Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Investigating Single Translation Function CycleGANs

Relatore:
Chiar.mo Prof.
Andrea Asperti

Presentata da:
Oleksandr Olmucci Poddubnyy

Sessione I
Anno Accademico 2017/2018

Contents

To my parents and my friends.

Introduction

With the advent of Deep Learning, exploiting neural networks with a large amount of layers, we have been able to find solutions to problems that do not have a known algorithmic solution. Among these problems there are many interesting computer vision and graphics tasks such as image recognition [14][19][10] (recognizing what a given image contains), image captioning [21][13] (producing a textual description of what an image contains), style transfer [6][15] (given two images, represent the first image with the graphical style of the second image) and many more. One interesting problem that got formulated recently is the problem of image-to-image translation, first defined in [12].
Image to image translation is a class of problems where the goal is to change the graphical representation of an image such that the semantics is preserved. We can conduct an analogy between the automated language translation task (for example a translation of a natural language sentence from French to English) and the image-to-image translation task, by noting that both of these tasks need to change the representation of a given concept. The language translation task simply needs to translate a concept expressed by words, into another sequence preserving the semantics, while the image-to-image translation task needs to translate a collection of pixels of an image into another collection of pixels.
With the introduction of the image-to-image translation task, there were proposed two different general-purpose techniques based on generative adversarial networks respectively using different kinds of training data, labeled and unlabeled. In machine learning, labeled training data means that for every training sample we know both its input and output values, while unlabeled training data doesn't contain any information about the output. The latter kind of training data is more available nowadays than the former one. In the context of image-to-image translation, labeled data is called paired data and unlabeled data is called unpaired data and the difference between the two is that with paired data we need multiple image representations of the same context, while with unpaired data we only need two sets of images. The main technique to solve the image-to-image translation task on paired data is called pix2pix [12], while a more interesting technique which uses unpaired data is called CycleGAN [22].
CycleGANs are a particular kind of neural networks that learn to generate images based on two domains (sets) of data. They include the word "Cycle" in their name due to

the fact that in order to learn to translate in a correct way, they use two translator networks between two domains of training data which capture the intuition that if we perform a translation towards a domain, then a translation back to the original domain should produce an output similar to the input data; more informally, in analogy with the automatic language translation where if we translate an input sentence from a language to another, and then translate it back, we should obtain a sentence which is similar to the input.

Even though CycleGANs are a powerful instrument, their original version is bound to be used with only two domains (or classes) of data at a time, making them unsuitable for tasks where we have potentially infinite domains, such as continuous season transfer or continuous version of finite class face aging [3].

This dissertation presents some attempts that were done in order to use the CycleGAN approach to perform image translations between more than two classes at a time: the goal was to obtain a continuous translation between two given classes.

In the first chapter we'll describe the theoretical background which will be used subsequently to understand the CycleGANs (some general details are out of the scope of this thesis, thus will be omitted).

The second chapter is about CycleGANs, we are going to describe the underlying ideas and mechanisms which make their training possible.

Finally the third chapter shows results of some experimentations that were obtained by modifying the existing CycleGAN's setup with the intention to allow a continuous translation between two given image classes using a single network for translation instead of two. Even though the final goal of learning to translate images between two domains in a continuous way wasn't reached, we were able to find another interesting result which allowed us to translate images with a quality close to CycleGAN using a single translation network.

Chapter 1

Theory

In this chapter we are going to describe the main building blocks of a CycleGAN. We'll emphasize the most on **Convolutional Neural Networks** (CNNs for short) which are networks that are nowadays used to all the image related tasks, **General Adversarial Networks** (GANs) which are a kind of generative model that is trained to generate realistic images and the problem of **Image-to-Image translation**. A brief part on how to use Tensorflow for simple deep learning related tasks is included as well.

1.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are Neural Networks that are used for image related tasks. They are different from the traditional fully-connected networks (also called Multilayer Perceptrons) due to the fact that these kinds of networks use particular kind of layers:

- Convolutional layers
- Deconvolutional layers
- Pooling layers

We omit the description of pooling layers as they are outside the scope of CycleGANs.

1.1.1 Convolutional Layers

Convolutional layers are the main component of Convolutional Neural Networks. They intuitively apply a set of filters to an image in order to produce a **feature map** of the image. The feature maps represent particular areas of interest of an image that the network considers important in order to perform classification.

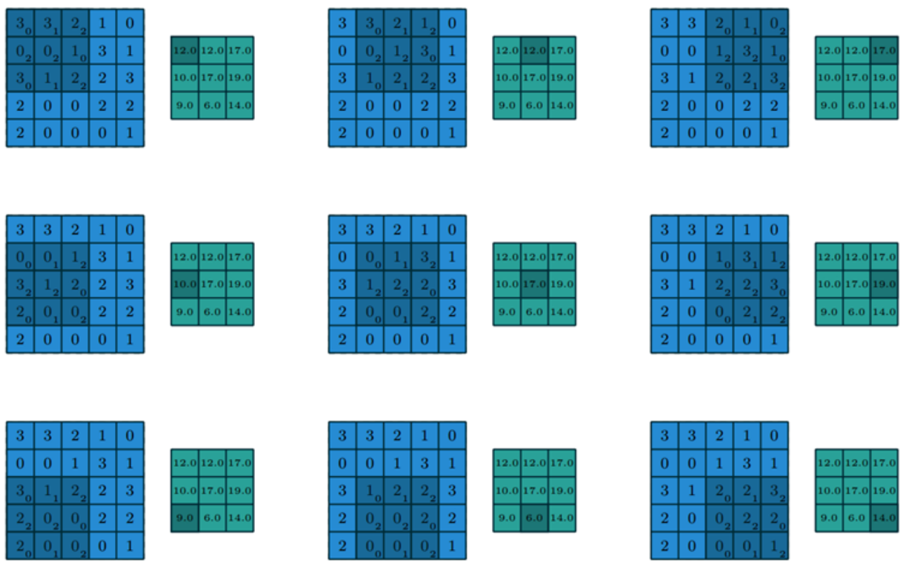


Figure 1.1: Convolution operation produces an output feature map (teal area) by "sliding" a kernel matrix (blue area) over an input image matrix (light blue area).

The convolution operation itself, applied by the layers between an input image and a learned kernel, is defined by equation

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

where I as an input image and K is a kernel, however most deep learning libraries implement the cross-correlation operation

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

due to optimization reasons [8], with the only difference between the two being the kernel flipping (however it is irrelevant in the context of neural networks, as the parameters are learned in the same way). A more intuitive explanation can be visualized in figure 1.1 besides the mathematical definition. A complete guide about the convolutions which includes better visualizations consisting of different cases can be found in [5]. All the images in this section have been borrowed form that article.
Lastly, convolutions allow a set of parameters that determine the output image's dimensionality, which are:

- **Number of filters**: this parameter contributes to the depth of the resulting image produced by the convolution. (i.e. if we apply a convolution with 32 filters, the resulting image will have 32 channels)

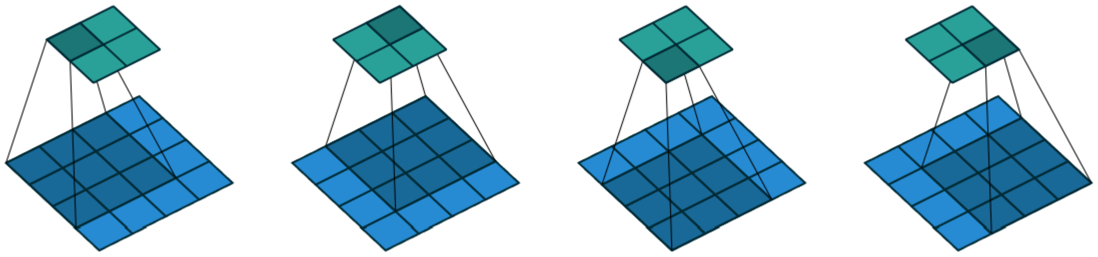


Figure 1.2: Convolution with a stride of 1×1, kernel size of 3×3 and no padding (also called valid padding).

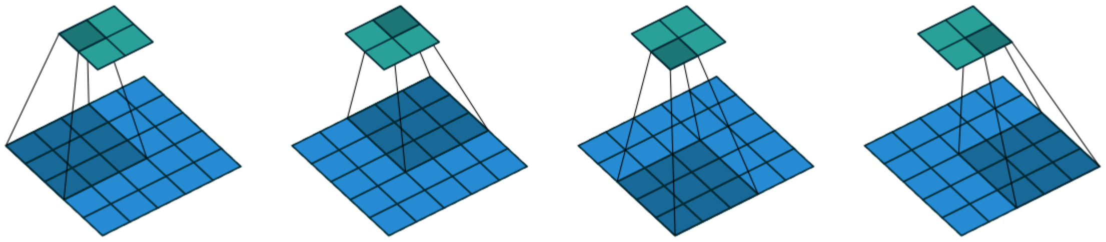


Figure 1.3: Convolution with a stride of 2×2, kernel size of 3×3 and no padding.

- **Kernel size**: determines the dimensions (only one dimension, as the kernel is usually a square) of the kernel which is slided over the input image.
- **Stride**: determines the step size used by the filter when moving over the input image. For example, in figure 1.2, unitary stride is used and we can see that the kernel is moved by one pixel at a time over the input image, while figure 1.3 shows a convolution with a stride of two, where the kernel is in fact moved by two pixels at a time.
- **Padding**: indicates the amount of padding that is added to the input image before performing the convolution, which changes its dimension from $w \times h$ to $(w + 2p) \times (h + 2p)$. While the padding can be arbitrary (as seen in figure 1.4), there are two particular kinds of padding which are interesting: half-padding (also known as same-padding) and full-padding. The former is performed by setting $p = \lfloor \frac{k}{2} \rfloor$ which allows the output image preserve the same dimensionality of the input image as seen in figure 1.5. The latter is performed by setting $p = k-1$ which allows the image to grow in dimension after the convolution has been performed, as we can see in figure 1.6.

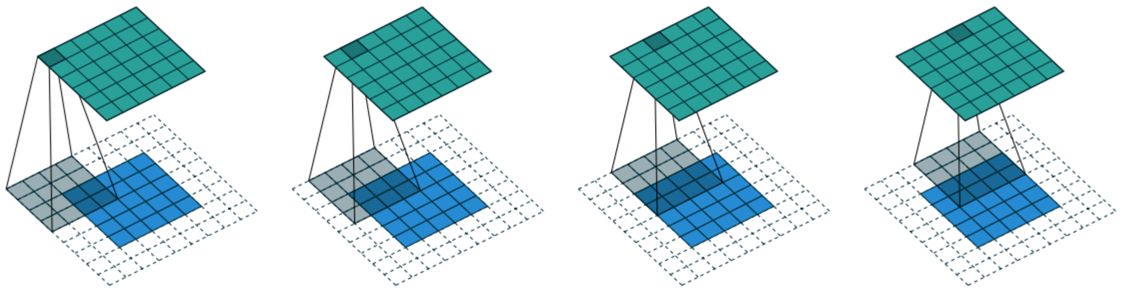


Figure 1.4: Convolution with a stride of 1×1, kernel size of 4×4 and an arbitrary padding of 2.

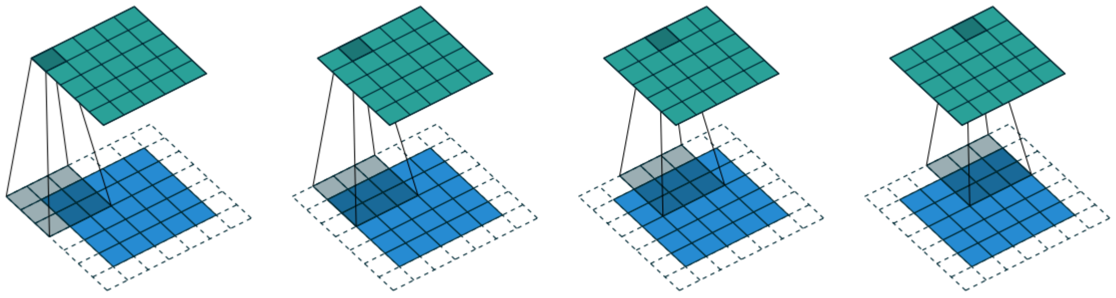


Figure 1.5: Convolution with a stride of 1×1, kernel size of 3×3 and a padding of 1 (half-padding), as we can observe, the produced output has the same dimensionality as the input.

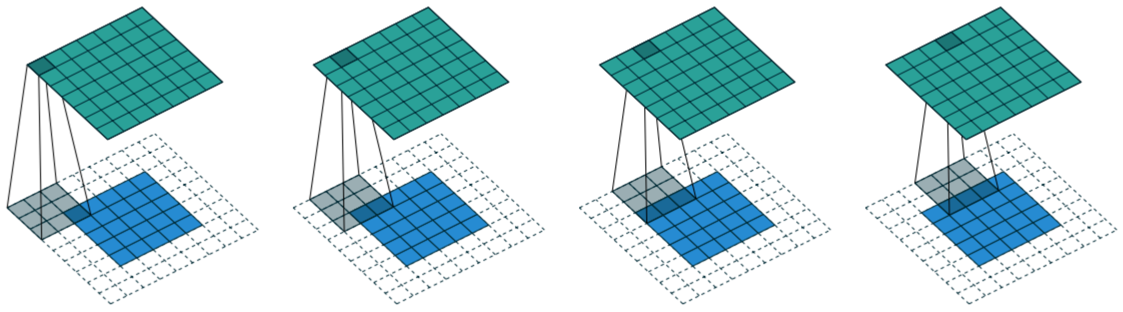


Figure 1.6: Convolution with a stride of 1×1, kernel size of 3×3 and a padding of 2 (full-adding), thus the output image dimensionality gets increased.

1.1.2 Deconvolutional Layers

Deconvolutional layers (also called Transposed Convolutions) allow to project a lower dimensionality input into a higher dimensionality output (up-sampling it) in an optimal way. We'll use a simple example to explain how these layers work.
Let us consider the following matrices x (input) and k (kernel).

$$x = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \qquad k = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$$

Assume that we want to perform a stride 1 and no padding convolution on them, we can do it by either using the formula defined previously or by performing a normal matrix multiplication between a flattened version of x matrix and an expanded version of k matrix. We expand the k matrix to produce a new $\hat{k}$ matrix as follows:

$$\hat{k} = \begin{bmatrix} w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 & 0 & 0 & 0 \\ 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 \\ 0 & 0 & 0 & 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} \end{bmatrix}$$

What we did there was to "highlight" the values covered by the kernel matrix on the input image, assigning the corresponding w_{ij} values to the items that were covered by kernel matrix, leaving uncovered values as 0, making the i'th row of $\hat{k}$ correspond to the i'th step of convolution. If k's dimensionality was 2×2 and x's dimensionality was 3×3 then $\hat{k}$'s dimensionality is 4×9 since we can only perform 4 steps in the assumed settings. Finally, to perform the convolution itself we multiply $\hat{k}$ by the flattened version of x. Since the multiplication is between a 4×9 matrix and a 9×1 matrix, we get a 4×1 dimensional matrix which gets reshaped into a 2×2 matrix, corresponding to the output feature map.
A transposed convolution is obtained in the same way, except that the matrix multiplication is performed on $\hat{k}^{\mathsf{T}}$ transposed matrix, mapping a lower dimensional matrix to a higher dimensional one.

1.1.3 Batch Normalization

Batch normalization [11] is a technique which can be used in order to reduce the internal covariate shift (change in the distribution of network's activations due to changes in network's parameters during training). By reducing the covariate shift the training process becomes more stable which allows to use higher learning rates and to be less careful about initialization as the output of each layer gets normalized.
The normalization is done by adding additional layers which are responsible of collecting

batch statistics during the training time that allow to re-normalize the batch mean and batch variance to learned values γ (learned variance) and β (learned mean). This means that normalization can be added as a part of the model's architecture.
As we can see in algorithm 1, batch normalization layer normalizes the previous layer's output by subtracting the batch mean $\mu_{\mathcal{B}}$ and diving by the batch standard deviation $\sigma_{\mathcal{B}}$, this makes so that the input data batch gets a mean of 0 and a variance of 1. After the normalization step, the batch is subject to a re-parametrization which gives it a mean of β and a variance of γ learned by the relative batch normalization layer. The re-parametrization step is needed to make sure that the transformation can represent an identity, because normalizing the input of a layer alone may change what that layer can represent (for example if we normalize the inputs of a sigmoid we might end up into the linear regime of the nonlinearity). In fact if $\beta = \mathbb{E}[\mathcal{B}]$ and $\gamma = \sqrt{\mathrm{Var}[\mathcal{B}]}$ then

$$\begin{aligned} y_i =& \gamma \hat{x}_i + \beta \\ =& \sqrt{\mathrm{Var}[\mathcal{B}]} \cdot \hat{x}_i + \mathbb{E}[\mathcal{B}] \\ =& \sqrt{\sigma_{\mathcal{B}}^2} \cdot \hat{x}_i + \mu_{\mathcal{B}} \\ =& \sqrt{\sigma_{\mathcal{B}}^2} \cdot \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2}} + \mu_{\mathcal{B}} \\ =& x_i \end{aligned}$$

allowing the training process to recover the original activations if needed.
During the inference phase population statistics are used instead of mini-batch ones, this means that the output of a batch normalization layer becomes just a linear transformation

$$y = \frac{\gamma}{\sqrt{\mathbb{E}[x] + \epsilon}} \cdot x + (\beta - \frac{\gamma}{\sqrt{\mathrm{Var}[x] + \epsilon}})$$

with the values of $\mathbb{E}[x]$ and $\mathrm{Var}[x]$ estimated by either computing them over all the batches or by a moving average ran during the training phase.

1.1.4 Instance Normalization

Instance normalization layers, also known as "contrast normalization" layers [20], are normalization layers similar to batch normalization that are used in image stylization applications due to the fact that they allow to apply style in an unique forward pass, rather than by applying it iteratively, making the translation process faster. The main difference between the two is that instance normalization normalizes each feature map on its own, unlike batch normalization which uses information from the whole batch to normalize. Being applied to the feature map only, their advantage over the latter is that they don't need to compute statistics and apply the normalization to the whole batch,

Algorithm 1: Algorithm implemented by a batch normalization layer

```
Data: Batch of samples $\mathcal{B} = \{x^{(1)}, ..., x^{(m)}\}$
      Learned parameters $\gamma$, $\beta$
/* Compute mini-batch mean */
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$
/* Compute mini-batch variance */
$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$
/* Normalize the input data */
$\hat{x}_i \leftarrow \dfrac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$
/* Perform scaling and shifting */
$y_i \leftarrow \gamma \hat{x}_i + \beta$
return $y_i$
```

which makes them more efficient than batch normalization while maintaining the same quality. Lastly, they behave the same way during inference time and training time, as opposed to batch normalization.
As seen in the previous paragraph, batch normalization for batch of images can be written as equation 1.1 if we denote x_{tijk} as $tijk$'th element where t is the index of image within the batch, i is the feature channel index (i.e. color channel in case of an RGB image) and j, k are the spatial dimensions (width and height) indexes then we can replace the batch normalization layer's internal computations into 1.2 to perform instance normalization.

$$y_{tijk} = \frac{x_{tijk} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}, \mu_i = \frac{1}{HWT} \sum_{t=1}^{T} \sum_{l=1}^{W} \sum_{m=1}^{H} x_{tilm}, \sigma_i^2 = \frac{1}{HWT} \sum_{t=1}^{T} \sum_{l=1}^{W} \sum_{m=1}^{H} (x_{tilm} - \mu_i)^2 \tag{1.1}$$

$$y_{tijk} = \frac{x_{tijk} - \mu_{ti}}{\sqrt{\sigma_{ti}^2 + \epsilon}}, \mu_{ti} = \frac{1}{HW} \sum_{l=1}^{W} \sum_{m=1}^{H} x_{tilm}, \sigma_{ti}^2 = \frac{1}{HW} \sum_{l=1}^{W} \sum_{m=1}^{H} (x_{tilm} - \mu_{ti})^2 \tag{1.2}$$

1.2 Generative Adversarial Networks(GANs)

Generative Adversarial Networks, first proposed by [7] are a kind of generative models (top three along with FVBN and VAE) which are used to generate images artificially. They offer a set of advantages when compared to other generative models such as allowing to generate samples in parallel and producing better quality samples than other models. When compared to models that optimize an average such as mean squared error, models with adversarial loss term perform better visually in multi-modal settings(when there are multiple possible output images) as they generate an image which “needs to look” more realistic rather than an image which is an average of all the possible output images, this can be observed in figure 1.7 where the task is to predict a next video frame based on the current input frame.
If viewed as a game between two players (explained more in detail in [9]), GAN training process can be viewed as follows. The game is divided in two scenarios, first scenario (training discriminator only) consists in the following steps:

1. Sample a data point $\boldsymbol{x}$ from the training data.
2. Feed the point $\boldsymbol{x}$ to the discriminator, represented by a differentiable function D.
3. Discriminator tries to make the $D(\boldsymbol{x})$ value close to 1.

the second scenario (training generator and discriminator) is then described by the following steps:

1. Sample a random input noise vector $\boldsymbol{z}$.

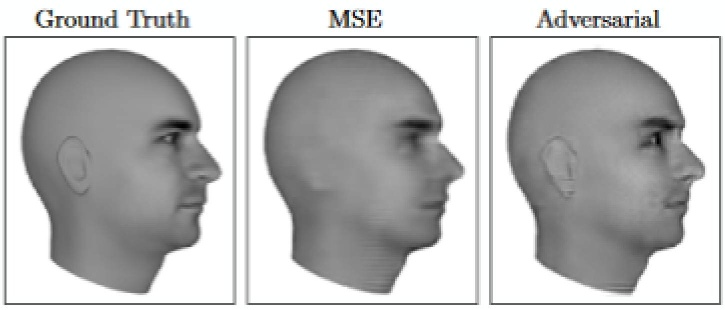


Figure 1.7: The leftmost image is the actual next frame which the model would ideally predict. The center image a result of a model trained with MSE loss between the ground truth frame and all the multiple predicted frames. The rightmost image is a result of a model trained with adversarial loss, which has learned to predict a sharp and realistic image by taking one among multiple possible outputs. (Figure borrowed from [16])

2. Apply a differentiable generator G function (which is usually a neural network) to $\boldsymbol{z}$, compute the value of $\hat{\boldsymbol{x}} \leftarrow G(\boldsymbol{z})$.
3. Generator tries to make $D(\hat{\boldsymbol{x}})$ near 1 while the discriminator tries to make $D(\hat{\boldsymbol{x}})$ near 0.

More formally, we define $D(\boldsymbol{x};\theta_D)$ as a function parametrized by θ_D that outputs the probability of an input $\boldsymbol{x}$ coming from p_{data}, rather than from the generator's distribution p_g. We then define a mapping $G(\boldsymbol{z};\theta_G)$ from $p_z(\boldsymbol{z})$ input noise prior to the data space, which corresponds to a new distribution p_g over the data $\boldsymbol{x}$. The discriminator is trained to maximize the probability of assigning correct labels to the training samples and generated samples as described by the equation:

$$\mathbb{E}_{x\sim p_{data}}[\log D(\boldsymbol{x})] + \mathbb{E}_z[\log(1 - D(G(\boldsymbol{z})))] \tag{1.3}$$

while the generator is trained to minimize the probability of its output getting labeled as fake by the discriminator:

$$\mathbb{E}_z[1 - \log D(G(\boldsymbol{z}))] \tag{1.4}$$

however, in practice, it is preferred for generator to maximize (or to minimize it after multiplying by -1) the equation 1.5 due to weak gradients by G in the early stages of the learning.

$$\mathbb{E}_z[\log D(G(\boldsymbol{z}))] \tag{1.5}$$

Both of the previous equations get combined into a unique function $V(G,D)$, defined by equation 1.6.

$$\min_G \max_D V(G,D) = \mathbb{E}_{x\sim p_{data}}[\log D(\boldsymbol{x})] + \mathbb{E}_z[\log(1 - D(G(\boldsymbol{z})))] \tag{1.6}$$

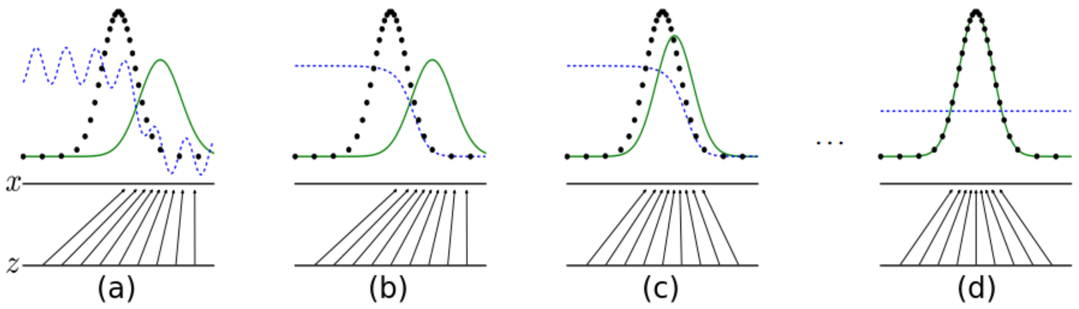


Figure 1.8: The blue dashed line represents the discriminator function D, which assigns a probability of samples coming from p_{data} data generating distribution, represented with a black dotted line or p_g generator's distribution, represented by a continuous green line. The lower line represents the domain from which $\boldsymbol{z}$ is samples, while the arrows indicate the mapping from random noise distribution to generated data, $\boldsymbol{x} = G(\boldsymbol{z})$. (a) If we consider D and G near convergence, p_g is similar to p_{data} and D is a partially accurate classifier. (b) D gets trained to discriminate generated samples from data. (c) G gets trained after being guided by D's gradient to produce samples which are more likely classified as data by D. (d) After enough training steps, D and G should have sufficient capacity, which won't allow both of them to improve anymore because $p_g = p_{data}$ making D assign a probability of $\frac{1}{2}$ to every point.

The training procedure which uses a stochastic gradient descent(SGD) as optimizer on minibatches is illustrated in algorithm 2, while a more intuitive version is shown in figure 1.8.

1.3 Image-to-Image Translation

The image-to-image translation problem consists in capturing special characteristics of a collection of images and finding a way to translate these characteristics into another collection of images. The concept itself can be though as an image version of language translation (for example translating a concept from English to French and vice-versa) and can be seen in figure 1.9 . This task was usually approached with special-purpose methods, even though the settings were always the same, predicting pixels from pixels. Recently a general framework based on Generative Adversarial Networks was proposed to approach this problem. In particular there are two different methods: Conditional GAN and CycleGAN (described in next chapter). The main difference between the two is the kind of training data they consume. In fact, Conditional GANs require paired training data, while CycleGANs don't require data to be paired, bringing a huge advantage due to the fact that unpaired data is way easier to obtain, as we can really grab two subsets of images of the categories we are interested in to perform the training. This difference

Algorithm 2: Minibatch SGD algorithm for GAN training

```
Data: Generator $G$, Discriminator $D$
      Data generating distribution $p_{data}(x)$
      number of epochs $n_{epochs}$, learning rate $\gamma$
for epoch ← 1 to $n_{epochs}$ do
    Sample a minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)},..., \boldsymbol{z}^{(m)}\}$ from prior noise distribution $p_g(\boldsymbol{z})$
    Sample a minibatch of $m$ data samples $\{\boldsymbol{x}^{(1)},..., \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{data}(\boldsymbol{x})$
    $\theta_D \leftarrow \theta_D + \gamma \nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^{m} [\log D(\boldsymbol{x}^{(i)}) + \log(1 - D(G(\boldsymbol{z}^{(i)})))]$
    Sample another minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)},..., \boldsymbol{z}^{(m)}\}$ from prior noise distribution $p_g(\boldsymbol{z})$
    $\theta_G \leftarrow \theta_G - \gamma \nabla_{\theta_G} \frac{1}{m} \sum_{i=1}^{m} [-\log D(G(\boldsymbol{z}^{(i)}))]$
end
```

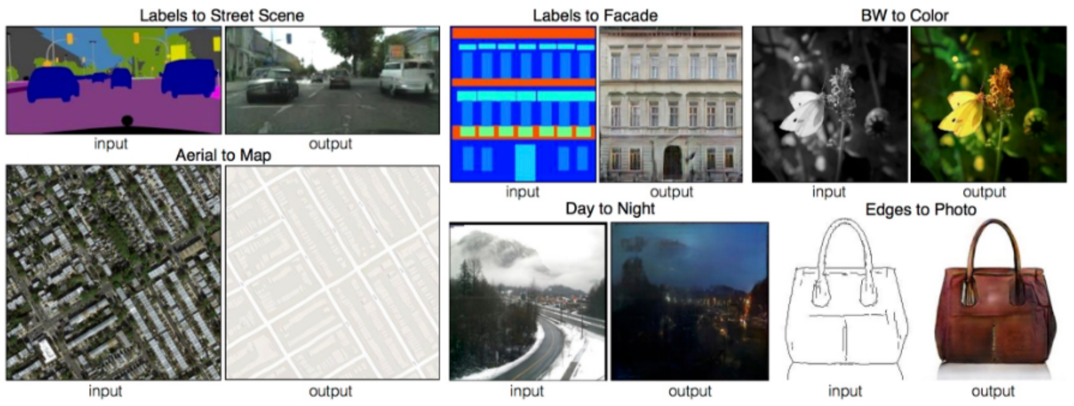


Figure 1.9: Image-to-Image translation task consists in transforming an image visually while preserving the same concept. In this figure we can see several examples of image-to-image translation.

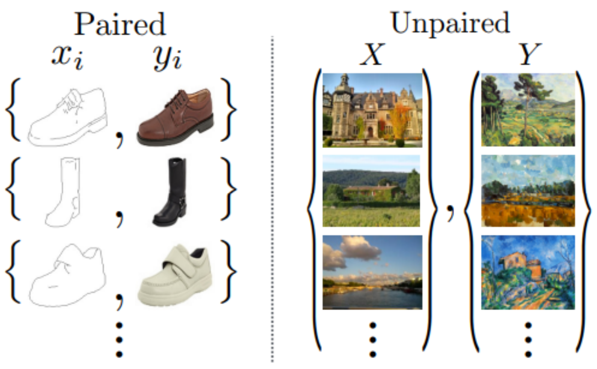


Figure 1.10: Paired image data (left) consists in pairs of data samples $\{x_i, y_i\}_{i=1}^N$ where there exists a correspondence between x_i and y_i. Unpaired image data (right) consists of a source set $\{x_i\}_{i=1}^N$ and a target set $\{y_j\}_{j=1}^M$ without any additional information about which x_i matches which y_j.

between paired and unpaired data can be seen in figure 1.10.

1.4 Tensorflow

Tensorflow[1] is a computational graph framework developed by Google. It is based on the Dataflow programming paradigm, thus it allows us to define programs (Tensorflow can be seen as a standalone language) in form of computational graphs where the nodes represent units of computation(inputs, operations etc.) and edges represent the data consumed or produced by the computation(the incoming edges represent input dependencies and outgoing edges represent the operation's result).

1.4.1 Graph definition

Before being able to perform a computation we need to define a computational graph of what we want to compute. For example listing 1.1 shows an example of a computational graph definition which computes a tanh of the sum of the two given scalar inputs *x1_input* and *x2_input*. This however doesn't let us compute anything, being only a definition of the computational graph that can be seen in figure 1.11. It is also worth noting that everytime we write an operation that uses a placeholder or a variable, a node relative to that operation gets added to the computational graph, called "default computational graph", which can be reset by calling *tf.reset_default_graph()*.

```
import tensorflow as tf
x1_input = tf.placeholder(tf.float32, shape=(), name='x1')
```

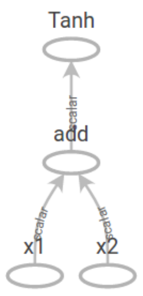


Figure 1.11: Graph created by the code defined in listing 1.1

```
x2_input = tf.placeholder(tf.float32, shape=(), name='x2')
out = tf.tanh(x1_input + x2_input)
```

Listing 1.1: Sum and tanh graph definition example

1.4.2 Sessions

To perform computations we need to start a session and evaluate the desired graph nodes after setting the values of the placeholders. In the listing 1.2 example we can see a way of opening the session which then gets used in order to run a computation on the defined graph. To run the computation we call the session's method *run* and give it either a list of graph nodes or a single node that we want to compute (*out* in this case) and a dictionary where we insert placeholders' values.

```
import tensorflow as tf
# define the computational graph
x1_input = tf.placeholder(tf.float32, shape=(), name='x1')
x2_input = tf.placeholder(tf.float32, shape=(), name='x2')
out = tf.tanh(x1_input + x2_input)
# start the session
session = tf.Session()
result = session.run(out, feed_dict={x1_input: 0.025, x2_input: 0.5})
print(result) # prints 0.012499349
# close the session
session.close()
```

Listing 1.2: Running a session example

1.4.3 Variables

We often need variables in our computation, Tensorflow allows to declare variables with *tf.Variable* (low-level) or *tf.get_variable* (high-level, thus recommended). The first approach is explained in listing 1.3, while the second approach is used in 1.4, the main difference between them is that *tf.Variable*, always creates a new variable and requires an initial value to be specified, while the latter approach checks if a variable already exists and reuses it in case it does. In listing 1.3 we show a computational graph that has a variable that gets incremented by a *tf.assign* operation. Note that we could've intuitively (but mistakenly) used *session.run*([counter, increment_counter]), however when called, we aren't guaranteed the execution order will be the same as in the list, thus it's recommended to make separate run calls when the variables' state is involved.

```
import tensorflow as tf
# declare the variable
counter = tf.Variable(0, dtype=tf.int32, name='counter')
# declare the increment op
increment_counter = tf.assign(counter, counter + 1)
# get the variables initializer op
init_op = tf.global_variables_initializer()
# start a session which will close once the block ends
with tf.Session() as session:
    # initialize the variables
    session.run(init_op)
    print(session.run(counter)) # prints 0
    session.run(increment_counter)
    print(session.run(counter)) # prints 1
```

Listing 1.3: Computational graph with variables example

1.4.4 Optimization

When working with neural networks, we usually need to optimize an objective function, Tensorflow allows us to do this by defining an optimizer operation. Tensorflow offers implementations of all gradient descent algorithm's variants known in literature. For example we define *GradientDescentOptimizer* in listing 1.4 with its relative minimization step operation which takes a loss function operation and a list of variables to optimize as its arguments, this implicitly adds nodes to the graph what allow gradients to be computed automatically (as seen in figure 1.12). We need to run the optimizer step operation every time we want to perform a backward pass and apply the gradients, thus optimizing the given variables list.

```
import tensorflow as tf
import numpy as np
# Declare placeholders
x = tf.placeholder(tf.float32, shape=(None, 2), name="x")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
# Declare variables
W = tf.get_variable("W", [1, 2], dtype=tf.float32)
b = tf.get_variable("b", [1], dtype=tf.float32)
W_T = tf.transpose(W)
```

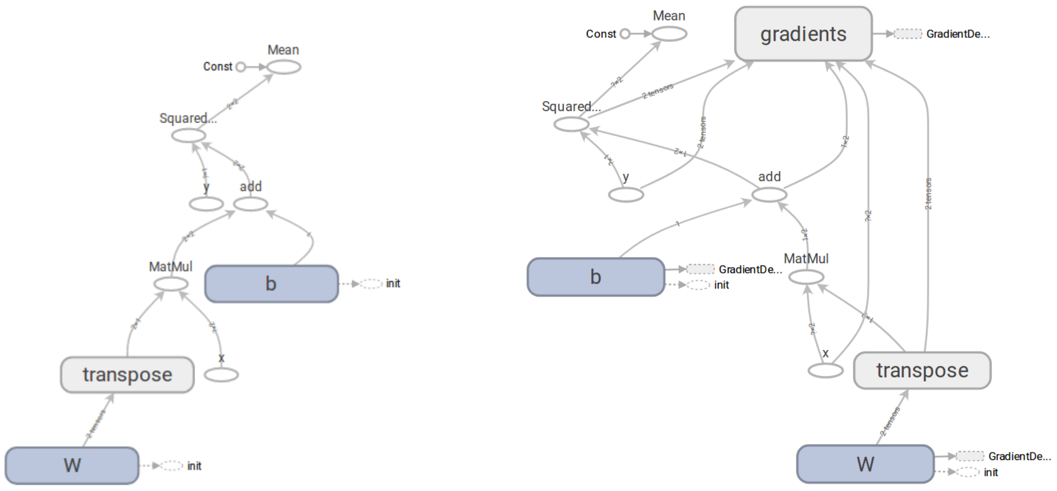


Figure 1.12: Graph created by the code defined in listing 1.4. As we can see, when we declare the optimizer step operation, gradient computation operation gets added implicitly to the graph on the left, producing a new graph which can be seen on the right.

```
y_hat = tf.matmul(W_T, x) + b
# MSE loss function
loss = tf.reduce_mean(tf.squared_difference(y, y_hat))
# SGD with 0.0002 learning rate
optimizer = tf.train.GradientDescentOptimizer(2e-4)
# Optimizer step op
optimizer_step = optimizer.minimize(loss, var_list=tf.trainable_variables
    ())
# Get the variables initializer op
init_op = tf.global_variables_initializer()
# Start a session which will close once the block ends
with tf.Session() as session:
    # Initialize the variables
    session.run(init_op)
    # For 1000 epochs
    for epoch in range(1000):
        # assume we have a ”load_batch” function that returns a batch of
            data
        batch_x, batch_y = load_batch()
        # Run the optimizer step and loss value computation
        _, loss_value = session.run([optimizer_step, loss], feed_dict={x:
            batch_x, y: batch_y})
```

Listing 1.4: Linear regression example

1.4.5 Higher level API for Neural Networks

In previous paragraph we described a low-level approach to define computational graphs, however when working with Neural Networks don't want to get into low-level operations, we want to build networks from more abstract building blocks used in neural networks' literature (dense layers, convolutional layers, known activation functions, etc.). Tensorflow offers two different kind of APIs, the first one *tf.nn* which is a lower level API (yet higher than doing all the operations by hand) and *tf.layers*, which is a higher level API. The *tf.nn* API offers almost the same functionality as the *tf.layers*, however it doesn't handle biases, requires weights to be declared and initialized manually, uses activation functions as external layers and doesn't handle regularizers. As shown in the listing 1.5, we can define a convolutional neural network that recognizes digits from images (trained on MNIST dataset) with a few lines of code. The listing shows the whole process of defining a model's computational graph, in *cnn_model* function, loading a predefined dataset (MNIST), defining an optimizer, creating a session and training the model.

```
import tensorflow as tf
import numpy as np
# Dataset
mnist = tf.contrib.learn.datasets.load_dataset('mnist')
X_train = mnist.train.images
y_train = mnist.train.labels
# Preprocess data
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
# Declare placeholders
def cnn_model(x_input):
    with tf.variable_scope('model'):
        # conv 1
        out = tf.layers.conv2d(x_input, 32, 5, 1)
        out = tf.nn.relu(out)
        out = tf.layers.max_pooling2d(out, 2, 2)
        # conv 2
        out = tf.layers.conv2d(out, 64, 5, 1)
        out = tf.nn.relu(out)
        out = tf.layers.max_pooling2d(out, 2, 2)
        # flatten
        out = tf.layers.flatten(out)
        # dense 1
        out = tf.layers.dense(out, 1024)
        out = tf.nn.relu(out)
        # dense 2
        out = tf.layers.dense(out, 10)
        out = tf.nn.softmax(out)
    return out

x = tf.placeholder(tf.float32, shape=(None, 28, 28, 1))
y = tf.placeholder(tf.float32, shape=(None))
```

```
# convert the target labels into one-hot labels
one_hot_y = tf.one_hot(indices=tf.cast(y, tf.int32), depth=10)
# model's predictions op
y_hat = cnn_model(x)
# loss function
loss = tf.reduce_mean(tf.keras.losses.categorical_crossentropy(y_true=
    one_hot_y, y_pred=y_hat))
# RMSProp with 0.0002 learning rate
optimizer = tf.train.RMSPropOptimizer(1e-4)
# Optimizer step op
optimizer_step = optimizer.minimize(loss, var_list=tf.trainable_variables
    ())
# Get the variables initializer op
init_op = tf.global_variables_initializer()
# Start a session which will close once the block ends
batch_size = 64
dataset_size = X_train.shape[0]
with tf.Session() as session:
    # Initialize the variables
    session.run(init_op)
    # For 30 epochs
    for epoch in range(30):
        # for each batch
        for index, batch_start in enumerate(range(0, dataset_size,
            batch_size)):
            start = batch_start
            end = batch_start+batch_size
            if end < dataset_size:
                        # Run the optimizer step and loss value
                            computation
                _, l = sess.run([optimizer_step, loss], feed_dict={
                x: X_train[start: end]/255.0, y: y_train[start: end]
             })
```

Listing 1.5: MNIST CNN example

1.4.6 Saving the graph

Once the model was trained, we might be interested into persisting it and using it in order to make predictions. In praticular we can save the session associated with the computational graph by using *tf.train.Saver* class as shown in listing 1.6.

```
import tensorflow as tf
...
saver = tf.train.Saver()
with tf.Session() as session:
    ...
    saver.save(session, 'checkpoint-name')
```

Listing 1.6: Saving a session example

When calling *save* method of *tf.train.Saver* the computational graph's state gets saved to secondary memory as a collection of four files listed below:

- **.meta**: contains the structure of saved graph.
- **.data**: contains the values of variables.
- **.index**: checkpoint's identification.
- **checkpoint**: a list of recent checkpoints.

We are usually interested in restoring the session as well, this can be done by calling the *restore* method of *tf.train.Saver*. We assume (in listing 1.7) that we've already defined the computational graph previously before restoring the session, as it would cause an exception to be thrown otherwise.

```
import tensorflow as tf
...
saver = tf.train.Saver()
with tf.Session() as session:
    ...
    saver.restore(session, 'checkpoint-name')
```

Listing 1.7: Restoring a session example

There is however a way to restore the session without declaring the graph before, this can be done by calling the *tf.train.import_meta_graph* with **.meta** file path method, which returns a *tf.train.Saver* instance that can then be used to restore the session as previously explained.

Chapter 2

CycleGAN

2.1 Introduction

We are given two domains of images X and Y with their respective training samples $\{x_i\}_{i=1}^{N}$ and $\{y_j\}_{j=1}^{M}$ such that $x_i \in X$ and $y_j \in Y$ and their data generating distributions $p_{data}(x)$ and $p_{data}(y)$. The goal here is to learn a mapping $G : X \to Y$, such that the distribution of images from G(X) becomes indistinguishable from the distribution Y using an adversarial loss.
We'll use a different notation from the original paper [22], to emphasize the translation part better, thus we rewrite $G : X \to Y$ as $G_{XY} : X \to Y$ and $F : Y \to X$ as $G_{YX} : Y \to X$. Additionally we introduce two adversarial discriminators D_X and D_Y to distinguish between $\{x\}$ and $\{G_{YX}(y)\}$ in case of D_X and similarly in case of D_Y.

2.2 Loss function

2.2.1 Adversarial loss term

There are two usual (as defined in [7]) adversarial loss terms shown in equations 2.1 and 2.2 which are applied to both G_{XY} and G_{YX} respectively.

$$\begin{aligned}\mathcal{L}_{GAN}(G_{XY}, D_Y, X, Y) &= \mathbb{E}_{y \sim p_{data(y)}}[\log(D_Y(y))] \\ &+ \mathbb{E}_{x \sim p_{data(x)}}[\log(1 - D_Y(G_{XY}(x)))]\end{aligned} \tag{2.1}$$

$$\begin{aligned}\mathcal{L}_{GAN}(G_{YX}, D_X, Y, X) &= \mathbb{E}_{x \sim p_{data(x)}}[\log(D_X(x))] \\ &+ \mathbb{E}_{y \sim p_{data(y)}}[\log(1 - D_X(G_{YX}(y)))]\end{aligned} \tag{2.2}$$

It is however preferable to use the LSGAN [17] objective for a more stable training (according to the authors of CycleGAN [22]), so instead of optimizing the objective

$$\min_{G_{XY}} \max_{D_Y} \mathcal{L}_{GAN}(G_{XY}, D_Y, X, Y)$$

from 2.1, we optimize the objective which is described in equation 2.3 for G_{XY} and D_Y.

$$\begin{aligned}&\min_{G_{XY}} \mathbb{E}_{x\sim p_{data(x)}}[(D_Y(G_{XY}(x))-1)^2]\\&\min_{D_Y} \mathbb{E}_{y\sim p_{data(y)}}[(D_Y(y)-1)^2]+\mathbb{E}_{x\sim p_{data(x)}}[D_Y(G_{XY}(x))^2]\end{aligned} \tag{2.3}$$

The equations for G_{YX} and D_X are described in a similar way thus are left out. Intuitively G_{XY} attempts to translate images from X to Y, generating $G_{XY}(x)$, while the discriminator D_Y distinguishes between translated $G_{XY}(x)$ and real samples y. The same reasoning is applied to G_{YX} and D_X for a symmetrical translation from Y to X.

2.2.2 Cycle Consistency loss term

$$\begin{aligned}\mathcal{L}_{cyc}(G_{XY}, G_{YX}) = &\,\mathbb{E}_{x\sim p_{data(x)}}[|G_{YX}(G_{XY}(x))-x|]\\&+\mathbb{E}_{y\sim p_{data(y)}}[|G_{XY}(G_{YX}(y))-y|]\end{aligned} \tag{2.4}$$

The term from equation 2.4 is introduced because the adversarial loss alone can't guarantee(in practice) that the learned mapping effectively maps the input image x_i to a correct image y_i. What can happen is that all of the images of X get mapped to an unique image of Y. The paper's authors argue that the learned translation functions should be cycle-consistent in order to further reduce the space of possible translation functions. Intuitively this term represents the *forward cycle consistency* constraint (as shown in figure 2.1 (b))

$$\forall x \in X.\, G_{YX}(G_{XY}(x)) \approx x$$

and the *backward cycle consistency* constraint (as shown in figure 2.1 (c))

$$\forall y \in Y.\, G_{XY}(G_{YX}(y)) \approx y$$

2.2.3 Full objective

We train to minimize a more stable version of the $\mathcal{L}_{GAN}(G_{XY}, D_Y, X, Y)$ by using equation 2.3 instead of adversarial loss terms. The full objective thus is defined in the following way:

$$\begin{aligned}\mathcal{L}(G_{XY}, G_{YX}, D_X, D_Y) = &\,\mathcal{L}_{GAN}(G_{XY}, D_Y, X, Y)\\&+\mathcal{L}_{GAN}(G_{YX}, D_X, Y, X)\\&+\lambda\mathcal{L}_{cyc}(G_{XY}, G_{YX})\end{aligned} \tag{2.5}$$

where λ indicates the importance of the cycle term and the optimal solution is:

$$G^*_{XY}, G^*_{YX} = \arg\min_{G_{XY}, G_{YX}} \max_{D_X, D_Y} \mathcal{L}(G_{XY}, G_{YX}, D_X, D_Y) \tag{2.6}$$

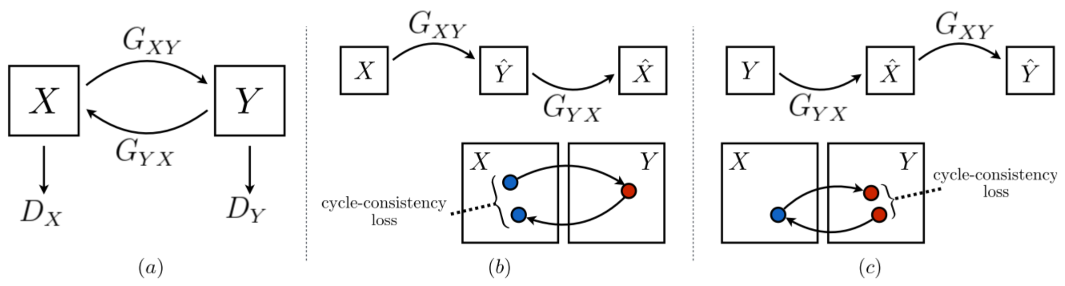


Figure 2.1: (a) The model consists of two translation functions $G_{XY} : X \to Y$ and $G_{YX} : Y \to X$, and their respective discriminators D_X and D_Y. The discriminators encourage their respective generators to produce outputs of the translated inputs indistinguishable from the opposite category (i.e. G_{XY} should translate X inputs into images indistinguishable from the domain Y). The translation is further regularized by adding two additional cycle consistency loss terms that capture the intuition that a translation performed from one domain to the other and back again should produce an almost identical result to the initial input image. (b) Forward cycle-consistency loss represents the loss term between an input image of domain X and its translation towards Y and backwards. (c) Backward cycle-consistency loss represents the loss term between an input image of domain Y and its translation towards X and backwards.

2.3 Implementation and training

2.3.1 Network Architectures

The generator network architecture is usually defined as either an U-Net (this approach is better described in [12]) or as a six-block residual network defined by

c7s1-32,d64,d128,R128,R128,R128,R128,R128,R128,u64,u32,c7s1-3

for 128×128 resolution images and a nine-block residual network defined by

c7s1-32,d64,d128,R128,R128,R128,R128,R128,R128,R128,R128,R128,u64,u32,c7s1-3

in case of 256×256 resolution images, where:

- c7s1-k denotes a 7×7 Convolution-InstanceNormalization-ReLU layer with k filters and a stride of 1.
- dk denotes a 3×3 Convolution-InstanceNormalization-ReLU layer with k filters, and a stride of 2.

- Rk denotes a residual block that contains two 3×3 convolutional layers with the same number of filters on both layers.
- uk denotes a 3×3 fractional-strided-Convolution-InstanceNormalization-ReLU layer with k filters, and a stride of $\frac{1}{2}$.

The discriminator network architecture is usually implemented by a 70×70 PatchGAN (based on [12]), which is a discriminator network that works on variable dimension images and outputs a matrix of labels (8×8 in case of 128×128 images and 16×16 in case of 256×256 images) rather than a single scalar, indicating the probability of a smaller image zone being real or fake.
The discriminator network is thus defined in the following way:

C64,C128,C256,C512

where Ck denotes a 4×4 Convolution-InstanceNormalization-LeakyReLU layer with k filters and a stride of 2 and a convolution is used in last layer to produce a one-dimensional output. The first layer doesn't use InstanceNormalization and $\alpha = 0.2$ for all the LeakyReLU layers.

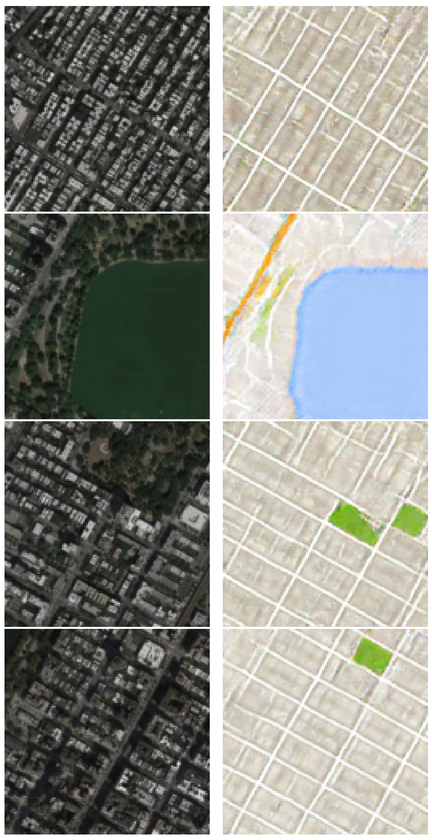
2.3.2 Training

During the training phase CycleGAN's authors used a strategy to reduce model's oscillation by updating the discriminators using a history of previously generated images rather than the currently generated ones (this allows to improve discriminator). The previously generated images buffer size was set to 50. The training phase was performed with λ set to 10, Adam optimizer with a batch size of 1 and an initial learning rate of 0.0002. The learning rate was kept 0.0002 for the first 100 epochs, and was linearly decayed to 0 over the next 100 epochs, as defined by the following function

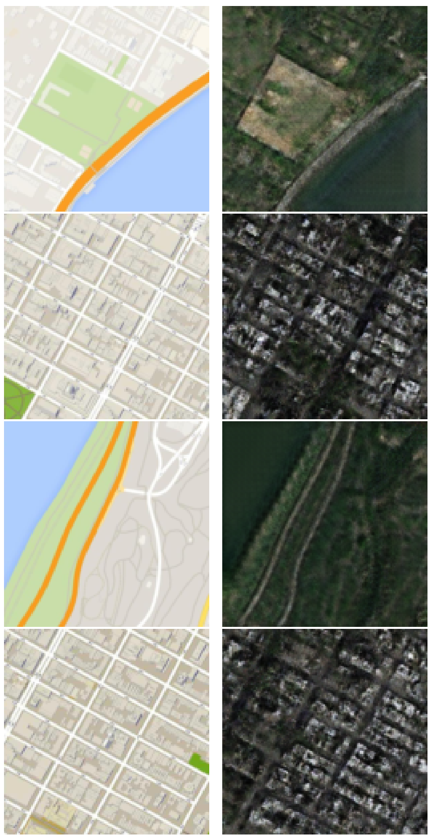
$$lr(i) = \begin{cases} 0.0002 * \dfrac{200 - i}{100} & \text{if } i > 100 \\ 0.0002 & \text{otherwise} \end{cases}$$

2.3.3 Implementation

We propose an implementation of the CycleGAN model which is based on [22]. The main difference between this implementation and the original one that we always resize images to a dimension of 128×128, thus we use the architecture containing six residual blocks. Despite training being done with a smaller network (original implementation used nine residual blocks and 256×256 images), the results are worth experimentations that will be described in the next chapter. The results over the **maps** dataset can be seen in figure 2.2, while the results for **horse2zebra** dataset are available in 2.3.

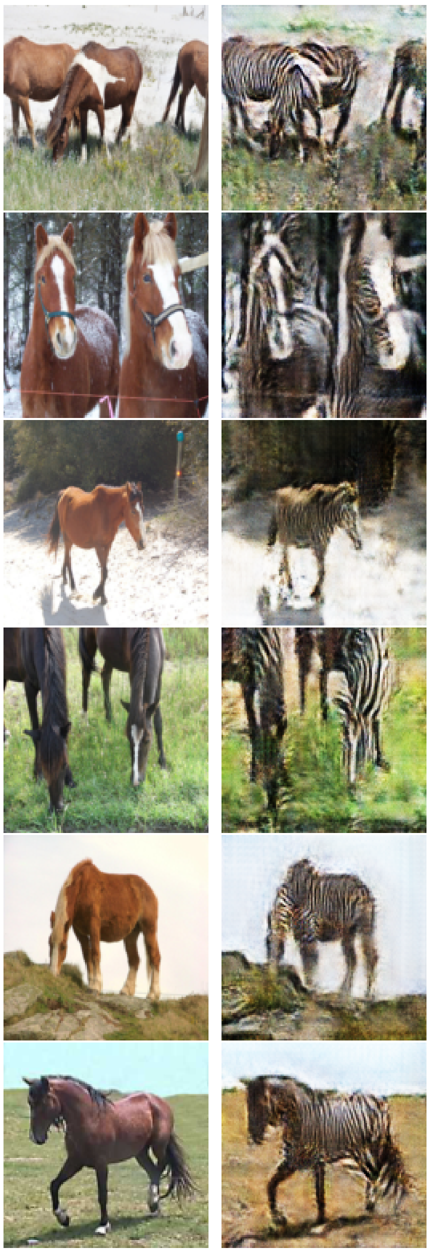


Input image x (left), generated $G_{XY}(x)$ (right)



Input image y (left), generated $G_{YX}(y)$ (right)

Figure 2.2: CycleGAN trained on maps dataset. The first two columns represent the forward translation with the original aerial view image on the left and a translated GoogleMaps image on the right. The last two columns represent the backward translation from GoogleMaps image to aerial view.


Input image x (left), generated $G_{XY}(x)$ (right)

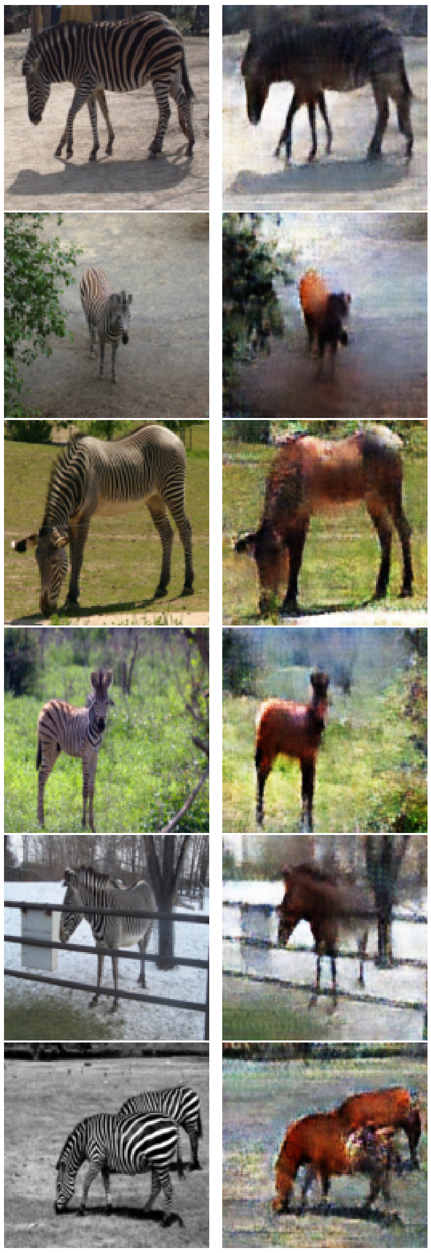

Input image y (left), generated $G_{YX}(y)$ (right)

Figure 2.3: CycleGAN trained on horse2zebra dataset. The first two columns represent the forward translation with original horse image on left and translated zebra image on right. The last two columns represent the backward translation with the original image being a zebra and translated image being a horse.

Chapter 3

Single Translation Function CycleGANs

The goal of this dissertation was to perform experimentations on a modified version the existing CycleGAN architecture in order to make it able to translate images in a gradual way. We call **gradual translation** an image translation process that doesn't necessarily fully translate from one class to another, it includes the intermediate classes as well. For example let us consider horse2zebra dataset and assume that the domain of images of horses corresponds to a certain $class_0$, let's say being labeled by a value of 0, and the images zebras corresponding to a $class_1$ labeled by 1, a CycleGAN as defined in previous chapter can only perform translations between two classes 0 and 1, we instead want to perform translations between classes in $[0, 1]$ interval.

3.1 Introduction

We define a CycleGAN with a single translation function as a mapping $G : Image \times Class \rightarrow Image_{Class}$ where the output image's class is conditioned by an input class label that we consider to be in $[0, 1]$ interval for simplicity sake. We ideally would like the network to learn a mapping which allows to translate images towards any given class, from any given class of the $[0, 1]$ interval.
There are two main problems which seem to occur during the training of such networks. The first problem comes from the fact that we have an unique translation function, the network doesn't have any information about the input image's class due to this, so it has to learn it somehow. The second problem comes from the fact that class information needs to be embedded into a network architecture which doesn't have a latent space, forcing us to either concatenate it to an intermediate layer or to the input image as an additional channel. Another thing that can be done is to sum the class information to

an intermediate representation of the image.

3.2 Attempts and Results

In our attempts we used a modified generator architecture similar to the one used by CycleGANs with the main difference being class information embeddings that needed to be added to the generator. First attempt used only a concatenation of 3 channels containing the class label to the original image. Attempts 2-4 had the class information pixel-wise summed to the processed image before being passed to each residual block. Attempts 5-8 has class information concatenated as an additional channel to the first residual block's input. Besides this each attempt had a different loss function (for discriminator, generator or both) which will be described in details successfully. Lastly, during the training we fixed the values of c_0 and c_1 to 0 and 1 respectively, and used horse2zebra dataset.

3.2.1 First

In our first attempt we used one parametrized discriminator only which had the job to detect whether an image was of the right class or not. The generator's loss term was defined as follows, with $\mathcal{L}_{cyc}$ and λ as seen in CycleGAN:

$$\begin{aligned}
\mathcal{L}_{G_0} &= \mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D(G(x_0, c_1), c_1) - 1)^2] \\
\mathcal{L}_{G_1} &= \mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D(G(x_1, c_0), c_0) - 1)^2] \\
\mathcal{L}_G &= \lambda \mathcal{L}_{cyc} + \mathcal{L}_{G_0} + \mathcal{L}_{G_1}
\end{aligned}$$

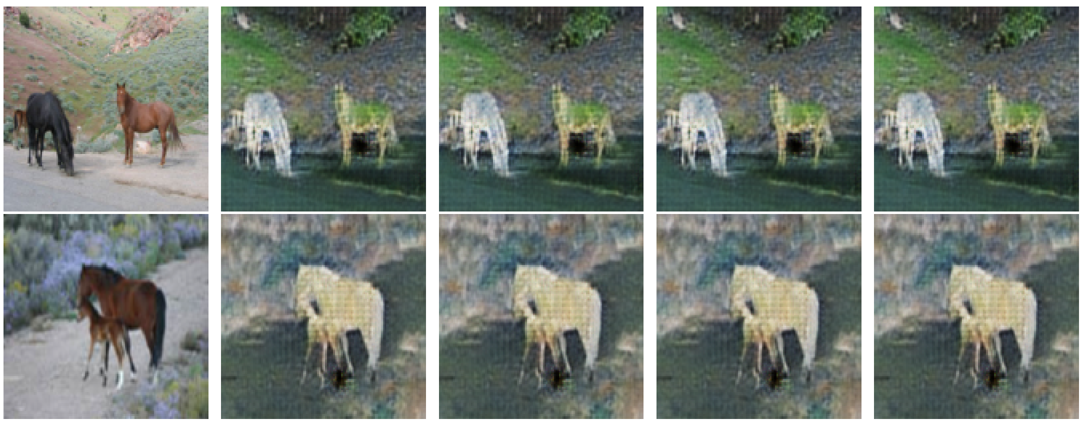
What we do there is apply the usual CycleGAN's loss term, except that we have two categories which have to be labeled as true by the discriminator. The discriminator's loss term was defined as a sum of two discriminator terms, one for each class, combined into one as follows:

$$\begin{aligned}
\mathcal{L}_{D_0} &= \mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D(x_0, c_0) - 1)^2] + \mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D(G(x_1, c_0), c_0))^2] \\
\mathcal{L}_{D_1} &= \mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D(x_1, c_1) - 1)^2] + \mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D(G(x_0, c_1), c_1))^2] \\
\mathcal{L}_D &= \mathcal{L}_{D_0} + \mathcal{L}_{D_1}
\end{aligned}$$

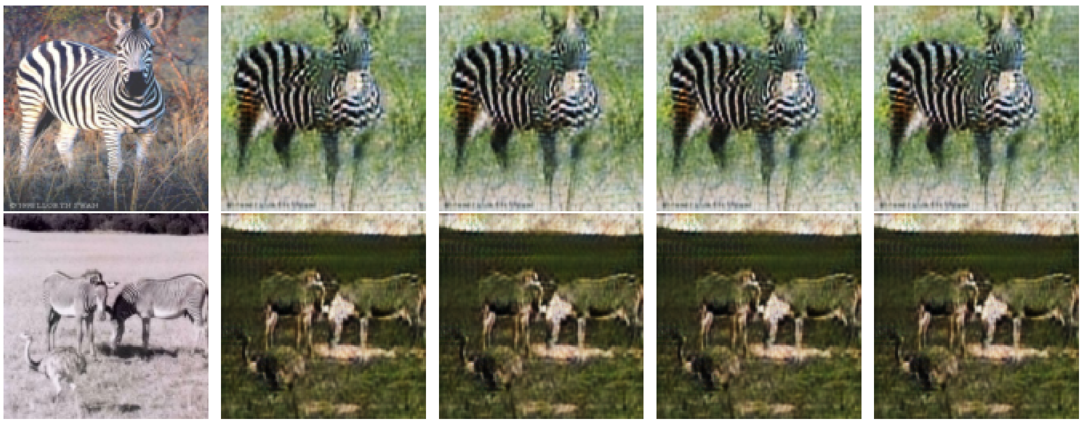
The results of this can be seen in figure 3.1. As we can notice, the class information seems to be ignored by the generator when translating the images.

3.2.2 Second

In the second attempt we used two distinct discriminators for each class, instead of one, as the problem might have been a low capacity of a single discriminator. We changed



(a) Translation from an original image x_0 (horse) towards categories 0 (identity), 0.2, 0.5, 1 (opposite).



(b) Translation from an original image x_1 (zebra) towards categories 1 (identity), 0.5, 0.2, 0 (opposite).

Figure 3.1: Results of the first attempt.

the generator's loss term into the following:

$$\begin{aligned}
&\mathcal{L}_{G_0} = \mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D_{c_1}(G(x_0, c_1)) - 1)^2] \\
&\mathcal{L}_{G_1} = \mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D_{c_0}(G(x_1, c_0)) - 1)^2] \\
&\mathcal{L}_G = \lambda \mathcal{L}_{cyc} + \mathcal{L}_{G_0} + \mathcal{L}_{G_1}
\end{aligned}$$

in order to do so, and defined two discriminator loss terms for each discriminator to minimize:

$$\begin{aligned}
&\mathcal{L}_{D_0} = \mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D_{c_0}(x_0) - 1)^2] + \mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D_{c_0}(G(x_1, c_0)))^2] \\
&\mathcal{L}_{D_1} = \mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D_{c_1}(x_1) - 1)^2] + \mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D_{c_1}(G(x_0, c_1)))^2]
\end{aligned}$$

This did not bring any improvement, as can be seen in figure 3.2.

3.2.3 Third

In the third attempt we included an identity term in the generator, as we though that it would it help to make use of class information. The generator thus included an additional term ($\mathcal{L}_{identity}$) which was intended to minimize the distance between the translated images into the same class and the original image as can be seen in equation 3.1.

$$\begin{aligned}
&\mathcal{L}_{identity} = \mathbb{E}_{x_0 \sim p_{data(x_0)}}[|G(x_0, c_0)) - x_0|] + \mathbb{E}_{x_1 \sim p_{data(x_1)}}[|G(x_1, c_1)) - x_1|] \\
&\mathcal{L}_{G_0} = \mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D_{c_1}(G(x_0, c_1)) - 1)^2] \\
&\mathcal{L}_{G_1} = \mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D_{c_0}(G(x_1, c_0)) - 1)^2] \\
&\mathcal{L}_G = \lambda_{cyc} \mathcal{L}_{cyc} + \lambda_{identity} \mathcal{L}_{identity} + \mathcal{L}_{G_0} + \mathcal{L}_{G_1}
\end{aligned} \tag{3.1}$$

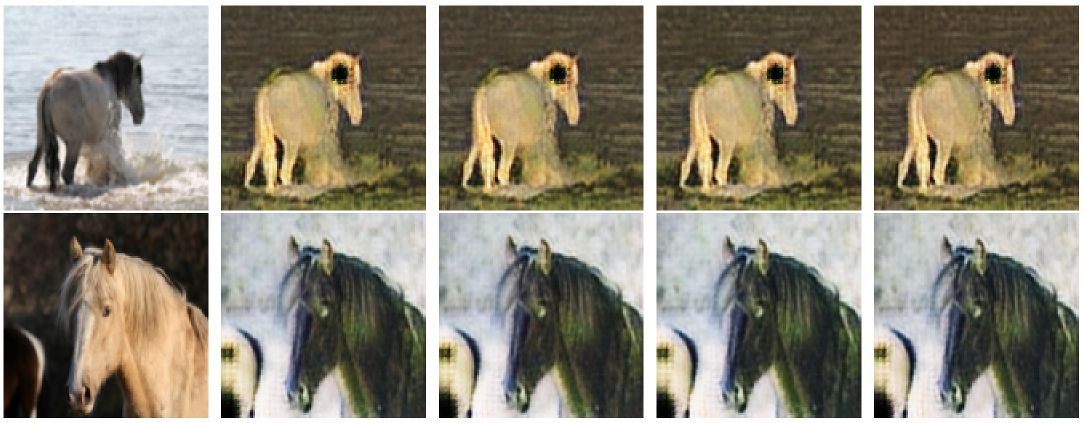
The results can be seen in figure 3.3. It is worth noting that the added term caused majority of the translated images to become the same as the input (learning the identity mapping).

3.2.4 Forth

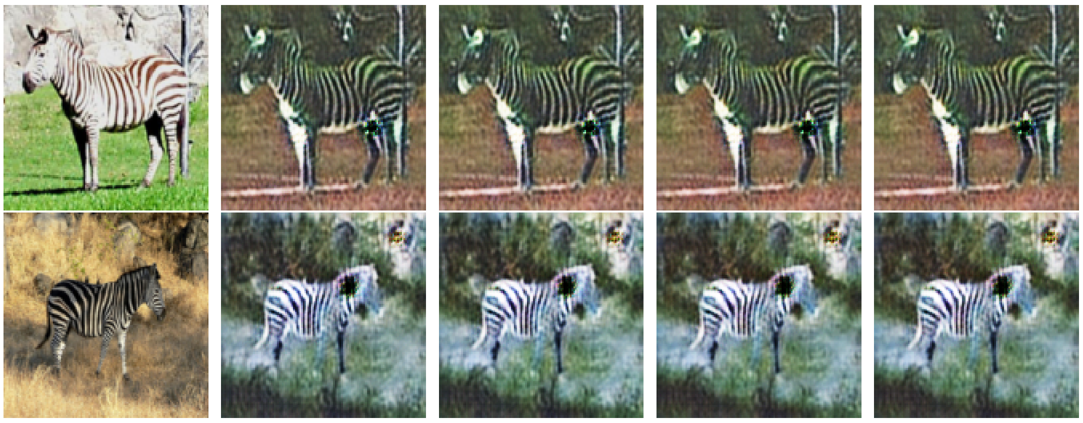
Forth attempt was similar to the third with $\lambda_{identity}$ set to 1, this was done with the idea that the generator wouldn't try to learn only the identity function by lowering the identity term's weight. The results however didn't match the expectations and can be observed in figure 3.4.

3.2.5 Fifth

In the fifth attempt we added an identity term to the discriminator as well and a discriminator term for identity images in the generator. The generator's $\mathcal{L}_{G_0}$ and $\mathcal{L}_{G_1}$ terms



(a) Translation from an original image x_0 (horse) towards categories 0 (identity), 0.2, 0.5, 1 (opposite).



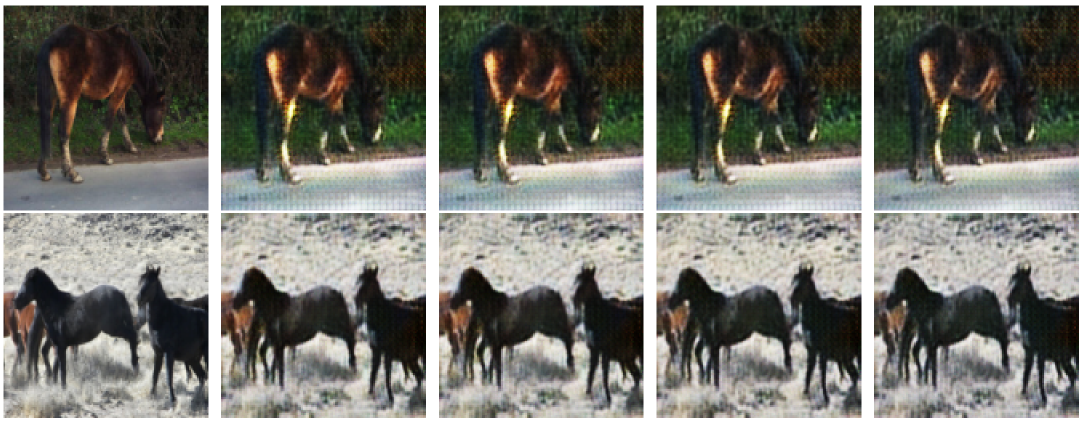
(b) Translation from an original image x_1 (zebra) towards categories 1 (identity), 0.5, 0.2, 0 (opposite).

Figure 3.2: Results of the second attempt.

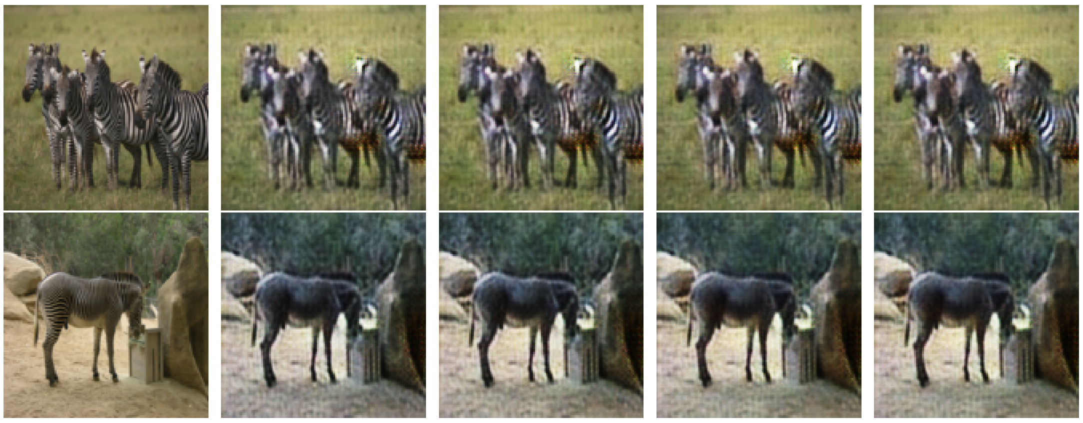
got changed into:

$$\begin{aligned}
\mathcal{L}_{identity} &= \mathbb{E}_{x_0 \sim p_{data(x_0)}}[|G(x_0, c_0)) - x_0|] + \mathbb{E}_{x_1 \sim p_{data(x_1)}}[|G(x_1, c_1)) - x_1|] \\
\mathcal{L}_{G_0} &= \mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D_{c_1}(G(x_0, c_1)) - 1)^2] + \mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D_{c_0}(G(x_0, c_0)) - 1)^2] \\
\mathcal{L}_{G_1} &= \mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D_{c_0}(G(x_1, c_0)) - 1)^2] + \mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D_{c_1}(G(x_1, c_1)) - 1)^2] \\
\mathcal{L}_G &= \lambda_{cyc}\mathcal{L}_{cyc} + \lambda_{identity}\mathcal{L}_{identity} + \mathcal{L}_{G_0} + \mathcal{L}_{G_1}
\end{aligned} \tag{3.2}$$

Discriminators' terms got changed to include the identity translation case, thus they now had to take into account that an identity translation would be a fake image (in order to



(a) Translation from an original image x_0 (horse) towards categories 0 (identity), 0.2, 0.5, 1 (opposite).



(b) Translation from an original image x_1 (zebra) towards categories 1 (identity), 0.5, 0.2, 0 (opposite).

Figure 3.3: Results of the third attempt.

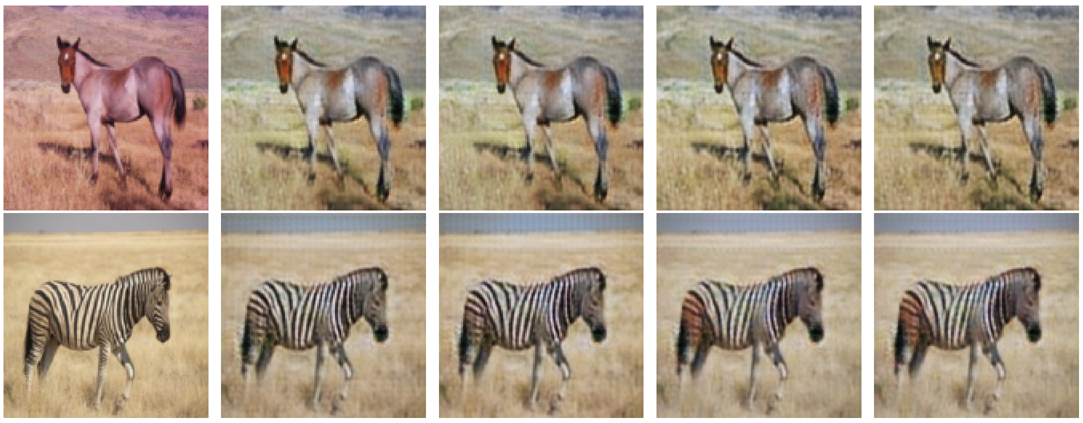


Figure 3.4: Results of the forth attempt. The upper image shows a translation from category 0 towards the four categories defined previously. The lower image shows the same but from category 1.

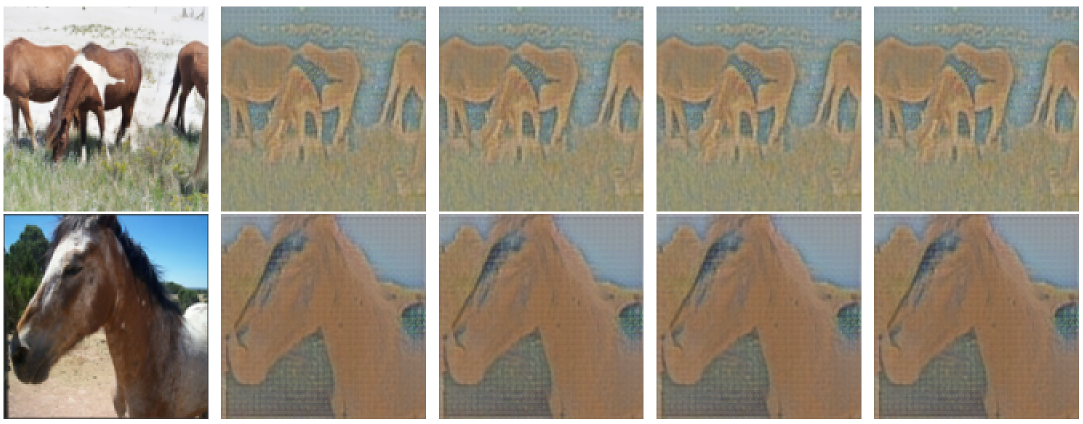
avoid learning the identity mapping):

$$\begin{aligned}
\mathcal{L}_{D_0} = &\mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D_{c_0}(x_0) - 1)^2] + \\
&\frac{\mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D_{c_0}(G(x_1, c_0)))^2] + \mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D_{c_0}(G(x_0, c_0)))^2]}{2} \\
\mathcal{L}_{D_1} = &\mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D_{c_1}(x_1) - 1)^2] + \\
&\frac{\mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D_{c_1}(G(x_0, c_1)))^2] + \mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D_{c_1}(G(x_1, c_1)))^2]}{2}
\end{aligned} \tag{3.3}$$

The results are shown in figure 3.5. We can observe that the identity mapping was indeed not learned, however, the translation quality did not improve as it started producing images with colors not related to the dataset.

3.2.6 Sixth

In the sixth attempt we added another multiplication constant λ_{disc} to the generator's $\mathcal{L}_{G_i}$ loss terms with it's value set to 5. The constant was added only to the opposite class translation term in order to make the translation towards the opposite class a higher priority than the identity. The generator's loss can be seen in equation 3.4, while the results are shown in 3.6. To our surprise, even though the network did seem to ignore the class information, the translation towards the opposite class was performed pretty


(a) Translation from an original image x_0 (horse) towards categories 0 (identity), 0.2, 0.5, 1 (opposite).

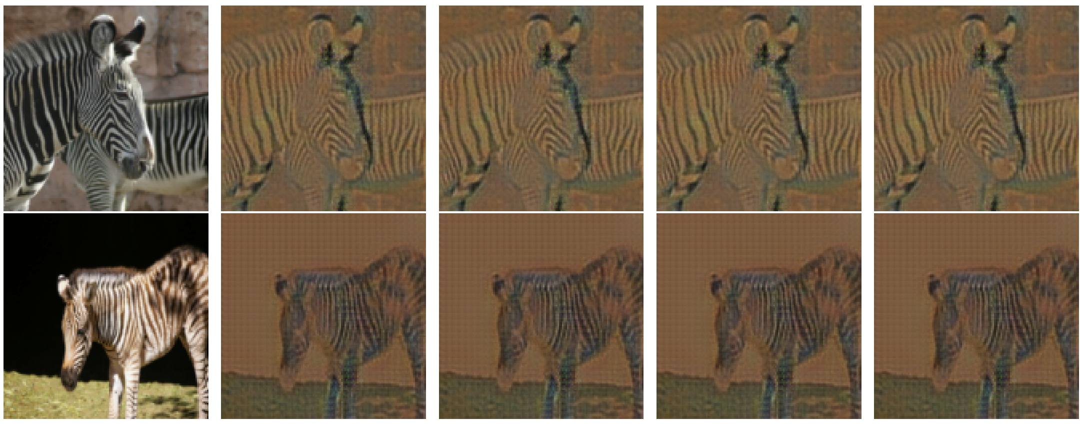
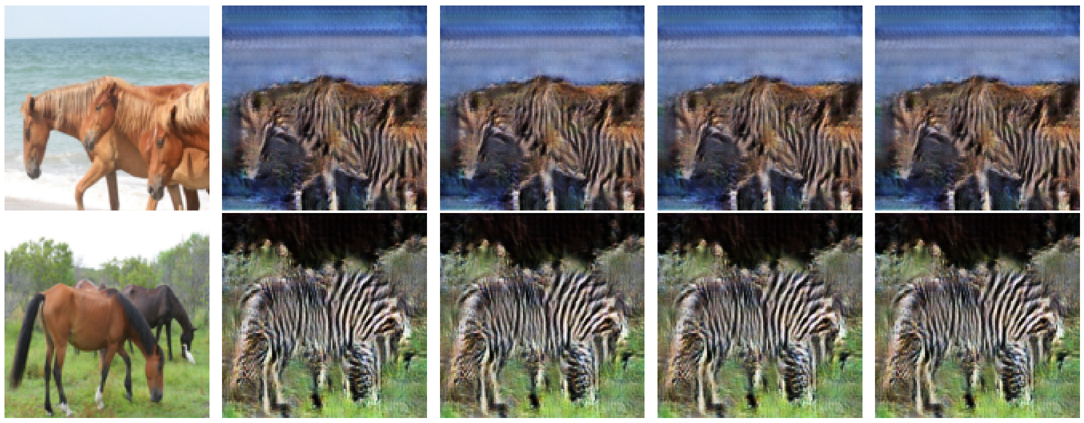
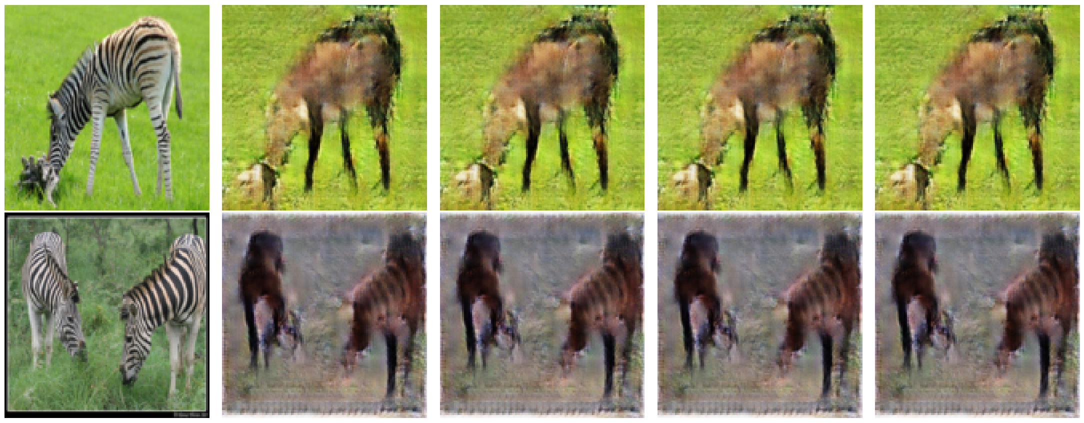

(b) Translation from an original image x_1 (zebra) towards categories 1 (identity), 0.5, 0.2, 0 (opposite).

Figure 3.5: Results of the fifth attempt.



(a) Translation from an original image x_0 (horse) towards categories 0 (identity), 0.2, 0.5, 1 (opposite).



(b) Translation from an original image x_1 (zebra) towards categories 1 (identity), 0.5, 0.2, 0 (opposite).

Figure 3.6: Results of the sixth attempt.

much correctly with only one generator network.

$$\begin{aligned}
&\mathcal{L}_{identity} = \mathbb{E}_{x_0\sim p_{data(x_0)}}[|G(x_0,c_0)) - x_0|] + \mathbb{E}_{x_1\sim p_{data(x_1)}}[|G(x_1,c_1)) - x_1|]\\
&\mathcal{L}_{G_0} = \lambda_{disc}\,\mathbb{E}_{x_0\sim p_{data(x_0)}}[(D_{c_1}(G(x_0,c_1)) - 1)^2] + \mathbb{E}_{x_0\sim p_{data(x_0)}}[(D_{c_0}(G(x_0,c_0)) - 1)^2]\\
&\mathcal{L}_{G_1} = \lambda_{disc}\,\mathbb{E}_{x_1\sim p_{data(x_1)}}[(D_{c_0}(G(x_1,c_0)) - 1)^2] + \mathbb{E}_{x_1\sim p_{data(x_1)}}[(D_{c_1}(G(x_1,c_1)) - 1)^2]\\
&\mathcal{L}_G = \lambda_{cyc}\mathcal{L}_{cyc} + \lambda_{identity}\mathcal{L}_{identity} + \mathcal{L}_{G_0} + \mathcal{L}_{G_1}
\end{aligned} \tag{3.4}$$

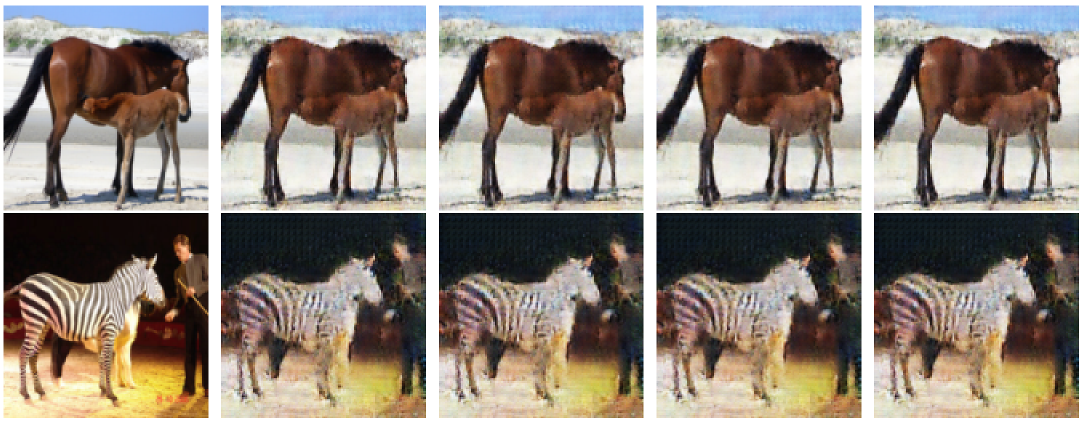


Figure 3.7: Results of the seventh attempt. The upper image shows a translation from category 0 towards the four categories defined previously. The lower image shows the same but from category 1.

3.2.7 Seventh

Encouraged by the previous results, we decided to multiply the whole $\mathcal{L}_{G_i}$ terms by λ_{disc} with the same value as used in previous attempt, however it did not bring any satisfactory results, which can be seen in figure 3.7. The generator's loss term became

$$\mathcal{L}_G = \lambda_{cyc}\mathcal{L}_{cyc} + \lambda_{identity}\mathcal{L}_{identity} + \lambda_{disc}\mathcal{L}_{G_0} + \lambda_{disc}\mathcal{L}_{G_1}$$

3.2.8 Eighth

In this last attempt we set the weight term for discriminated identity images $\lambda_{disc_{identity}}$ to 5 and the weight term λ_{disc} for discriminated opposite class images in generator to 2 with the expectation to improve results of sixth attempt. The generator's loss function became a general case of 3.4, and was defined as follows:

$$\mathcal{L}_{identity} = \mathbb{E}_{x_0 \sim p_{data(x_0)}}[|G(x_0, c_0)) - x_0|] + \mathbb{E}_{x_1 \sim p_{data(x_1)}}[|G(x_1, c_1)) - x_1|]$$
$$\mathcal{L}_{G_0} = \lambda_{disc}\,\mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D_{c_1}(G(x_0, c_1)) - 1)^2] + \lambda_{disc_{identity}}\,\mathbb{E}_{x_0 \sim p_{data(x_0)}}[(D_{c_0}(G(x_0, c_0)) - 1)^2]$$
$$\mathcal{L}_{G_1} = \lambda_{disc}\,\mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D_{c_0}(G(x_1, c_0)) - 1)^2] + \lambda_{disc_{identity}}\,\mathbb{E}_{x_1 \sim p_{data(x_1)}}[(D_{c_1}(G(x_1, c_1)) - 1)^2]$$
$$\mathcal{L}_G = \lambda_{cyc}\mathcal{L}_{cyc} + \lambda_{identity}\mathcal{L}_{identity} + \mathcal{L}_{G_0} + \mathcal{L}_{G_1}$$

The results didn't match our expectations and can be seen in figure 3.8

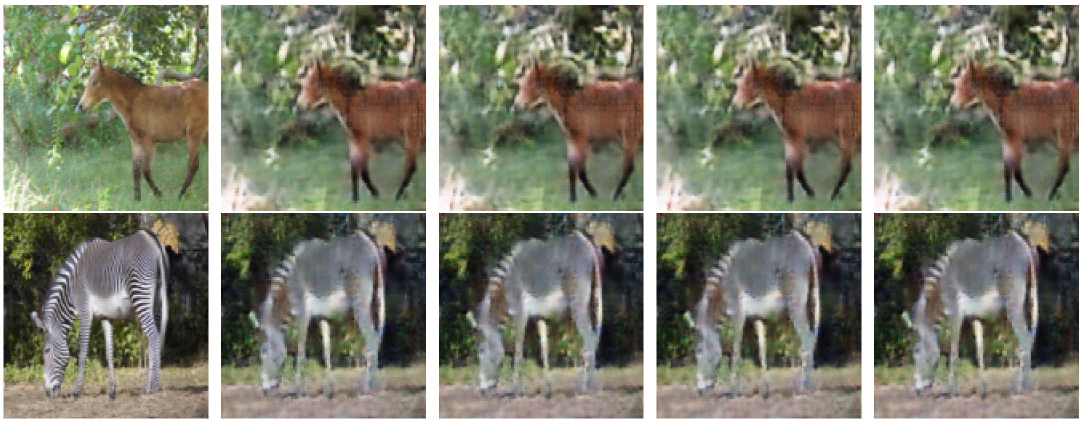


Figure 3.8: Results of the eighth attempt. The upper image shows a translation from category 0 towards the four categories defined previously. The lower image shows the same but from category 1.

Chapter 4

Conclusions

Our goal for this dissertation was to use the CycleGAN framework in order to allow a continuous translation between two given classes by using a single translator network parametrized by class information.
Even though the initial problem of translating the images towards any class of the $[0, 1]$ interval wasn't solved, the results of sixth attempt were interesting due to the fact that we were able to obtain a translation between two opposite classes by using a single generator network instead of two. What can be done next is to use that same loss term with a higher capacity network (for example with nine residual blocks and images with a resolution of 256×256).
Another thing that can be attempted to parametrize the network is to follow the Augmented CycleGAN's [2] approach to embed conditional information (class information in our case) by using Conditional Normalization [18][4] for all normalization layers instead of channel concatenation, as the authors claim that they found it more effective in similar settings as ours.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Amjad Almahairi, Sai Rajeswar, Alessandro Sordoni, Philip Bachman, and Aaron C. Courville. Augmented cyclegan: Learning many-to-many mappings from unpaired data. *CoRR*, abs/1802.10151, 2018.

[3] Yunjey Choi, Min-Je Choi, Munyoung Kim, Jung-Woo Ha, Sunghun Kim, and Jaegul Choo. Stargan: Unified generative adversarial networks for multi-domain image-to-image translation. *CoRR*, abs/1711.09020, 2017.

[4] Vincent Dumoulin, Jonathon Shlens, and Manjunath Kudlur. A learned representation for artistic style. *CoRR*, abs/1610.07629, 2016.

[5] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *ArXiv e-prints*, mar 2016.

[6] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. *CoRR*, abs/1508.06576, 2015.

[7] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *ArXiv e-prints*, June 2014.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[9] Ian J. Goodfellow. NIPS 2016 tutorial: Generative adversarial networks. *CoRR*, abs/1701.00160, 2017.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[12] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. *CoRR*, abs/1611.07004, 2016.

[13] Andrej Karpathy and Fei-Fei Li. Deep visual-semantic alignments for generating image descriptions. *CoRR*, abs/1412.2306, 2014.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[15] Yijun Li, Chen Fang, Jimei Yang, Zhaowen Wang, Xin Lu, and Ming-Hsuan Yang. Universal style transfer via feature transforms. *CoRR*, abs/1705.08086, 2017.

[16] William Lotter, Gabriel Kreiman, and David D. Cox. Unsupervised learning of visual structure using predictive generative networks. *CoRR*, abs/1511.06380, 2015.

[17] X. Mao, Q. Li, H. Xie, R. Y. K. Lau, Z. Wang, and S. P. Smolley. Least Squares Generative Adversarial Networks. *ArXiv e-prints*, November 2016.

[18] Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron C. Courville. Film: Visual reasoning with a general conditioning layer. *CoRR*, abs/1709.07871, 2017.

[19] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[20] Dmitry Ulyanov, Andrea Vedaldi, and Victor S. Lempitsky. Instance normalization: The missing ingredient for fast stylization. *CoRR*, abs/1607.08022, 2016.

[21] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. *CoRR*, abs/1411.4555, 2014.

[22] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.