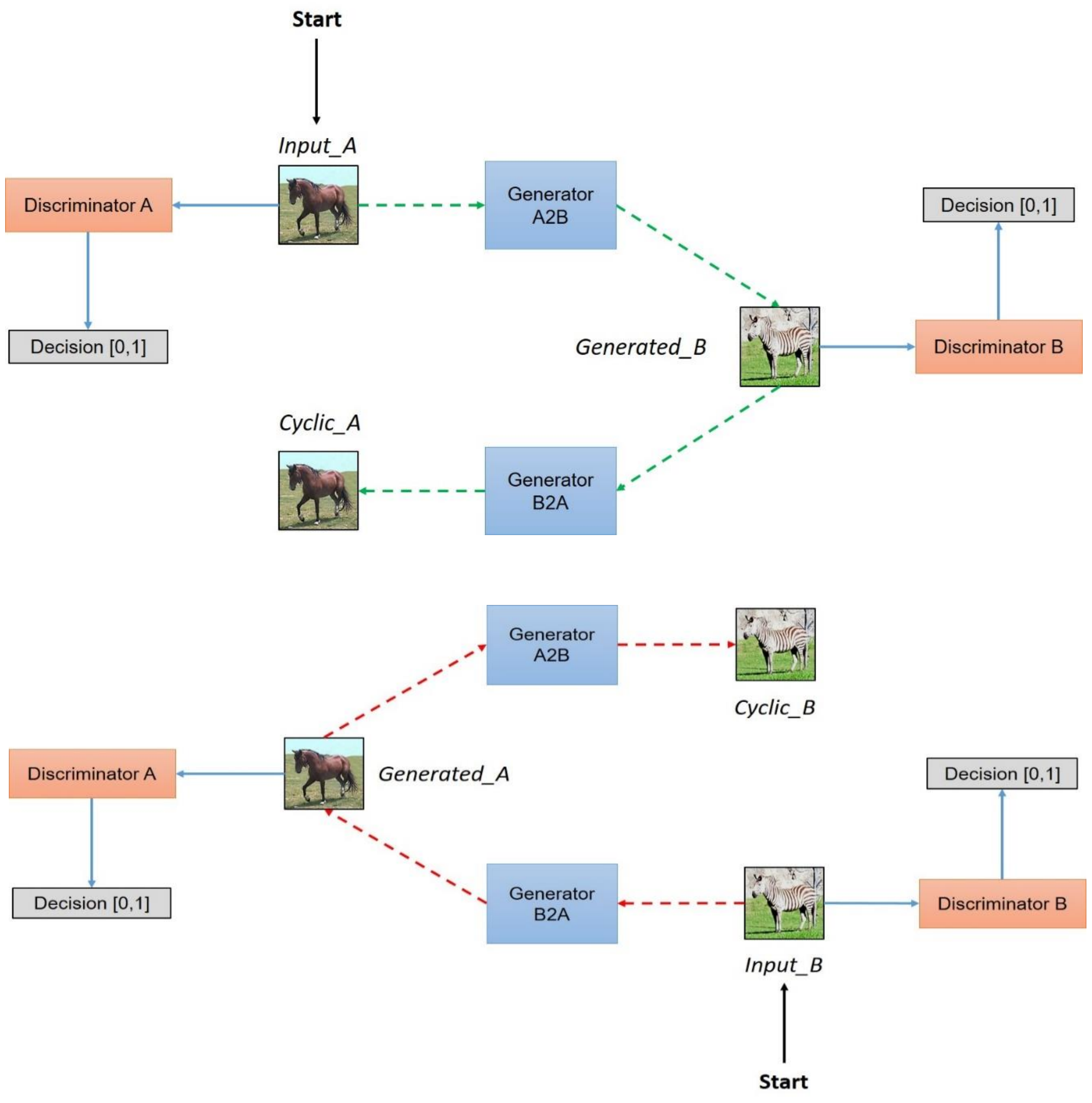

Fig 4.3 Simplified view of CycleGAN architecture

In a paired dataset, every image, say imgAimgA, is manually mapped to some image, say imgBimgB, in target domain, such that they share various features. Features that can be used to map an image (imgA/imgB)(imgA/imgB) to its correspondingly mapped counterpart (imgB/imgA)(imgB/imgA). Basically, pairing is done to make input and output share some common features. This mapping defines meaningful transformation of an image from one damain to another domain. So, when we have paired dataset, generator must take an input, say inputAinputA, from domain DADA and map this image to an output image, say genBgenB, which must be close to its mapped counterpart. But we don't have this luxury in unpaired dataset, there is no pre-defined meaningful transformation that we can learn, so, we will create it. We need to make sure that there is some meaningful relation between input image and generated image. So, authors tried to enforce this by saying that Generator will map input image (inputA)(inputA) from domain DADA to some image in target domain DBDB, but to make sure that there is meaningful relation between these images, they must share some feature, features that can be used to map this output image back to input image, so there must be another generator that must be able to map back this output image back to original input. So, you can see this condition defining a meaningful mapping between inputAinputA and genBgenB.

In a nutshell, the model works by taking an input image from domain DADA which is fed to our first generator GeneratorA→BGeneratorA→B whose job is to transform a given image from domain DADA to an image in target domain DBDB. This new generated image is then fed to another generator GeneratorB→AGeneratorB→A which converts it back into an image, CyclicACyclicA, from our original domain DADA (think of autoencoders, except that our latent space is DtDt). And as we discussed in above paragraph, this output image must be close to original input image to define a meaningful mapping that is absent in unpaired dataset.

As you can see in above figure, two inputs are fed into each discriminator(one is original image corresponding to that domain and other is the generated image via a generator) and the job of discriminator is to distinguish between them, so that discriminator is able to defy the adversary (in this case generator) and reject images generated by it. While the generator would like to make sure that these images get accepted by the discriminator, so it will try to generate images which are very close to original images in Class DBDB. (In fact, the generator and discriminator are actually playing a game whose Nash equilibrium is achieved when the generator's distribution becomes same as the desired distribution)

4.4.1 Generator Architecture

Each CycleGAN generator has three sections: an *encoder*, a *transformer*, and a *decoder*. The input image is fed directly into the encoder, which shrinks the representation size while increasing the number of channels. The encoder is composed of three convolution layers. The resulting activation is then passed to the transformer, a series of six residual blocks. It is then expanded again by the decoder, which uses two transpose convolutions to enlarge the representation size, and one output layer to produce the final image in RGB.
You can see the details in the figure below. Please note that each layer is followed by an instance normalization and a ReLU layer, but these have been omitted for simplicity.

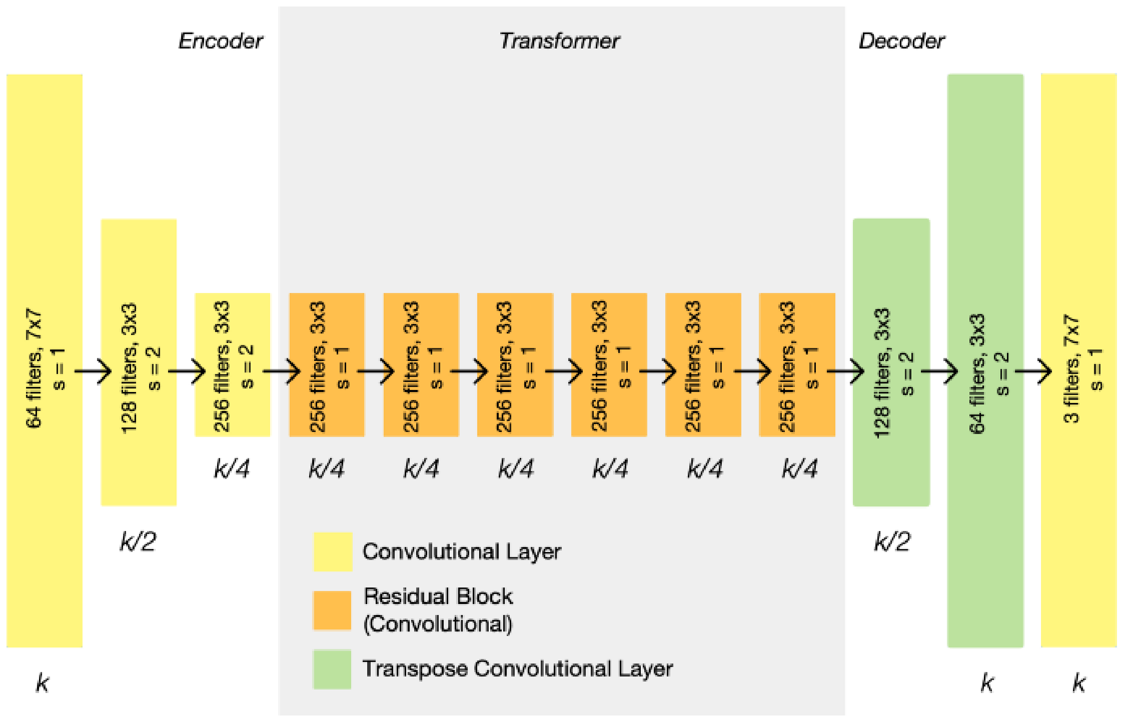

Fig 4.4 An architecture for a CycleGAN generator.

An architecture for a CycleGAN generator. As you can see above, the representation size shrinks in the encoder phase, stays constant in the transformer phase, and expands again in the decoder phase. The representation size that each layer outputs is listed below it, in terms of the input image size, k. On each layer is listed the number of filters, the size of those filters, and the stride. Each layer is followed by an instance normalization and ReLU activation.
Since the generators' architecture is fully convolutional, they can handle arbitrarily large input once trained.

4.4.2 Discriminator Architecture

The discriminators are PatchGANs, fully convolutional neural networks that look at a “patch” of the input image and output the probability of the patch being “real”. This is both more computationally efficient than trying to look at the entire input image and is also more effective — it allows the discriminator to focus on more surface-level features, like texture, which is often the sort of thing being changed in an image translation task.

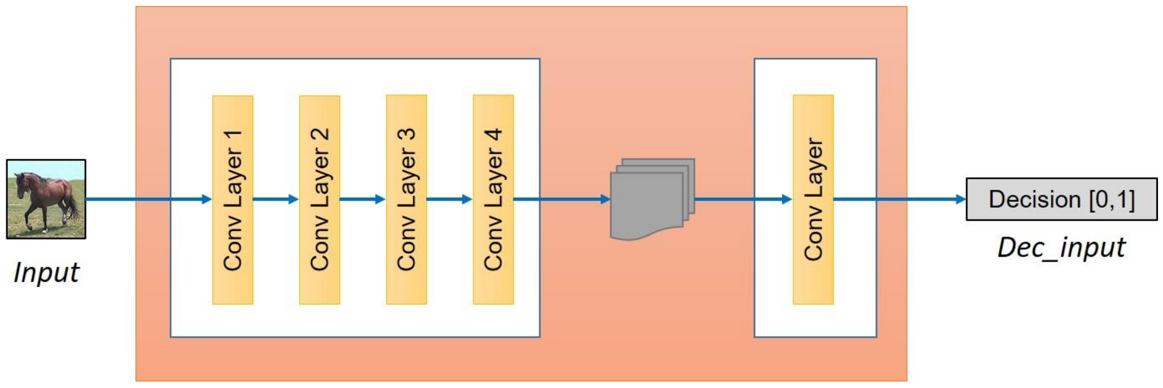

Fig 4.5 An architecture for a CycleGAN discriminator

CHAPTER 5
LIBRARIES USED

5.1 FastAI

The fastai library simplifies training fast and accurate neural nets using modern best practices. It's based on research in to deep learning best practices undertaken at fast.ai, including "out of the box" support for vision, text, tabular, and collab (collaborative filtering) models.

5.1.1 fastai vision

The vision module of the fastai library contains all the necessary functions to define a Dataset and train a model for computer vision tasks. It contains four different submodules to reach that goal:

- vision.image contains the basic definition of an Image object and all the functions that are used behind the scenes to apply transformations to such an object.
- vision.transform contains all the transforms we can use for data augmentation.
- vision.data contains the definition of ImageDataBunch as well as the utility function to easily build a DataBunch for Computer Vision problems.
- vision.learner lets you build and fine-tune models with a pretrained CNN backbone or train a randomly initialized model from scratch.

Each of the four-module links above includes a quick overview and examples of the functionality of that module, as well as complete API documentation. Below, we'll provide a walk-thru of end to end computer vision model training with the most commonly used functionality.

5.1.2 The data block API

The data block API lets you customize the creation of a DataBunch by isolating the underlying parts of that process in separate blocks.
Each of these may be addressed with a specific block designed for your unique setup. Your inputs might be in a folder, a csv file, or a dataframe. You may want to split them randomly, by certain indices or depending on the folder they are in. You can have your labels in your csv file or your dataframe, but it may come from folders or a specific function of the input.

You may choose to add data augmentation or not. A test set is optional too. Finally you have to set the arguments to put the data together in a DataBunch (batch size, collate function...)
The data block API is called as such because you can mix and match each one of those blocks with the others, allowing for a total flexibility to create your customized DataBunch for training, validation and testing. The factory methods of the various DataBunch are great for beginners but you can't always make your data fit in the tracks they require.

5.1.3 Training

The fastai library structures its training process around the Learner class, whose object binds together a PyTorch model, a dataset, an optimizer, and a loss function; the entire learner object then will allow us to launch training.
basic_train defines this Learner class, along with the wrapper around the PyTorch optimizer that the library uses. It defines the basic training loop that is used each time you call the fit method (or one of its variants) in fastai. This training loop is very bare-bones and has very few lines of codes; you can customize it by supplying an optional Callback argument to the fit method.
callback defines the Callback class and the CallbackHandler class that is responsible for the communication between the training loop and the Callback's methods. The CallbackHandler maintains a state dictionary able to provide each Callback object all the information of the training loop it belongs to, putting any imaginable tweaks of the training loop within your reach.
callbacks implements each predefined Callback class of the fastai library in a separate module. Some modules deal with scheduling the hyperparameters, like callbacks.one_cycle, callbacks.lr_finder and callback.general_sched. Others allow special kinds of training like callbacks.fp16 (mixed precision) and callbacks.rnn. The Recorder and callbacks.hooks are useful to save some internal data generated in the training loop.

CHAPTER 6
PROJECT DESCRIPTION

6.1 Dateset

The dataset used is the Yosemite summer2winter dataset which contains images of Yosemite National Park in both summer and winter. We have a total of 2744 images

6.2 Data Preprocessing

Below we enumerate the steps followed to prepare our data for the study.

6.2.1 Resizing of Images

In order to train our model effectively under hardware constraints of GPU memory and memory, we had to scale down our images to a uniform, smaller size of 64 x 64 pixels before we could use the images to train out model. This was done because pre-trained models tend to perform better on square images of similar sizes.

6.2.2. Date Augmentation

On application of the NN models, certain basic transformations were applied to the input dataset. These transformations include rotation, flipping, slight change in lighting and zooming. In our model, these transformations are applied to images with a probability of 75%. Augmentations help to prevent out models from being overfitted.

6.3 Implementation

We implemented CycleGAN using fastai . We used Resnet Blocks in our generator, and we created two resnet generators and two discriminators.

The main loss used to train the generators. It has three parts:

- the classic GAN loss: they must make the critics believe their images are real
- identity loss: if they are given an image from the set they are trying to imitate, they should return the same thing
- cycle loss: if an image from A goes through the generator that imitates B then through the generator that imitates A, it should be the same as the initial image. Same for B and switching the generators

6.3.1 Structure of Discriminator

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (1): LeakyReLU(negative_slope=0.2, inplace)
  (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (3): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
  (4): LeakyReLU(negative_slope=0.2, inplace)
  (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
  (7): LeakyReLU(negative_slope=0.2, inplace)
  (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
  (9): InstanceNorm2d(512, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
  (10): LeakyReLU(negative_slope=0.2, inplace)
  (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
)
```

6.3.2 Structure of Generator

```
Sequential(
  (0): ReflectionPad2d((3, 3, 3, 3))
  (1): Conv2d(3, 64, kernel_size=(7, 7), stride=(1, 1))
  (2): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
  (3): ReLU(inplace)
  (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (5): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
  (6): ReLU(inplace)
  (7): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (8): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
  (9): ReLU(inplace)
  (10): ResnetBlock(
    (conv_block): Sequential(
      (0): ReflectionPad2d((1, 1, 1, 1))
      (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
      (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      (3): ReLU(inplace)
      (4): ReflectionPad2d((1, 1, 1, 1))
      (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
      (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    )
  )
  (11): ResnetBlock(
    (conv_block): Sequential(
      (0): ReflectionPad2d((1, 1, 1, 1))
      (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
      (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      (3): ReLU(inplace)
      (4): ReflectionPad2d((1, 1, 1, 1))
      (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
      (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    )
  )
  (12): ResnetBlock(
    (conv_block): Sequential(
      (0): ReflectionPad2d((1, 1, 1, 1))
      (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
      (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      (3): ReLU(inplace)
      (4): ReflectionPad2d((1, 1, 1, 1))
```

```
          (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
          (6):   InstanceNorm2d(256,   eps=1e-05,   momentum=0.1,   affine=False,
track_running_stats=False)
       )
    )
    (13): ResnetBlock(
       (conv_block): Sequential(
          (0): ReflectionPad2d((1, 1, 1, 1))
          (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
          (2):   InstanceNorm2d(256,   eps=1e-05,   momentum=0.1,   affine=False,
track_running_stats=False)
          (3): ReLU(inplace)
          (4): ReflectionPad2d((1, 1, 1, 1))
          (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
          (6):   InstanceNorm2d(256,   eps=1e-05,   momentum=0.1,   affine=False,
track_running_stats=False)
       ) )
    (14): ResnetBlock(
       (conv_block): Sequential(
          (0): ReflectionPad2d((1, 1, 1, 1))
          (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
          (2):   InstanceNorm2d(256,   eps=1e-05,   momentum=0.1,   affine=False,
track_running_stats=False)
          (3): ReLU(inplace)
          (4): ReflectionPad2d((1, 1, 1, 1))
          (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
          (6):   InstanceNorm2d(256,   eps=1e-05,   momentum=0.1,   affine=False,
track_running_stats=False)
       ) )
    (15): ResnetBlock(
       (conv_block): Sequential(
          (0): ReflectionPad2d((1, 1, 1, 1))
          (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
          (2):   InstanceNorm2d(256,   eps=1e-05,   momentum=0.1,   affine=False,
```