Project Report: Image-to-Image Translation

Gurdit Singh Siyan
gurdit@cmi.ac.in
gurdit.siyan@gmail.com

Chennai Mathematical Institute

Contents

Chapter 1

Introduction

Many problems in image processing, graphics, and vision involve translating an input image into a corresponding output image. There exist a wide variety of domain specific solutions for many problems, even though the goal is always the same: mapping pixels to pixels. The objective is always to find a map from an input domain X to an output domain Y. That is, for image domains X and Y, we want to find a function $G : X \rightarrow Y$. One of the questions we can ask is: does there exist a general solution to this problem?

Solving such a general problem using traditional Computer Vision techniques and algorithms can indeed feel impossible. However, there has been a relatively recent breakthrough in the field of deep learning. The introduction of a class of generative deep learning models known as Generative Adversarial Networks [3], can be used to approach generative problems which were considered very hard(or downright impossible) for a long time.

The goal of this project is to implement two of the most influential models in two settings of the Image-to-Image translation problem:

- **Supervised setting**: In this setting we have paired input-output images between domain X and domain Y.
- **Unsupervised setting**: In this harder setting we just have two image domains(sets) X and Y and paired examples are unavailable.

Figure 1.1: Colorization: Black & White → Color

Figure 1.2: Semantic Segmentation: Cityscape → Segmented Image

Figure 1.3: Style Transfer: Portrait → Stylized Image

Figure 1.4: Background Removal: Image → Image without Background

Chapter 2

Background

2.1 Neural Networks

Neural networks [2], also called **feedforward neural networks** or **multilayer perceptrons** (MLPs) are a network or a circuit composed of artificial neurons. The objective for a neural network is to approximate some function f^*. For example, for a classifier, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to a category y. A neural network defines a mapping $\hat{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the value of the weights(or parameters) θ that result in the best function approximation.

These models are called **networks** because they are represented by composing together many different functions called **layers**. And, these models are called **feedforward** because to evaluate the network at some value $\boldsymbol{x}$ we pass it through a sequence of layers whose result depend upon the values of previous layers, i.e., under evaluation the computation flows in one direction only.

2.1.1 Computing with Neural Networks

The model is associated with a directed acyclic graph describing how the functions are composed together. For example, in the figure 2.1 we have 3 layers(functions) $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$ connected in a chain to form the network $f(\boldsymbol{x}) = f^{(1)}(f^{(2)}(f^{(3)}(\boldsymbol{x})))$, the final function $f^{(3)}$ is called the output layer.

These layer functions $f^{(i)}$ in general look like:

$$f^{(i)}(\boldsymbol{a}^{(i-1)}) = g^{(i)}((\boldsymbol{W}^{(i)})^T \boldsymbol{a}^{(i-1)} + \boldsymbol{b}^{(i)}) \tag{2.1}$$

Here $\boldsymbol{a}^{(i-1)}$ is the result vector of computation of the previous layer, note that $\boldsymbol{a}^{(0)}$ is defined to be $\boldsymbol{x}$. $\boldsymbol{W}^{(i)}$ is defined to be the weight matrix for the layer i where each column correspond to weighted edges connected to a neuron, and $\boldsymbol{b}^{(i)}$ is the bias vector for the layer (i). To elaborate if the number of elements

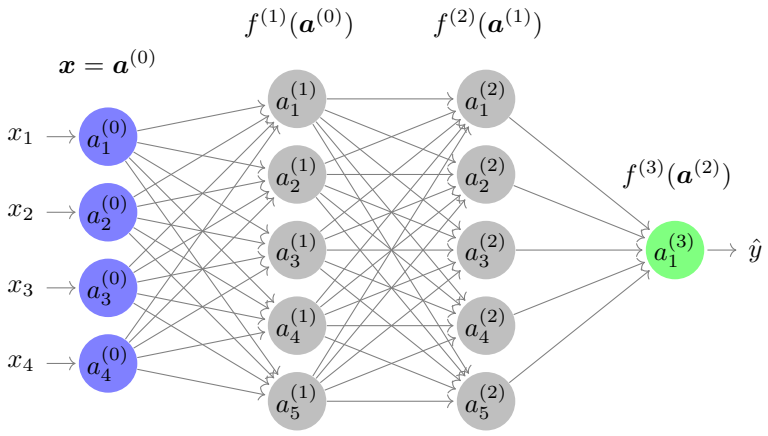


Figure 2.1: A simple neural network.

in vector $\boldsymbol{a}^{(i-1)}$ is $n^{(i-1)}$ and in $\boldsymbol{a}^{(i)}$ is $n^{(i)}$, then the shape of matrix $\boldsymbol{W}^{(i)}$ is $(n^{(i-1)} \times n^{(i)})$ and length for the vector $\boldsymbol{b}$ is $n^{(i)}$. Also, $g^{(i)} : \mathbb{R} \to \mathbb{R}$ is just a real valued function which is applied(broadcasted) to every element of the input vector. Some commonly used activation functions are:

- **Linear** is just the identity function.

$$g(x) = x$$

- **ReLU** (Rectified Linear Unit) activation function is an activation function defined as the positive part of its argument:

$$g(x) = \max(0, x)$$

- **Sigmoid** function is a mathematical function having a characteristic "S"-shaped curve or sigmoid curve. Some examples are:

Logistic:	$g(x) = \dfrac{1}{1 + e^{-x}}$
Hyperbolic Tangent:	$g(x) = \tanh x$

2.1.2 Learning Parameters for Neural Networks

Neural networks are trained similarly to other machine learning models and that is via Gradient Descent.

Let the training set for the model be defined as:

$$\{(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), (\boldsymbol{x}^{(2)}, \boldsymbol{y}^{(2)}), \ldots, (\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}), \ldots, (\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})\}$$

Where $\boldsymbol{x}^{(i)}$ is the input and $\boldsymbol{y}^{(i)}$ is the corresponding output. We also need to define a **cost function** J which for parameters $\boldsymbol{\theta}$ measures the distance between the predicted vector $f(\boldsymbol{x}^{(i)}) = \hat{\boldsymbol{y}}^{(i)}$ and the actual vector $\boldsymbol{y}^{(i)}$. One common loss function is the **Mean Squared Error** (MSE) loss, defined as:

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)})^2$$

Then we use any form of gradient based learning to minimize this loss function. For example, for **Stochastic Gradient Descent**(SGD) we estimate the weights as follows:

Algorithm 1 Stochastic Gradient Descent training for Neural Networks

- Initialize random weights $\boldsymbol{\theta}$ and chose learning rate η.

for epoch $\leftarrow$ number of training iterations **do**
  for $i \leftarrow$ number of training samples **do**
    - Update $\boldsymbol{\theta}$ by descending its gradient evaluated at sample i.

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \cdot \nabla_{\theta} J^{(i)}(\boldsymbol{\theta})$$

  end for
end for

2.2 Convolutional Neural Networks

Convolutional Neural Networks [2] or CNNs, are a specialized form of Neural Network for processing input which has a sequential grid-like format. For example, in one dimensional data we can use CNNs for time-series data, natural language represented as a sequence of word vectors, etc., and in two dimensions we most commonly use CNNs for images. We will only discuss 2D CNNs in this section.

The input to a CNN is a tensor of shape $m \times h \times w \times c$, where m is the number of samples, h and w are the height and width of the input image and c is the number of channels in the image.

For CNNs, in addition to **fully connected** layers as discussed in the previous section, we also have extra kind of layers such as **convolution** and **pooling** layers.

2.2.1 Convolution Layers

Convolution layers "convolve" the input and pass its result to the next layer. The convolution operation is a mathematical operation defined as follows, if we have an input image I, and we apply a convolution filter K(which is also a

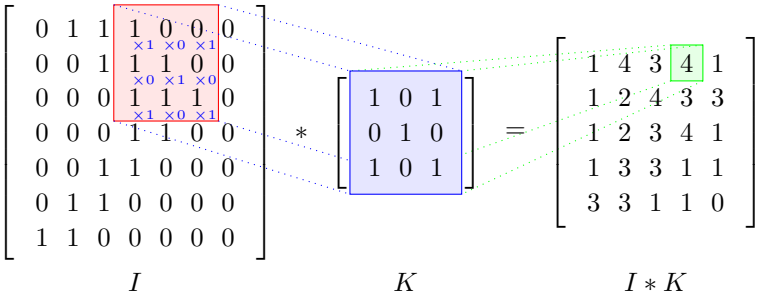


Figure 2.2: Cross-correlation in 2D.

matrix), the resulting computation would be a matrix $S = I * k$. Note: here $*$ is the convolution operator.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

However, in practice we implement a related function called the **cross-correlation** 2.2, which is the same as convolution except we don't flip the kernel.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

For CNNs the entire matrix K is learned during training.

2.2.2 Pooling Layers

After passing the input to some convolution layers and activation functions, we also use a new kind of layer called the pooling layer. A pooling layer, like an activation function, contains no learnable parameters but rather is just a normal function.

Similar to a convolution, pooling layer is applied to a rectangular region and returns a single number. Some common examples of pooling layers include max pooling, which takes the maximum value in a region, average pooling with takes the average value, weighted average, L^2 norm of the region etc.

2.2.3 Architecture of a CNN

A CNN usually consist of many alternating convolution-pooling layer pairs and ultimately connected to a fully connected layer. Learning the weights for CNN work similarly to a normal neural network.

2.3 Generative Adversarial Networks

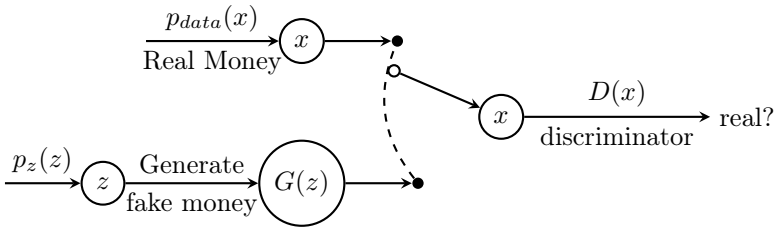


Figure 2.3: A visual representation of Generative Adversarial Network

Generative adversarial networks [3] are a framework for estimating generative models via an adversarial process, in which two models: a generative model **G** that captures the data distribution, and a discriminative model **D** that estimates the probability that the sample came from the training data rather than **G**, are trained simultaneously.

The canonical example to understand GANs is as follows: imagine G as a fraud trying to create fake currency and use it without detection and imagine D is working for the police who is trying to detect counterfeit money.

The goal of the generator is to generate samples that are indistinguishable from the real data, i.e., $p_G(\boldsymbol{x}) = p_{\text{data}}(\boldsymbol{x})$.

To build the optimization function for the discriminator D, we want to make it recognized the samples from the actual distribution. That is, we try to maximize:

$$\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})} \log D(\boldsymbol{x})$$

But we also want to make it so that the discriminator D rejects samples generated by G:

$$\mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})} \log(1 - D(G(\boldsymbol{z})))$$

Maximizing these terms for D mean $\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})$ implies $D(\boldsymbol{x}) \approx 1$, and $D(G(\boldsymbol{z})) \approx 0$.

The generator G will try to minimize these terms, which makes this a two-player minimax game with the value function $V(G, D)$:

$$\min_G \max_D V(G, D) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})} \log D(\boldsymbol{x}) + \mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})} \log(1 - D(G(\boldsymbol{z}))) \quad (2.2)$$

2.3.1 Training the Network

The Generative Adversarial Network can be trained by training both the discriminator D and the generator G on a turn by turn basis.

Note, in practice rather than minimizing(maximizing) $\log(1 - D(G(\boldsymbol{z})))$ we maximize(minimize) $\log(D(G(\boldsymbol{z})))$ which provides much stronger gradients early in the training when G is weak, but results in the same fixed point dynamics.

Algorithm 2 Minibatch SGD training of Generative Adversarial Networks

for number of training iterations **do**

for k steps **do**

- Sample minibatch m noise examples $\{\boldsymbol{z}^{(1)}, \boldsymbol{z}^{(2)}, \ldots, \boldsymbol{z}^{(m)}\}$ from $p_z(\boldsymbol{z})$.
- Sample minibatch m examples $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$ from the actual distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the Discriminator D by ascending its stochastic gradient:

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

end for

- Sample minibatch m noise examples $\{\boldsymbol{z}^{(1)}, \boldsymbol{z}^{(2)}, \ldots, \boldsymbol{z}^{(m)}\}$ from $p_z(\boldsymbol{z})$.
- Update the Generator G by descending its stochastic gradient:

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

end for

2.3.2 Theoretical Results

In the space of arbitrary functions G and D, a unique solution exists, with G recovering the training data and D being equal to $\frac{1}{2}$ everywhere.

Proposition 1. *For a fixed G, the optimal discriminator D is*

$$D^*_G(\boldsymbol{x}) = \frac{p_{data}(\boldsymbol{x})}{p_{data}(\boldsymbol{x}) + p_G(\boldsymbol{x})} \tag{2.3}$$

Proof. The training criterion for D is to maximize the following function:

$$\begin{aligned} V(G, D) &= \int_x p_{\text{data}}(\boldsymbol{x}) \log D(\boldsymbol{x})\, dx + \int_z p_z(\boldsymbol{z}) \log(1 - D(G(\boldsymbol{z})))\, dz \\ &= \int_x \left(p_{\text{data}}(\boldsymbol{x}) \log D(\boldsymbol{x}) + p_G(\boldsymbol{x}) \log(1 - D(G(\boldsymbol{x}))) \right) dx \end{aligned}$$

The equation reduces to finding the maximum for the equation

$$f(y) = a \log y + b \log(1 - y)$$

To find the maxima, we calculate the derivatives:

$$f'(y) = \frac{a}{y} - \frac{b}{1-y}$$
$$f'(y) = -\frac{a}{y^2} - \frac{b}{(1-y)^2}$$

We have the extreme point when,

$$f'(y) = 0 \Rightarrow y = \frac{a}{a+b}$$

But also note:

$$f''(\frac{a}{a+b}) = -\frac{a}{(\frac{a}{a+b})^2} - \frac{b}{(1-(\frac{a}{a+b}))^2} < 0$$

So, the function f is maximum at $\frac{a}{a+b}$, which means $V(G,D)$ is maximum when you let

$$D_G^*(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x})}$$

□

By the previous proposition we can see that for an optimal generator G i.e., when $p_G(\boldsymbol{x}) = p_{\text{data}}(\boldsymbol{x})$, there exists a unique discriminator D, which is:

$$\begin{aligned} D_G^*(\boldsymbol{x}) &= \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x})} \\ &= \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_{\text{data}}(\boldsymbol{x})} \\ &= \frac{1}{2} \end{aligned}$$

Theorem 1. *The global minimum of the virtual training criterion* $C(G) = \max_D V(G,D)$ *is reached if and only if* $p_G(\boldsymbol{x}) = p_{data}(\boldsymbol{x})$*, and the minimum value is* $-\log(4)$

Proof. ($\Leftarrow$) Assuming $p_G(\boldsymbol{x}) = p_{\text{data}}(\boldsymbol{x})$,

$$\begin{aligned} V(G, D_G^*) &= \int_x \left(p_{\text{data}}(\boldsymbol{x}) \log \frac{1}{2} + p_{\text{data}}(\boldsymbol{x}) \log(1 - \frac{1}{2}) \right) dx \\ &= -2 \log 2 \int_x p_{\text{data}}(\boldsymbol{x})\, dx \\ &= -2 \log 2 \\ &= -\log 4 \end{aligned}$$

We will later prove that this is the absolute minimum.

($\Rightarrow$) Plugging in value for D_G^* we get,

$$\begin{aligned} C(G) &= \int_x \Big(p_{\text{data}}(\boldsymbol{x}) \log \left(\frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x})} \right) \\ &\quad + p_G(\boldsymbol{x}) \log \left(1 - \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x})} \right) \Big) \, dx \\ &= \int_x \Big(p_{\text{data}}(\boldsymbol{x}) \log \left(\frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x})} \right) \\ &\quad + p_G(\boldsymbol{x}) \log \left(\frac{p_G(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x})} \right) \Big) \, dx \end{aligned}$$

Adding and subtracting $\log 2$

$$\begin{aligned} &= \int_x \Big(p_{\text{data}}(\log 2 - \log 2) + p_{\text{data}}(\boldsymbol{x}) \log \left(\frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x})} \right) \\ &\quad + p_G(\log 2 - \log 2) + p_G(\boldsymbol{x}) \log \left(\frac{p_G(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x})} \right) \Big) \, dx \\ &= -\log 2 \int_x p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x}) \\ &\quad + \int_x p_{\text{data}}(\boldsymbol{x}) \log \left(\frac{p_{\text{data}}(\boldsymbol{x})}{(p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x}))/2} \right) \, dx \\ &\quad + \int_x p_G(\boldsymbol{x}) \log \left(\frac{p_G(\boldsymbol{x})}{(p_{\text{data}}(\boldsymbol{x}) + p_G(\boldsymbol{x}))/2} \right) \, dx \\ &= -\log 4 + KL \left(p_{\text{data}} \middle| \frac{p_{\text{data}} + p_G}{2} \right) + KL \left(p_G \middle| \frac{p_{\text{data}} + p_G}{2} \right) \end{aligned}$$

Here KL is the Kullback-Leibler divergence, which is always non-negative, and thus validates our assumption that $-\log 4$ is the global minimum.

Also note, we can rewrite the last equation as

$$= -\log 4 + 2 \cdot JSD(p_{\text{data}} \mid p_G)$$

where JSD is the Jenson-Shannon divergence, which is 0 only when $p_{\text{data}} = p_G$, which means the global minimum occurs when $p_{\text{data}} = p_G$, i.e., the generative model perfectly replicates the training data. $\square$

Chapter 3

Supervised Image-to-Image Translation

In supervised Image-to-Image translation there exist "paired" examples of input-ouput images for the problem we are solving.

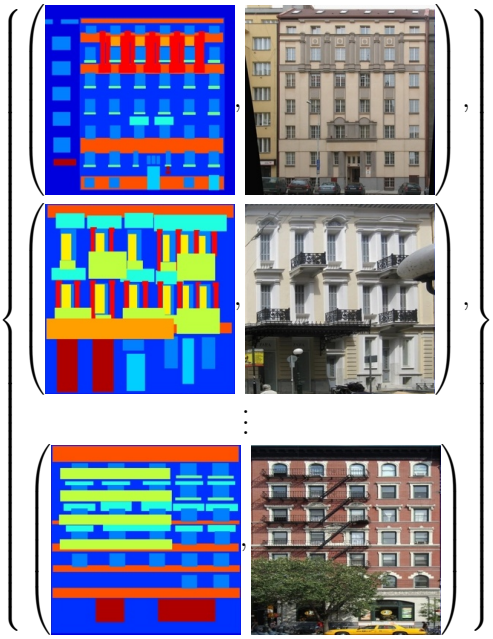
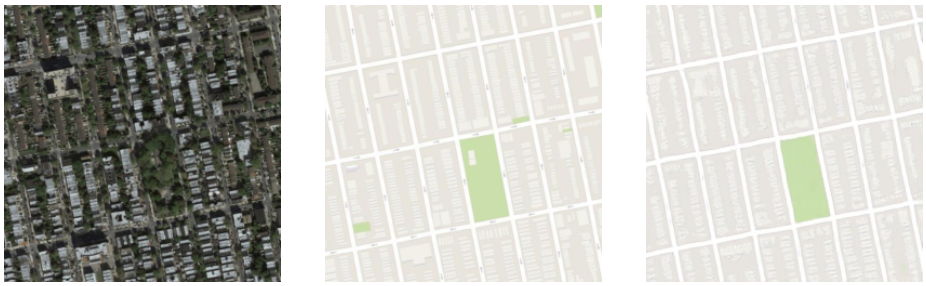


Figure 3.1: Supervised Image-to-Image Translation



(a) Input Image (b) Paired Output Image (c) Predicted Image by G

Figure 3.2: Example output on Satellite2Map dataset

3.1 Pix2Pix

Pix2Pix was one of the first general-purpose Image-to-Image translation network which was introduced in the seminal paper "Image-to-Image Translation with Conditional Adversarial Networks" [4]. They applied the same architecture on a wide variety of problems and got very promising results.

The first question we should ask before using GANs for this problem is: do we need GANs at all? or can we just use a simple CNN? There exist several generative CNN models — for example Encoder-Decoder, and U-Net — so why don't we use them? The problem is that CNNs learn to minimize a loss function and although the learning process is automatic, we still need to carefully design the loss function. That is, we need to tell the CNN what we want to minimize.

But if we take a naive approach and tell the CNN to minimize the Euclidean distance between the input image and output image, then we will get blurry results. This happens because it is much "safer" for the L2 loss to predict the mean of the distribution, because this minimizes the mean pixel-wise error, but results in a blurry averaged image. [12]

Rather than specifying a carefully crafted model and loss for each Image-to-Image problem. It would be better if we could specify a goal like: *"make the output indistinguishable from reality"* While this may seem like a vague goal, this is exactly what Generative Adversarial Networks claim to do [3].

The main idea in the **Pix2Pix** paper was to use a Generative Adversarial Network to learn the loss function, rather than hand-coding a loss function such as Euclidean distance between target and input images.

3.2 Objective

The goal is to generate a target image based on some source image and we don't want the GAN to generate random images, so we use a variant of GANs called **Conditional GANs** [8]

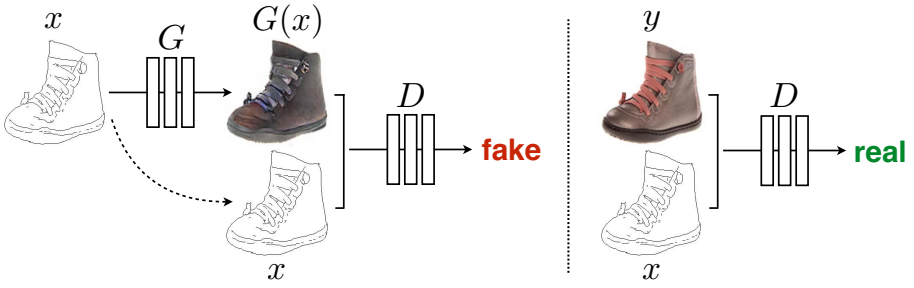


Figure 3.3: Conditional GAN used in Pix2Pix

As the name suggest Conditional GANs (figure 3.3) take an input to both the generator G and the discriminator D, for example, if we have a training pair $(\boldsymbol{c}, \boldsymbol{y})$, where we want to generate the item $\boldsymbol{y}$ based on the value of $\boldsymbol{c}$, we pass $\boldsymbol{c}$ to both G and D, then the loss function becomes:

$$\mathcal{L}_{\text{cGAN}}(G, D) = \mathbb{E}_{\boldsymbol{x},\boldsymbol{c}}[\log D(\boldsymbol{c}, \boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z},\boldsymbol{c}}[\log(1 - D(\boldsymbol{c}, G(\boldsymbol{c}, \boldsymbol{z})))] \tag{3.1}$$

In their experiments they found mixing both the adversarial loss and a more traditional loss such as L1 was helpful for training the GAN. The loss function for the discriminator remains unchanged but the generator now not only has to fool the discriminator but also minimize the L1 loss between the predicted and target image.

$$\mathcal{L}_{\text{L1}}(G, D) = \mathbb{E}_{\boldsymbol{y},\boldsymbol{c},\boldsymbol{z}}[\|y - G(\boldsymbol{c}, \boldsymbol{z})\|] \tag{3.2}$$

The final objective becomes for the generator becomes:

$$G^* = \arg\min_G \max_D \mathcal{L}_{\text{cGAN}}(G, D) + \lambda \mathcal{L}_{\text{L1}}(G, D) \tag{3.3}$$

For training the generator, instead of providing random noise as a direct input, they provide noise in the form of dropout.

3.3 Discriminator Architecture

Since we are already adding a traditional loss such as L1(or L2) which — although produce blurry results when used alone — are very useful for enforcing correctness at the low frequency. This is the main idea behind the **PatchGAN** [4] discriminator.

PatchGAN architecture restricts the discriminator to model only high frequency structure in the generated image. To model this, it is sufficient to restrict the attention to local image patches (figure 3.4). Therefore, rather than generating a single probability in range $[0, 1]$, the discriminator generates a $k \times k$ matrix of values in range $[0, 1]$ where each element in the matrix corresponds to a local patch(or receptive field) of the image.

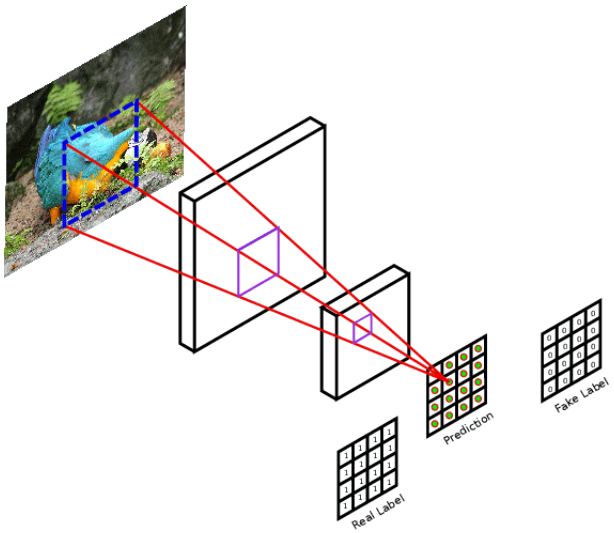


Figure 3.4: Visualization of **PatchGAN**

One advantage of **PatchGAN** is that the same architecture can extend to images of higher resolution with same number of parameters, and it would generate a similar, but larger, $k' \times k'$ matrix.

Inspired by **DCGAN** [9], the discriminator is implemented as a Deep CNN mostly build up of repetitions of modules of form: Convolution - BatchNorm - ReLU, where we down-sample until we reach the required $k \times k$ matrix, which we then pass through the logistic function to get the elements in range $[0, 1]$.

3.4 Generator Architecture

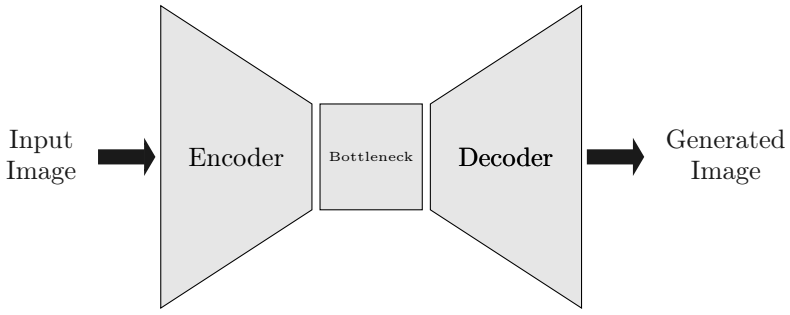


Figure 3.5: Encoder-Decoder Architecture

One of the obvious choices for a generator architecture for Image-to-Image translation, is the encoder-decoder network (figure 3.5). In this kind of network, the input image is passed through a series of layers which down-sample the resolution but increases the number of channels until we reach an image of size 1×1, after that the process is reversed until we reach the resolution of the original image. Here also we use modules of the form: Convolution - BatchNorm - LeakyReLU for encoder, for decoder we use: TransposeConvolution - BatchNorm - ReLU.

For many interesting image-translation problems, there is a lot of low-level information shared between the input and output image. For example, in the case of colorization, the input and output share the location of edges. To preserve this information, we add skip connections to create a "U-net" [10] like architecture.

This network is similar to Encoder-Decoder network , but we add some extra skip-connections between layer i in the encoder and layer $n - i$ in the decoder.

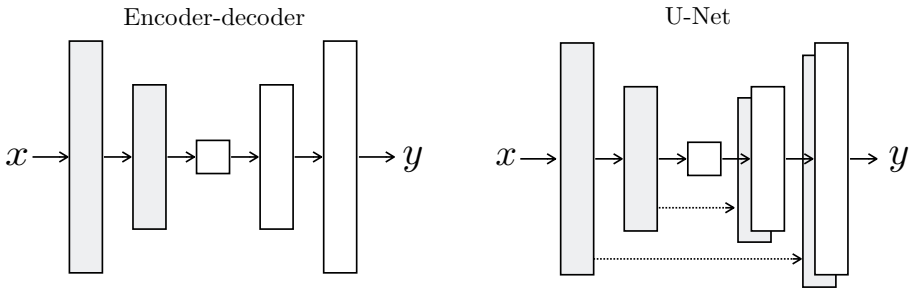


Figure 3.6: Difference between Encoder-Decoder and U-Net architectures.

3.5 Experiments

To demonstrate the versatility of the **Pix2Pix** approach, I applied the model on three different datasets with seemingly very different problem setting.

The datasets used are: **Satellite2Map** [4], **Edges2Shoes** [4], **Anime Sketch Colorization** (link). All datasets were trained on a single GPU and took ≈ 1 day to train.

I used the **Adam** Optimizer [6] with hyperparameters $\beta_1 = 0.5$ and $\beta_2 = 0.999$. Learning rate was set to 0.0002, and λ used was 100.

3.5.1 Satellite2Map

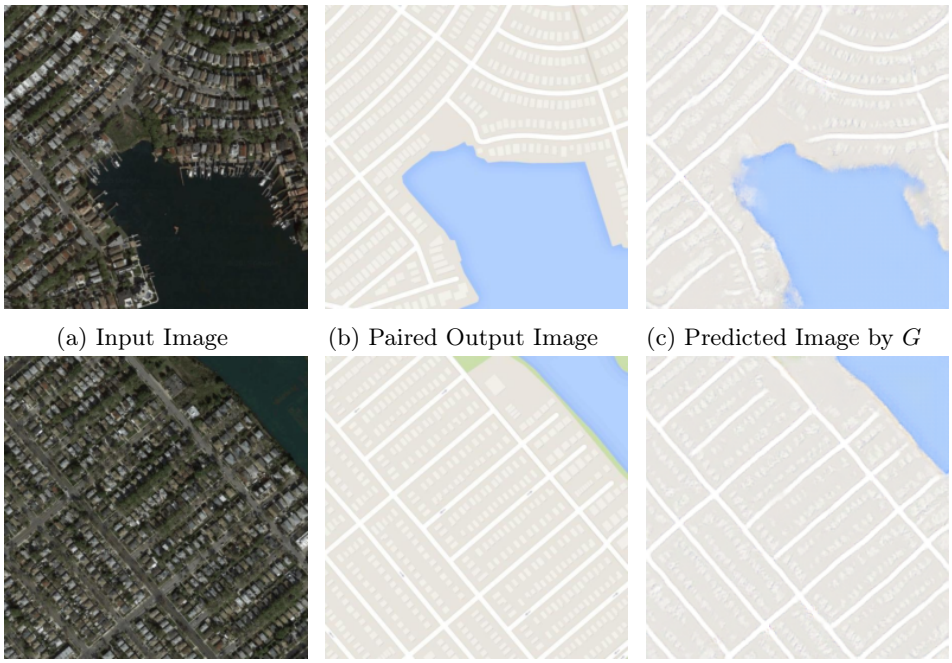
This dataset [4] consists of paired images of aerial photography and their corresponding map view scraped using google maps. There are 1096 training images of size 256×256. The model was trained for 150 epochs with batch size 1.

(a) Input Image (b) Paired Output Image (c) Predicted Image by G

(d) Input Image (e) Paired Output Image (f) Predicted Image by G

(g) Input Image (h) Paired Output Image (i) Predicted Image by G

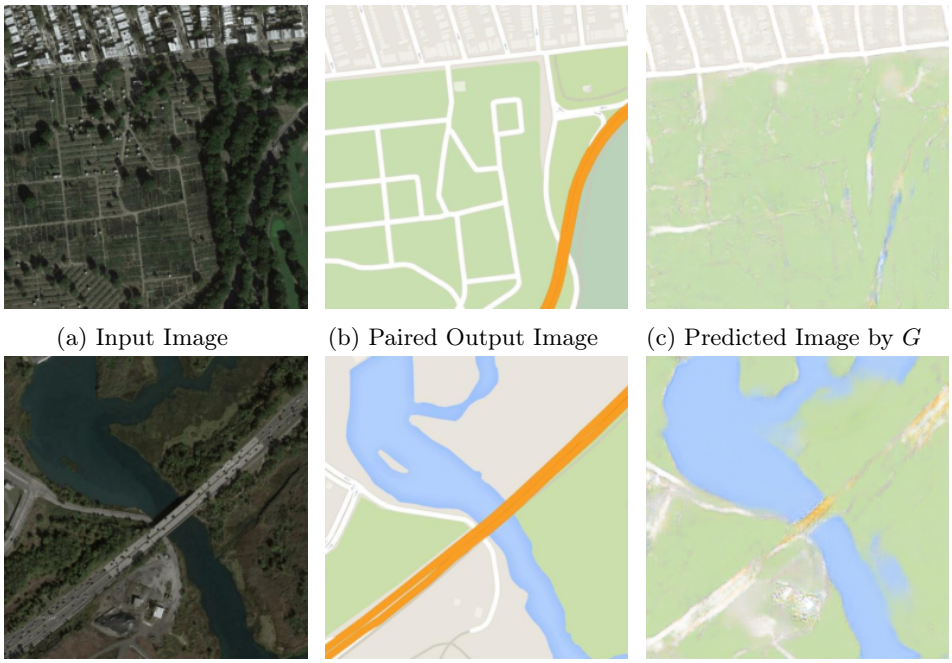
Figure 3.7: Example output on Satellite2Map dataset



(a) Input Image (b) Paired Output Image (c) Predicted Image by G

(d) Input Image (e) Paired Output Image (f) Predicted Image by G

Figure 3.8: Example output on Satellite2Map dataset



(a) Input Image (b) Paired Output Image (c) Predicted Image by G

(d) Input Image (e) Paired Output Image (f) Predicted Image by G

Figure 3.9: Example bad cases on Satellite2Map dataset

3.5.2 Edges2Shoes

This dataset [4] consists of paired images of shoe sketch image and the corresponding actual image (created using edge detection). There are 50k training images of size 256×256. The model was trained for 30 epochs with batch size 1.

(a) Input Image (b) Paired Output Image (c) Predicted Image by G

(d) Input Image (e) Paired Output Image (f) Predicted Image by G

Figure 3.10: Example output on Edges2Shoes dataset

(a) Input Image (b) Paired Output Image (c) Predicted Image by G

(d) Input Image (e) Paired Output Image (f) Predicted Image by G

(g) Input Image (h) Paired Output Image (i) Predicted Image by G

Figure 3.11: Example output on Edges2Shoes dataset

3.5.3 Anime Sketch Colorization

This dataset (link) consists of paired images of anime line sketch images and the corresponding colored image. There are 14k training images of size 256×256. The model was trained for 75 epochs with batch size 1.

(a) Input Image (b) Paired Output Image (c) Predicted Image by G

(d) Input Image (e) Paired Output Image (f) Predicted Image by G

(g) Input Image (h) Paired Output Image (i) Predicted Image by G

Figure 3.12: Example output on Anime Sketch Colorization dataset

(a) Input Image (b) Paired Output Image (c) Predicted Image by G

Figure 3.13: Example output on Anime Sketch Colorization dataset

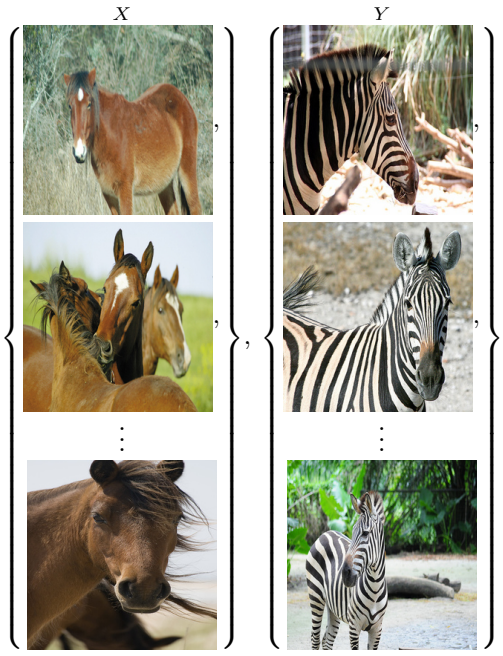
(a) Input Image (b) Paired Output Image (c) Predicted Image by G

Figure 3.14: Example bad case on Anime Sketch Colorization dataset

Chapter 4

Unsupervised Image-to-Image Translation

In unsupervised Image-to-Image translation there does not exist "paired" examples of input-ouput images, but only a "domain" (set) of images X and Y and thus, usual models cannot be applied.



4.1 CycleGAN

In the previous chapter we saw how we can utilize Conditional Generative Adversarial Networks [8] to tackle the supervised problem. Surprisingly, there is a way to extend the previous approach to work in an unsupervised setting!

CycleGAN [13] is a new model which allows for approaching the Image-to-Image translation problem from an unsupervised learning point of view. It is built using two GANs, capturing both transformations between domains as $G : X \to Y$ and also $F : Y \to X$.

The main idea behind this model is to add a cycle-consistency loss in addition to an adversarial loss. That is, if $K \in X$ and $K' \in Y$, then if we apply G and F sequentially on K and vice-versa on K', then we should get the same image back:

$$K \approx F(G(K)) \text{ and } K' \approx G(F(K'))$$

4.2 Objective

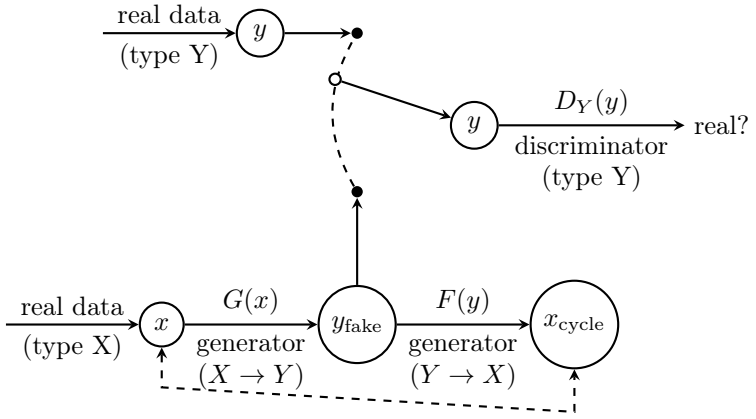


Figure 4.1: CycleGAN Visual Representation

The objective for the full CycleGAN model is the combination of adversarial loss for the two GANs:

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

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

And an additional cycle-consistency loss:

$$\begin{aligned}\mathcal{L}_{\text{cyc}}(G, F) = &\mathbb{E}_{x \sim p_{\text{data}}(x)}\|F(G(x)) - x\|_1 \\ &+ \mathbb{E}_{y \sim p_{\text{data}}(y)}\|G(F(y)) - y\|_1\end{aligned}$$

The final objective becomes:

$$\begin{aligned} G^*, F^* &= \arg\min_{G,F} \max_{D_X, D_Y} \mathcal{L}(G, F, D_X, D_Y) \\ &= \arg\min_{G,F} \max_{D_X, D_Y} \mathcal{L}_{\text{GAN}}(G, D_Y) + \mathcal{L}_{\text{GAN}}(F, D_X) + \lambda \mathcal{L}_{\text{cyc}}(G, F) \end{aligned}$$

4.3 Discriminator Architecture

The discriminator architecture is mostly the same as in the **Pix2Pix** model, but there are a few changes:

- We replace BatchNorm with InstanceNorm [11]
- We replace the negative log-likelihood objective by a least-squares loss, as this loss was found to be more stable during training and generate higher quality images [7].

That is, we now train G to minimize:

$$\mathbb{E}_{x \sim p_{\text{data}}(x)}[(D_Y(G(x)) - 1)^2]$$

And we train D_Y to minimize:

$$\mathbb{E}_{y \sim p_{\text{data}}(y)}[(D_Y(y) - 1)^2] + \mathbb{E}_{x \sim p_{\text{data}}(x)}[D_Y(G(x))^2]$$

The objective is similarly replaced for F and D_X.

4.4 Generator Architecture

The generator has a completely new architecture inspired networks presented in Perceptual Loss paper [5].

Similarly to the previous generator, we build the model in modules of Convolution - InstanceNorm - ReLU, but instead of the usual Encoder-Decoder network we have a network consisting of:

- 3 Convolution - InstanceNorm - ReLU Blocks.
- Several residual blocks (9 in the case of 256×256 images).
- 2 fractionally-strided convolution blocks to sample up.
- And finally a convolution block to map to RGB (or any variable number of channels).

4.5 Experiments

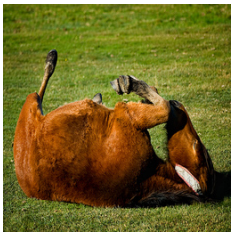
CycleGAN is a considerably heavier model than pix2pix and consists of atleast twice as many parameters, but, there is also additional complexity since the two GANs are in some sense learning from each-other.

The only dataset used for this model is the **Horse2Zebra** [13] dataset. It was trained on a single GPU and took $\approx$ 1 day to train.

I used the **Adam** Optimizer [6] with hyperparameters $\beta_1 = 0.5$ and $\beta_2 = 0.999$. Learning rate was set to 0.0002, and λ used was 10.

4.5.1 Horse2Zebra

This dataset [13] consists of two unpaired sets of images of horses and zebras, extracted from the ImageNet dataset [1]. There are 1067 images of horses and 1334 images of zebras, all of size 256×256. The model was trained for 10 epochs with batch size 1.



(a) Input Image

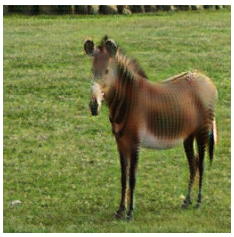
(b) Predicted Image

(c) Input Image

(d) Predicted Image

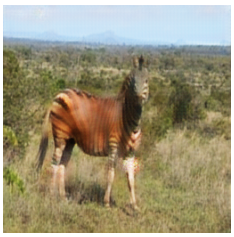
Figure 4.2: Horse $\rightarrow$ Zebra Examples

(a) Input Image



(b) Predicted Image

(c) Input Image



(d) Predicted Image

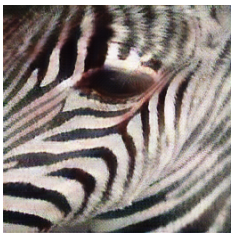
Figure 4.3: Zebra → Horse Examples

(a) Input Image

(b) Predicted Image

Figure 4.4: Horse → Zebra Bad Case

(a) Input Image



(b) Predicted Image

Figure 4.5: Zebra → Horse Bad Case

Appendix A

PyTorch Implementation

A.1 constants.py

```python
import torch
import torchvision.transforms as transforms

torch.backends.cudnn.benchmark = True
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
NUM_WORKERS = 2
LOAD_CHECKPOINT = True
SAVE_CHECKPOINT = False

NUM_EPOCHS = 200
BATCH_SIZE = 1

LEARNING_RATE = 0.0002
L1_LAMBDA = 100
L1_LAMBDA_CYCLE = 10
L1_LAMBDA_IDENTITY = 0
BETAS = (0.5, 0.999)

BASE_TRANSFORMATIONS = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
])
```

A.2 datasets.py

```python
import constants
import os
from PIL import Image
from torch.utils.data import Dataset

```

```
class SideBySideDataset(Dataset):
    def __init__(self, root_dir, output = 'right'):
        self.root_dir   = root_dir
        self.list_files = os.listdir(self.root_dir)
        self.output = output

    def __len__(self):
        return len(self.list_files)

    def __getitem__(self, index):
        img_file = self.list_files[index]
        img_path = os.path.join(self.root_dir, img_file)
        image = Image.open(img_path).convert('RGB')

        w,h = image.size

        img1 = image.crop((0, 0, w//2, h))
        img2 = image.crop((w//2, 0, w, h))

        img1 = constants.BASE_TRANSFORMATIONS(img1)
        img2 = constants.BASE_TRANSFORMATIONS(img2)

        if self.output == 'right':
            input_image, output_image = img1, img2
        elif self.output == 'left':
            input_image, output_image = img2, img1

        return input_image, output_image

class UnpairedDataset(Dataset):
    def __init__(self, root_dir_X, root_dir_Y):
        self.root_dir_X   = root_dir_X
        self.list_files_X = os.listdir(self.root_dir_X)

        self.root_dir_Y   = root_dir_Y
        self.list_files_Y = os.listdir(self.root_dir_Y)

    def __len__(self):
        return max(len(self.list_files_X), len(self.list_files_Y))

    def __getitem__(self, index):
        img_file_X = self.list_files_X[index % len(self.
    list_files_X)]
        img_path_X = os.path.join(self.root_dir_X, img_file_X)
        image_X = Image.open(img_path_X).convert('RGB')

        img_file_Y = self.list_files_Y[index % len(self.
    list_files_Y)]
        img_path_Y = os.path.join(self.root_dir_Y, img_file_Y)
        image_Y = Image.open(img_path_Y).convert('RGB')

        image_X = constants.BASE_TRANSFORMATIONS(image_X)
        image_Y = constants.BASE_TRANSFORMATIONS(image_Y)

        return image_X, image_Y
```

A.3 networks.py

```
import torch
import torch.nn as nn

class ConvBlock(nn.Module):
    def __init__(self, in_c, out_c, kind, act, drop=None, **kwargs)
    :
        super().__init__()
        Conv = nn.Conv2d if kind=='down' else nn.ConvTranspose2d

        self.block = [Conv(in_c, out_c, **kwargs)]
        self.block.append(
            nn.InstanceNorm2d(out_c))

        if act == 'relu':
            self.block.append(nn.ReLU(True))
        elif act == 'leaky':
            self.block.append(nn.LeakyReLU(0.2, True))
        elif act != 'none':
            raise Exception(f'Invalid Activation')

        if drop is not None:
            self.block.append(nn.Dropout(drop))

        self.block = nn.Sequential(*self.block)

    def forward(self, X):
        return self.block(X)

class EncoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=2):
        super().__init__()
        self.block = ConvBlock(
            in_channels, out_channels, kind='down', act='leaky',
            kernel_size=4,  stride=stride, padding=1, bias=False,
    padding_mode='reflect')

    def forward(self, X):
        return self.block(X)

class DecoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=2,
    dropout_rate=None):
        super().__init__()
        self.block = ConvBlock(
            in_channels, out_channels, kind='up', act='relu', drop
    =0.5,
            kernel_size=4,  stride=stride, padding=1, bias=False)

    def forward(self, X):
        return self.block(X)

class PatchDiscriminator(nn.Module):
    def __init__(self, net_in_channels, conditional=True, filters
    =[64, 128, 256, 512]):
        super().__init__()
```

```
        # Notes:
        # - For conditional, We get 2 images stacked on last axis
        #   so first in_channels = in_channels*2

        self.conditional = conditional
        net_in_channels = net_in_channels*2 if conditional else net_in_channels
        initial = nn.Sequential(
            nn.Conv2d(net_in_channels, filters[0], 4, 2, padding=1, padding_mode='reflect'),
            nn.LeakyReLU(0.2, inplace=True)
        )

        layers = nn.ModuleList([initial])
        in_channels = filters[0]

        for out_channels in filters[1:]:
            layers.append(
                EncoderBlock(in_channels, out_channels, stride=(1 if out_channels == filters[-1] else 2))
            )
            in_channels = out_channels

        layers.append(
            nn.Conv2d(
                in_channels, out_channels=1, kernel_size=4,
                stride=1, padding=1, padding_mode='reflect'
            )
        )

        layers.append(nn.Sigmoid())

        self.model = nn.Sequential(*layers)

    def forward(self, X, Y=None):
        if self.conditional:
            XY = torch.cat([X, Y], dim = 1)
            return self.model(XY)
        else:
            return self.model(X)

class UNetGenerator(nn.Module):
    def __init__(
            self,
            net_in_channels,
            net_out_channels,
            encoder_filters = [64, 128, 256, 512, 512, 512, 512, 512],
            decoder_filters = [512, 512, 512, 512, 256, 128, 64]):
        super().__init__()

        # Encoder
        self.encoder_layers = nn.ModuleList([nn.Sequential(
            nn.Conv2d(net_in_channels, encoder_filters[0], 4, 2, 1, padding_mode="reflect"),
            nn.LeakyReLU(0.2),
        )])
```

```
103
104         for i in range(len(encoder_filters)-2):
105             self.encoder_layers.append(
106                 EncoderBlock(encoder_filters[i], encoder_filters[i
    +1])
107             )
108
109         # Note: See Errata
110         self.encoder_layers.append(nn.Sequential(
111             nn.Conv2d(encoder_filters[-2], encoder_filters[-1], 4,
    2, 1),
112             nn.ReLU()
113         ))
114
115         # Decoder
116         self.decoder_layers = nn.ModuleList([DecoderBlock(
    encoder_filters[-1], decoder_filters[0])])
117
118         for i in range(len(decoder_filters)-1):
119             self.decoder_layers.append(DecoderBlock(2*
    decoder_filters[i], decoder_filters[i+1]))
120
121         self.decoder_layers.append(nn.Sequential(
122             nn.ConvTranspose2d(2*decoder_filters[-1],
    net_out_channels, 4,2,1),
123             nn.Tanh()
124         ))
125
126     def forward(self, X):
127         enc_activations = [X]
128         for enc in self.encoder_layers:
129             enc_activations.append(enc(enc_activations[-1]))
130
131         activation = self.decoder_layers[0](enc_activations[-1])
132
133         for i in range(1, len(self.decoder_layers)):
134             activation = torch.cat([activation, enc_activations[-(i
    +1)]], dim = 1)
135             activation = self.decoder_layers[i](activation)
136
137         return activation
138
139 class ResNetBlock(nn.Module):
140     def __init__(self, channels):
141         super().__init__()
142         self.block = nn.Sequential(
143             ConvBlock(channels, channels, kind='down', act='relu',
    kernel_size=3, padding=1),
144             ConvBlock(channels, channels, kind='down', act='none',
    kernel_size=3, padding=1),
145         )
146
147     def forward(self, x):
148         return x + self.block(x)
149
150 class ResNetGenerator(nn.Module):
```

```
    def __init__(self, net_in_channels, net_out_channels, filters =
     [64, 128, 256], num_residuals=9):
        super().__init__()

        self.encoder_layers = nn.ModuleList()
        self.encoder_layers.append(nn.Sequential(
            nn.Conv2d(net_in_channels, filters[0], kernel_size=7,
    stride=1, padding=3, padding_mode="reflect"),
            nn.InstanceNorm2d(filters[0]),
            nn.ReLU(inplace=True),
        ))

        for i in range(len(filters)-1):
            self.encoder_layers.append(ConvBlock(
                filters[i], filters[i+1], kind='down', act='relu',
    kernel_size=3, stride=2, padding=1))

        self.residual_layers = nn.Sequential(*[ResNetBlock(filters
    [-1]) for _ in range(num_residuals)])

        self.decoder_layers = nn.ModuleList()
        for i in range(len(filters)-1):
            self.decoder_layers.append(ConvBlock(
                filters[-(i+1)], filters[-(i+2)], kind='up', act='
    relu', kernel_size=3, stride=2, padding=1, output_padding=1))

        self.decoder_layers.append(nn.Conv2d(
            filters[0], net_out_channels, kernel_size=7, stride=1,
    padding=3, padding_mode="reflect"))

        self.decoder_layers.append(nn.Tanh())

    def forward(self, X):
        for layer in self.encoder_layers:
            X = layer(X)
        for layer in self.residual_layers:
            X = layer(X)
        for layer in self.decoder_layers:
            X = layer(X)

        return X
```

A.4 models.py

```
import os
from networks import ResNetGenerator

from pix2pix import Pix2Pix
from cyclegan import CycleGAN
from datasets import SideBySideDataset, UnpairedDataset

def satellite_to_map_model():
    folder = os.path.abspath('data/satellite_to_map')
    train_dataset = SideBySideDataset(os.path.join(folder, 'train')
    , output='right')
```

```
    test_dataset  = SideBySideDataset(os.path.join(folder, 'val'),
    output='right')

    model = Pix2Pix(
        train_dataset,
        test_dataset,
        os.path.join(folder, 'model.pth.tar'),
        os.path.join(folder, 'example_outputs'),
        num_examples=3
    )

    # model.train()
    model.save_examples('final', 200)

def segmentation_to_facade_model():
    folder = os.path.abspath('data/segmentation_to_facade')
    train_dataset = SideBySideDataset(os.path.join(folder, 'train')
    , output='left')
    test_dataset  = SideBySideDataset(os.path.join(folder, 'test'),
     output='left')

    model = Pix2Pix(
        train_dataset,
        test_dataset,
        os.path.join(folder, 'model.pth.tar'),
        os.path.join(folder, 'example_outputs'),
        num_examples=3,
    )

    # model.train()
    model.save_examples('final', 200)

def horse_zebra():
    folder = os.path.abspath('data/horse_zebra')
    train_dataset = UnpairedDataset(
        root_dir_X = os.path.join(folder, 'trainA'),
        root_dir_Y = os.path.join(folder, 'trainB')
    )
    test_dataset = UnpairedDataset(
        root_dir_X = os.path.join(folder, 'testA'),
        root_dir_Y = os.path.join(folder, 'testB')
    )

    model = CycleGAN(
        train_dataset,
        test_dataset,
        os.path.join(folder, 'model.pth.tar'),
        os.path.join(folder, 'example_outputs'),
        num_examples=3,
    )

    # model.train()

    model.save_examples('final', 200)

def apple_orange():
    folder = os.path.abspath('data/apple_orange')
```

```
    train_dataset = UnpairedDataset(
        root_dir_X = os.path.join(folder, 'trainA'),
        root_dir_Y = os.path.join(folder, 'trainB')
    )
    test_dataset = UnpairedDataset(
        root_dir_X = os.path.join(folder, 'testA'),
        root_dir_Y = os.path.join(folder, 'testB')
    )

    model = CycleGAN(
        train_dataset,
        test_dataset,
        os.path.join(folder, 'model.pth.tar'),
        os.path.join(folder, 'example_outputs'),
        num_examples=3,
    )

    model.train()

def ukiyoe():
    folder = os.path.abspath('data/ukiyoe2photo')
    train_dataset = UnpairedDataset(
        root_dir_X = os.path.join(folder, 'trainA'),
        root_dir_Y = os.path.join(folder, 'trainB')
    )
    test_dataset = UnpairedDataset(
        root_dir_X = os.path.join(folder, 'testA'),
        root_dir_Y = os.path.join(folder, 'testB')
    )

    model = CycleGAN(
        train_dataset,
        test_dataset,
        os.path.join(folder, 'model.pth.tar'),
        os.path.join(folder, 'example_outputs'),
        num_examples=3,
    )

    model.train()


def tester(p, out):
    folder = os.path.abspath(p)
    train_dataset = SideBySideDataset(os.path.join(folder, 'test'),
     output=out)
    test_dataset  = SideBySideDataset(os.path.join(folder, 'test'),
     output=out)

    model = Pix2Pix(
        train_dataset,
        test_dataset,
        os.path.join(folder, 'model.pth.tar'),
        os.path.join(folder, 'example_outputs'),
        num_examples=3,
    )

    # model.train()
```

```
    model.save_examples('final', 200)

if __name__ == '__main__':
    # satellite_to_map_model()
    # segmentation_to_facade_model()
    # horse_zebra()
    # ukiyoe()
    tester('data/shoes', 'right')
    # apple_orange()
```

A.5 pix2pix.py

```
import constants
from networks import ResNetGenerator, UNetGenerator,
    PatchDiscriminator

import os

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from tqdm import tqdm
from torchvision.utils import save_image

class Pix2Pix:
    def __init__(
            self,
            train_dataset,
            test_dataset,
            checkpoint_filename,
            examples_folder,
            discriminator = PatchDiscriminator,
            generator = ResNetGenerator,
            loss_type = 'mse',
            num_examples = 1
        ):
        self.train_dataset = train_dataset
        self.test_dataset  = test_dataset
        self.checkpoint_filename = checkpoint_filename
        self.examples_folder = examples_folder
        self.num_examples = num_examples

        self.train_loader  = DataLoader(
            self.train_dataset,
            batch_size = constants.BATCH_SIZE,
            shuffle = True,
            num_workers = constants.NUM_WORKERS,
        )
        self.test_loader = DataLoader(test_dataset, batch_size=1,
    shuffle=False)

        self.D = discriminator(net_in_channels=3).to(constants.
    DEVICE)
        self.G = generator(net_in_channels=3, net_out_channels=3).
    to(constants.DEVICE)
```

```
41
42          self.opt_D = optim.Adam(
43              self.D.parameters(), lr=constants.LEARNING_RATE, betas=
    constants.BETAS)
44          self.opt_G = optim.Adam(
45              self.G.parameters(), lr=constants.LEARNING_RATE, betas=
    constants.BETAS)
46
47          self.D_scaler = torch.cuda.amp.GradScaler()
48          self.G_scaler = torch.cuda.amp.GradScaler()
49
50          self.loss = nn.MSELoss() if loss_type == 'mse' else nn.
    BCELoss()
51          self.l1_loss = nn.L1Loss()
52
53          if constants.LOAD_CHECKPOINT:
54              self.load_checkpoint()
55
56      def train(self):
57          for epoch in range(constants.NUM_EPOCHS):
58              print(f'Epoch {epoch}')
59              self.train_one_epoch()
60
61              if constants.SAVE_CHECKPOINT and epoch % 5 == 0:
62                  self.save_checkpoint()
63
64              self.save_examples(epoch)
65
66      def train_one_epoch(self):
67          pbar = tqdm(self.train_loader, leave=True)
68
69          for idx, (X, Y) in enumerate(pbar):
70              X = X.to(constants.DEVICE)
71              Y = Y.to(constants.DEVICE)
72
73              with torch.cuda.amp.autocast():
74                  D_real = self.D(X, Y)
75                  D_real_loss = self.loss(D_real, torch.ones_like(
    D_real))
76
77                  Y_fake = self.G(X)
78                  D_fake = self.D(X, Y_fake.detach())
79                  D_fake_loss = self.loss(D_fake, torch.zeros_like(
    D_fake))
80
81                  D_loss = (D_real_loss + D_fake_loss) / 2
82
83              self.opt_D.zero_grad()
84              self.D_scaler.scale(D_loss).backward()
85              self.D_scaler.step(self.opt_D)
86              self.D_scaler.update()
87
88              with torch.cuda.amp.autocast():
89                  D_fake = self.D(X, Y_fake)
90                  G_fake_loss = self.loss(D_fake, torch.ones_like(
    D_fake))
91                  L1 = self.l1_loss(Y_fake, Y) * constants.L1_LAMBDA
```

```
                G_loss = G_fake_loss + L1

            self.opt_G.zero_grad()
            self.G_scaler.scale(G_loss).backward()
            self.G_scaler.step(self.opt_G)
            self.G_scaler.update()

            if idx % 10 == 0:
                pbar.set_postfix(
                    D_real=torch.sigmoid(D_real).mean().item(),
                    D_fake=torch.sigmoid(D_fake).mean().item(),
                )

    def save_examples(self, epoch, n=None):
        folder = self.examples_folder
        i = iter(self.test_loader)
        for n in range(self.num_examples if n is None else n):
            X, Y = next(i)
            X, Y = X.to(constants.DEVICE), Y.to(constants.DEVICE)

            self.G.eval()
            with torch.no_grad():
                Z = self.G(X)
                save_image(Z*0.5 + 0.5, os.path.join(folder, f"{epoch}_{n}_Y_fake.png"))
                save_image(X*0.5 + 0.5, os.path.join(folder, f"{epoch}_{n}_X.png"))
                save_image(Y*0.5 + 0.5, os.path.join(folder, f"{epoch}_{n}_Y.png"))
            self.G.train()


    def save_checkpoint(self):
        filename = self.checkpoint_filename
        checkpoint = {
            "G_state_dict":     self.G.state_dict(),
            "opt_G_state_dict": self.opt_G.state_dict(),

            "D_state_dict":     self.D.state_dict(),
            "opt_D_state_dict": self.opt_D.state_dict(),
        }
        torch.save(checkpoint, filename)
        print(">> Checkpoint Saved")


    def load_checkpoint(self):
        checkpoint = torch.load(self.checkpoint_filename, map_location=constants.DEVICE)
        self.G.load_state_dict(checkpoint['G_state_dict'])
        self.opt_G.load_state_dict(checkpoint['opt_G_state_dict'])

        self.D.load_state_dict(checkpoint['D_state_dict'])
        self.opt_D.load_state_dict(checkpoint['opt_D_state_dict'])

        for optimizer in (self.opt_G, self.opt_D):
            for param_group in optimizer.param_groups:
                param_group["lr"] = constants.LEARNING_RATE
```

```
145         print('>> Checkpoint Loaded')
```

A.6 cyclegan.py

```
1  import constants
2  from networks import ResNetGenerator, PatchDiscriminator
3
4  import os
5
6  import torch
7  import torch.nn as nn
8  import torch.optim as optim
9  from torch.utils.data import DataLoader
10 from tqdm import tqdm
11 from torchvision.utils import save_image
12
13 class CycleGAN:
14     def __init__(
15             self,
16             train_dataset,
17             test_dataset,
18             checkpoint_filename,
19             examples_folder,
20             discriminator = PatchDiscriminator,
21             generator = ResNetGenerator,
22             loss_type = 'mse',
23             num_examples = 1):
24
25         self.train_dataset = train_dataset
26         self.test_dataset  = test_dataset
27         self.checkpoint_filename = checkpoint_filename
28         self.examples_folder = examples_folder
29         self.num_examples = num_examples
30
31         self.train_loader  = DataLoader(
32             self.train_dataset,
33             batch_size = constants.BATCH_SIZE,
34             shuffle = True,
35             num_workers = constants.NUM_WORKERS)
36
37         self.test_loader = DataLoader(test_dataset, batch_size=1,
    shuffle=False)
38
39
40         # Discrimator of the domain X
41         self.D_X  = discriminator(net_in_channels=3, conditional=
    False).to(constants.DEVICE)
42
43         # Discrimator of the domain Y
44         self.D_Y  = discriminator(net_in_channels=3, conditional=
    False).to(constants.DEVICE)
45
46         # Generator from Y -> X
47         self.G_YX = generator(net_in_channels=3, net_out_channels
    =3).to(constants.DEVICE)
48
```

```
        # Generator from X -> Y
        self.G_XY = generator(net_in_channels=3, net_out_channels=3).to(constants.DEVICE)

        self.opt_D = optim.Adam(
            list(self.D_X.parameters()) + list(self.D_Y.parameters()),
            lr=constants.LEARNING_RATE, betas=constants.BETAS)

        self.opt_G = optim.Adam(
            list(self.G_YX.parameters()) + list(self.G_XY.parameters()),
            lr=constants.LEARNING_RATE, betas=constants.BETAS)

        # Float-16 Training
        self.D_scaler = torch.cuda.amp.GradScaler()
        self.G_scaler = torch.cuda.amp.GradScaler()

        self.loss = nn.MSELoss() if loss_type == 'mse' else nn.BCELoss()
        self.l1_loss = nn.L1Loss()

        if constants.LOAD_CHECKPOINT:
            self.load_checkpoint()

    def train(self):
        for epoch in range(constants.NUM_EPOCHS):
            print(f'Epoch {epoch}')
            self.train_one_epoch(epoch)

            if constants.SAVE_CHECKPOINT and epoch % 5 == 0:
                self.save_checkpoint()

            self.save_examples(epoch)

    def train_one_epoch(self, epoch):
        pbar = tqdm(self.train_loader, leave=True)

        for idx, (image_X, image_Y) in enumerate(pbar):
            image_X = image_X.to(constants.DEVICE)
            image_Y = image_Y.to(constants.DEVICE)

            # Train Discriminators
            with torch.cuda.amp.autocast():
                # D_X
                fake_X = self.G_YX(image_Y)

                D_X_real = self.D_X(image_X)
                D_X_fake = self.D_X(fake_X.detach())

                D_X_real_loss = self.loss(D_X_real, torch.ones_like(D_X_real))
                D_X_fake_loss = self.loss(D_X_fake, torch.zeros_like(D_X_fake))

                D_X_loss = D_X_real_loss + D_X_fake_loss

```

```
                # D_Y
                fake_Y = self.G_XY(image_X)

                D_Y_real = self.D_Y(image_Y)
                D_Y_fake = self.D_Y(fake_Y.detach())

                D_Y_real_loss = self.loss(D_Y_real, torch.ones_like
    (D_Y_real))
                D_Y_fake_loss = self.loss(D_Y_fake, torch.
    zeros_like(D_Y_fake))

                D_Y_loss = D_Y_real_loss + D_Y_fake_loss

                # Total Loss
                D_loss = (D_X_loss + D_Y_loss)/2

            self.opt_D.zero_grad()
            self.D_scaler.scale(D_loss).backward()
            self.D_scaler.step(self.opt_D)
            self.D_scaler.update()

            # Train Generators
            with torch.cuda.amp.autocast():
                # Adversarial Loss
                D_X_fake = self.D_X(fake_X)
                D_Y_fake = self.D_Y(fake_Y)

                G_YX_loss = self.loss(D_X_fake, torch.ones_like(
    D_X_fake))
                G_XY_loss = self.loss(D_Y_fake, torch.ones_like(
    D_Y_fake))

                # Cycle Loss
                cycle_X = self.G_YX(fake_Y)
                cycle_Y = self.G_XY(fake_X)

                cycle_X_loss = self.l1_loss(image_X, cycle_X)
                cycle_Y_loss = self.l1_loss(image_Y, cycle_Y)

                G_loss = (
                    G_XY_loss + G_YX_loss
                    + constants.L1_LAMBDA_CYCLE*(cycle_X_loss +
    cycle_Y_loss)
                )

                # Identity Loss
                if constants.L1_LAMBDA_IDENTITY > 0:
                    identitiy_loss_X = self.l1_loss(image_X, self.
    G_YX(image_X))
                    identitiy_loss_Y = self.l1_loss(image_Y, self.
    G_XY(image_Y))
                    G_loss += constants.L1_LAMBDA_IDENTITY*(
    identitiy_loss_X+identitiy_loss_Y)

            self.opt_G.zero_grad()
            self.G_scaler.scale(G_loss).backward()
            self.G_scaler.step(self.opt_G)
```

```
            self.G_scaler.update()

            if idx % 200 == 0:
                self.save_examples(f'{epoch}_{idx}')

            if idx % 500 == 0 and idx > 0:
                self.save_checkpoint()

            if idx % 10 == 0:
                pbar.set_postfix(
                    D_X_real=torch.sigmoid(D_X_real).mean().item(),
                    D_Y_real=torch.sigmoid(D_Y_real).mean().item(),
                    D_X_fake=torch.sigmoid(D_X_fake).mean().item(),
                    D_Y_fake=torch.sigmoid(D_Y_fake).mean().item(),
                )

    def save_examples(self, epoch, n = None):
        folder = self.examples_folder
        i = iter(self.test_loader)
        for n in range(self.num_examples if n is None else n):
            X, Y = next(i)
            X, Y = X.to(constants.DEVICE), Y.to(constants.DEVICE)

            self.G_XY.eval()
            self.G_YX.eval()
            with torch.no_grad():
                fake_Y = self.G_XY(X)
                fake_X = self.G_YX(Y)
                save_image(Y*0.5 + 0.5, os.path.join(folder, f"{
    epoch}_{n}_Y.png"))
                save_image(X*0.5 + 0.5, os.path.join(folder, f"{
    epoch}_{n}_X.png"))
                save_image(fake_Y*0.5 + 0.5, os.path.join(folder, f
    "{epoch}_{n}_Y_fake.png"))
                save_image(fake_X*0.5 + 0.5, os.path.join(folder, f
    "{epoch}_{n}_X_fake.png"))

            self.G_XY.train()
            self.G_YX.train()

    def save_checkpoint(self):
        filename = self.checkpoint_filename
        checkpoint = {
            "D_X_state_dict": self.D_X.state_dict(),
            "D_Y_state_dict": self.D_Y.state_dict(),
            "opt_D_state_dict": self.opt_D.state_dict(),

            "G_XY_state_dict":  self.G_XY.state_dict(),
            "G_YX_state_dict":  self.G_YX.state_dict(),
            "opt_G_state_dict": self.opt_G.state_dict(),
        }
        torch.save(checkpoint, filename)
        print(">> Checkpoint Saved")


    def load_checkpoint(self):
```

```
        checkpoint = torch.load(self.checkpoint_filename,
    map_location=constants.DEVICE)
        self.G_XY.load_state_dict(checkpoint['G_XY_state_dict'])
        self.G_YX.load_state_dict(checkpoint['G_YX_state_dict'])
        self.opt_G.load_state_dict(checkpoint['opt_G_state_dict'])

        self.D_X.load_state_dict(checkpoint['D_X_state_dict'])
        self.D_Y.load_state_dict(checkpoint['D_Y_state_dict'])
        self.opt_D.load_state_dict(checkpoint['opt_D_state_dict'])

        for optimizer in (self.opt_G, self.opt_D):
            for param_group in optimizer.param_groups:
                param_group["lr"] = constants.LEARNING_RATE
        print('>> Checkpoint Loaded')
```

A.7 models.py

```
import os
from networks import ResNetGenerator

from pix2pix import Pix2Pix
from cyclegan import CycleGAN
from datasets import SideBySideDataset, UnpairedDataset

def satellite_to_map_model():
    folder = os.path.abspath('data/satellite_to_map')
    train_dataset = SideBySideDataset(os.path.join(folder, 'train')
    , output='right')
    test_dataset  = SideBySideDataset(os.path.join(folder, 'val'),
    output='right')

    model = Pix2Pix(
        train_dataset,
        test_dataset,
        os.path.join(folder, 'model.pth.tar'),
        os.path.join(folder, 'example_outputs'),
        num_examples=3
    )

    # model.train()
    model.save_examples('final', 200)

def segmentation_to_facade_model():
    folder = os.path.abspath('data/segmentation_to_facade')
    train_dataset = SideBySideDataset(os.path.join(folder, 'train')
    , output='left')
    test_dataset  = SideBySideDataset(os.path.join(folder, 'test'),
     output='left')

    model = Pix2Pix(
        train_dataset,
        test_dataset,
        os.path.join(folder, 'model.pth.tar'),
        os.path.join(folder, 'example_outputs'),
        num_examples=3,
    )
```

```
36
37      # model.train()
38      model.save_examples('final', 200)
39
40  def horse_zebra():
41      folder = os.path.abspath('data/horse_zebra')
42      train_dataset = UnpairedDataset(
43          root_dir_X = os.path.join(folder, 'trainA'),
44          root_dir_Y = os.path.join(folder, 'trainB')
45      )
46      test_dataset = UnpairedDataset(
47          root_dir_X = os.path.join(folder, 'testA'),
48          root_dir_Y = os.path.join(folder, 'testB')
49      )
50
51      model = CycleGAN(
52          train_dataset,
53          test_dataset,
54          os.path.join(folder, 'model.pth.tar'),
55          os.path.join(folder, 'example_outputs'),
56          num_examples=3,
57      )
58
59      # model.train()
60
61      model.save_examples('final', 200)
62
63  def apple_orange():
64      folder = os.path.abspath('data/apple_orange')
65      train_dataset = UnpairedDataset(
66          root_dir_X = os.path.join(folder, 'trainA'),
67          root_dir_Y = os.path.join(folder, 'trainB')
68      )
69      test_dataset = UnpairedDataset(
70          root_dir_X = os.path.join(folder, 'testA'),
71          root_dir_Y = os.path.join(folder, 'testB')
72      )
73
74      model = CycleGAN(
75          train_dataset,
76          test_dataset,
77          os.path.join(folder, 'model.pth.tar'),
78          os.path.join(folder, 'example_outputs'),
79          num_examples=3,
80      )
81
82      model.train()
83
84  def ukiyoe():
85      folder = os.path.abspath('data/ukiyoe2photo')
86      train_dataset = UnpairedDataset(
87          root_dir_X = os.path.join(folder, 'trainA'),
88          root_dir_Y = os.path.join(folder, 'trainB')
89      )
90      test_dataset = UnpairedDataset(
91          root_dir_X = os.path.join(folder, 'testA'),
92          root_dir_Y = os.path.join(folder, 'testB')
```

```
    )

    model = CycleGAN(
        train_dataset,
        test_dataset,
        os.path.join(folder, 'model.pth.tar'),
        os.path.join(folder, 'example_outputs'),
        num_examples=3,
    )

    model.train()


def tester(p, out):
    folder = os.path.abspath(p)
    train_dataset = SideBySideDataset(os.path.join(folder, 'test'),
     output=out)
    test_dataset  = SideBySideDataset(os.path.join(folder, 'test'),
     output=out)

    model = Pix2Pix(
        train_dataset,
        test_dataset,
        os.path.join(folder, 'model.pth.tar'),
        os.path.join(folder, 'example_outputs'),
        num_examples=3,
    )

    # model.train()
    model.save_examples('final', 200)

if __name__ == '__main__':
    # satellite_to_map_model()
    # segmentation_to_facade_model()
    # horse_zebra()
    # ukiyoe()
    tester('data/shoes', 'right')
    # apple_orange()
```

Bibliography

[1] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[3] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[4] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017.

[5] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European conference on computer vision*, pages 694–711. Springer, 2016.

[6] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[7] Xudong Mao, Qing Li, Haoran Xie, Raymond YK Lau, Zhen Wang, and Stephen Paul Smolley. Least squares generative adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2794–2802, 2017.

[8] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[9] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[10] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Con-*

ference on Medical image computing and computer-assisted intervention, pages 234–241. Springer, 2015.

[11] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

[12] Richard Zhang, Phillip Isola, and Alexei A Efros. Colorful image colorization. In *European conference on computer vision*, pages 649–666. Springer, 2016.

[13] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.