Applying Deep Learning/GANs to Histology Image for Data Augmentation: A General Study

Juan Pablo Garcia Martinez
MSc in Bioinformatics and Biostatistics
Final Master Thesis – Machine Learning

Director: Esteban Vegas Lozano

GSK Supervisor: Jin Yao

In collaboration with gsk

Title:	Deep Learning/GANs to Histological Image Data Augmentation
Author:	Juan Pablo Garcia Martinez
Advisor:	Esteban Vegas Lozano
Co-advisor:	Jin Yao (GSK)
Date:	Jan-2018
MSc Title:	MSc in Bioinformatics and Biostatistics
Thesis Area:	Final Master Thesis in Machine Learning
Language:	English
Keywords:	Histology, GAN, DCGAN, Data Augmentation

Abstract (in English, 250 words or less):

In medical imaging tasks, annotations are made by radiologists with expert knowledge on the data and task. Therefore, Histology images are especially difficult to collect as they are: expensive, time consuming and information can not be always disclosed for research. To tackle all these issues data augmentation is a popular solution. Data augmentation, consist of generating new training samples from existing ones, boosting the size of the dataset. When applying any type of artificial neural network, the size of the training is key factor to be successful. especially when employing supervised machine learning algorithms that require labelled data and large training examples.

We present a method for generating synthetic medical images using recently presented deep learning Generative Adversarial Networks (GANs). Furthermore, we show that generated histology images can be used for synthetic data augmentation and improve the performance of CNN for medical image classification. The GAN is a non-supervised machine learning technique where one network generates candidates (generative) and the other evaluates them (discriminative) to generate new sample like the original. In our case we will focus in a type of GAN called Deep Convolution Generative Convolutional Network (DCGAN) where the CNN architecture is used in both networks and the discriminator is reverting the process created by the generator.

Finally, we will apply this technique, for data augmentation, with two different datasets: Narrow bone and Breast tissue histology image. To check the result, we will classify the synthetic images with a pre-trained CNN with real images and labelled by specialist.

Table of contents

1. Introduction

1.1 Context and motivations

One of the greatest challenge in histology imaging domain is how to cope with the small datasets and limited number of annotated samples.

Over the last decade Deep Neural Networks have produced unprecedented performance for data classification with the sufficient data. In many realistic settings we need to achieve goals with limited datasets; in those cases, deep neural networks seem to fall short, overfitting on the training set and producing poor generalisation on the test set.

We propose that one way to build good image representations is by training Generative Adversarial Networks (GANs) [1] (Goodfellow et al., 2014) for data augmentation proposes and data privacy. This process will allow us to train better classification method as CNN or RNN. We propose and evaluate a set of constraints on the architectural topology of Convolutional GANs that make them stable to train in most settings. We name this class of architectures Deep Convolutional GANs (DCGAN).
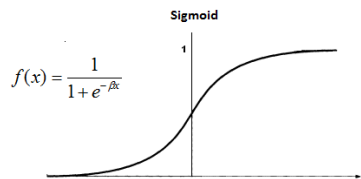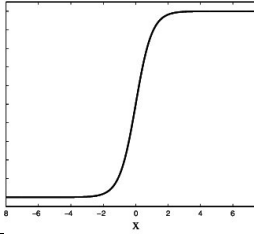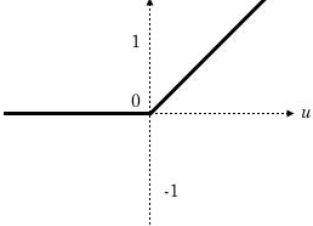
1.2 Important Definitions:

Artificial Neural Network (ANN) An ANN is based on a collection of connected units or nodes called artificial neurons. The signal at a connection between artificial neurons is a real number, and the output of each artificial neuron is computed by some non-linear function of the sum of its inputs. Artificial neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection.

Multi-Layer Perceptron (MLP) is composed of one (passthrough) input layer, one or more layers of (threshold logic unit) **TLU**s. The TLU receives one or more inputs and sums them to produce an output associated with a weight. The MLP is formed of hidden layers, and one final layer called *output layer*.

Backpropagation in each training instance, the algorithm feeds it to the network and computes the output of every neuron in each consecutive layer and measures the network's output error (difference between predicted and real output). It calculates how much of the error in each neuron contribute to the total, so we get the error gradient across all the connection weights in the network by propagating the error gradient backward in the network.

Activation Functions this helps the linear operation of inputs to a non-linear operation. If there is no activation function, all the layers can be collapsed to a single layer. Therefore, it allows ANNs to classify the non-linear sample space.

Sigmoid function	
Sigmoid function takes a real-valued number and "squashes" it into range between 0 and 1. Large negative numbers become 0 and large positive numbers become 1.	Sigmoid $f(x) = \dfrac{1}{1+e^{-\beta x}}$
Hyperbolic Tangent	

Tanh function is sigmoidal with a range from -1 to 1. The benefit is that only zero-valued inputs are mapped to near-zero so the network less likely to get "stuck" during training.	
ReLu, Leaky ReLu	$f(u) = \max(0, u)$
The **ReLU** function is $f(x)=max(0, x)$. It works very well as is very fast to compute and there is not max value. **Leaky ReLU** allow a small, positive gradient when the unit is not active. Avoid some neurons to die during the training process. $f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$	
Table 1. Activation Function	

Deep Neural Network (DNN) A deep neural network (DNN) is an **artificial neural network** (ANN) with multiple layers between the input and output layers. Deep neural networks use sophisticated mathematical modelling to process data in complex ways.

Part	Example
Initialization	He initialization
Activation Function	ReLu
Normalization	Batch Normalization
Regularization	Dropout
Optimizer	Accelerated Gradient
Learning Rate Schedule	
Table 2. Default DNN configuration	

Data Augmentation very useful for classification problems, allow us to generate new training instances. It can be generated synthetically (Randomly, GANs) or by small modification in real data as:

- Rotation
- Transposition
- Segmentation
- Patching
- Zooming

By combining these techniques, we will increase the size of our training data set. This forces to the model to be more flexible with positions, sizes, orientation, light condition or colours. This will reduce the risk of **overfitting**.

Generative Adversarial Networks (GAN) (Goodfellow et al., 2014), An approach inspired by game theory for training a model that syntheses images are known as Generative Adversarial Networks (GANs). The model consists of two networks that are trained in an adversarial process where one network generates fake images and the other network discriminates between real and fake images repeatedly.

GAN samples noise z using normal or uniform distribution and utilizes a deep network generator G to create an image x (x=G(z)). In the other hand, we add a discriminator to

distinguish whether the discriminator input is real or generated. It outputs a value D(x) to estimate the chance that the input is real.

GAN is defined as a minimax game with the following objective function.

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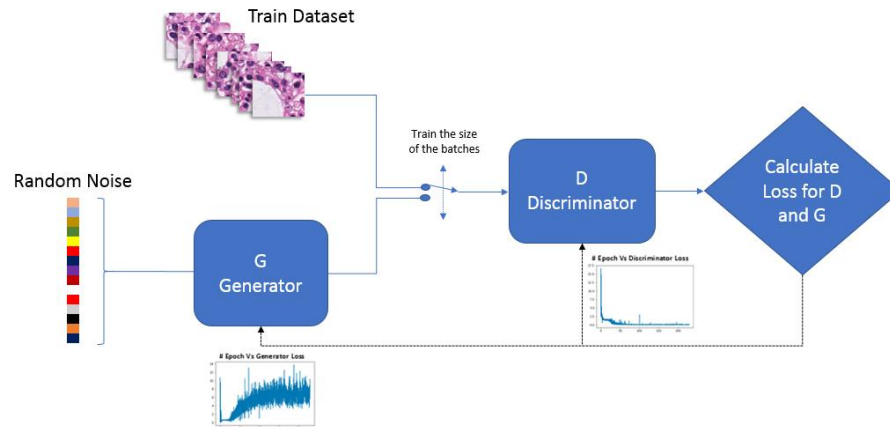

Figure 1. GAN Scheme

Convolutional Neural Network (CNN) Created from a study regarding the brain's visual cortex for image recognition. These networks are composed of an input layer, an output layer and several hidden layers, some of which are convolutional. In the first convolutional layer not all the neurons are connected to every pixel, only those in a delimited area allowing the method to focus on very specific in very simple shapes, the next layer will be able to identify complex forms, and so on.
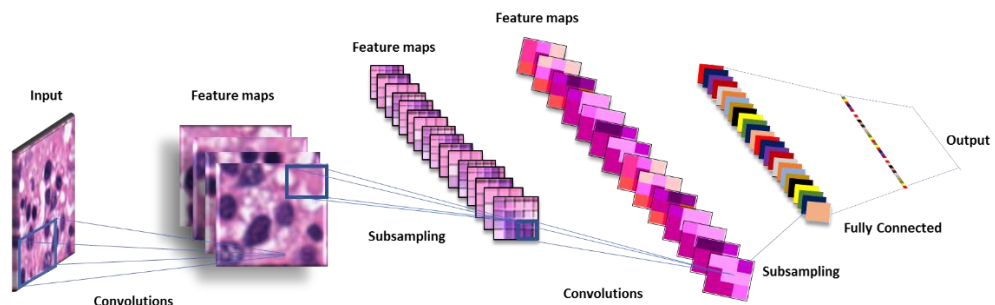

Figure 2. CNN Scheme

Deep Convolutional Generative Adversarial Network (DCGAN) are a deep learning architecture that generate outputs like the data in the training set. This model replaces the fully connected layers of the generative adversarial network model with convolution layers.
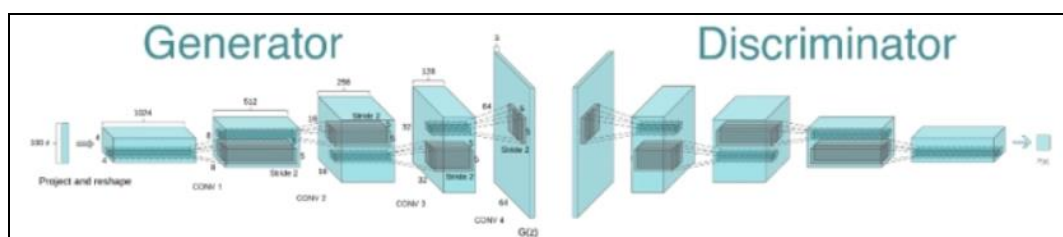

Figure 3. DCGAN Scheme

- **Convolutional Layers**

Local (sparse) connectivity. In dense layers each neuron is connected to all neurons of the previous layer (that's why it's called dense). In the convolutional layer each neuron is connected only to the small portion of the previous layer neurons.

The spatial size of the region to which the neuron is connected is called filter size (filter length in the case of 1D data like time series, and width/height in the case of 2D data like images). Stride is the size of the step with which we slide the filter over the data. The idea of local connectivity is no more than the sliding kernel. Each neuron in the convolutional layer represents and implements one position of the kernel sliding across the initial image.

- **Pooling Layers**

Pooling filters the neighbourhood region of each input pixel with some predefined aggregation function such as maximum, average, and so on. Pooling is effectively the same as convolution, but the pixel combination function is not restricted to the dot product. One more crucial difference is that pooling acts only in spatial dimension. The characteristic feature of the pooling layer is that stride usually equals the filter size.

- **Dense Layers**

The last step is to classify the input image based on the detected features. In CNNs it is done with dense layers on top of the network. This is called the classification part. It may contain several stacked, fully-connected layers, but it usually ends up with the softmax activated layer, where the number of units is equal to the number of classes. The softmax layer outputs the probability distribution over the classes for the input object.

1.3 Datasets Used

We trained DCGANs on three datasets: MNIST (example dataset), Marron Bone Histology Images and Breast Cancer Histology Images.

MNIST: Database of handwritten digits with a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centred in a fixed-size image. Especially useful for learning purposed, it is used in most of the courses, books and tutorials available now. link

Figure 4. MNIST handwritten digits

Marron Bone Histology Images: Data obtain from "Bo Hu" Github repository. This repository contains the Python implementation and datasets for the paper [13] Unsupervised Learning for Cell-level Visual Representation in Histopathology Images with Generative Adversarial Networks. link

Bone narrow is the key component of both the hematopoietic system and the lymphatic system. by producing copious amounts of blood cells. The cell lines undergoing maturation in the tissue

mostly include myeloid cells (granulocytes, monocytes, megakaryocytes, and their precursors), erythroid cells (normoblasts), and lymphoid cells (lymphocytes and their precursors).
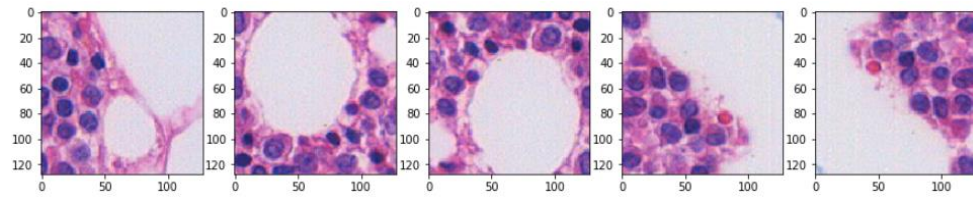

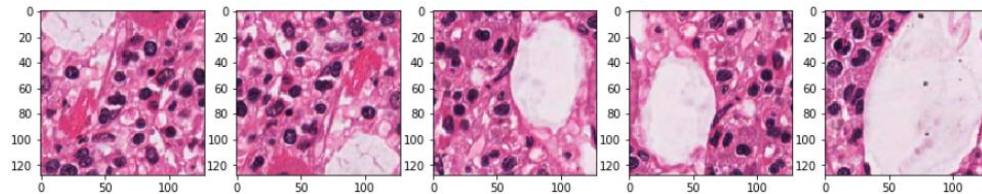**Figure 5. Marrow Bone Histology Images Patched. Positive**

Figure 6. Marrow Bone Histology Images Patched. Negative

We won't work with this data in a cell-detail level, instead we will apply our models to all some patches and segmented data to explore the possibility of working with complete images, as it is shown in the figue above **Figure 5 & 6**. This dataset it is very interesting as Marrow cell follow a clear pattern and it is a small data set that requires a lot data of augmentation.

Breast Cancer Histology Images: Data obtained from the C-BER Bioimaging Challenge 2015 Breast Histology Dataset [15]. The dataset is composed of Haematoxylin and eosin (H&E) stained breast histology microscopy and whole-slide images. Microscopy images are labelled as normal, benign, in situ and invasive carcinoma according to the predominant cancer type in each image. The annotation was performed by two medical experts and images where there was disagreement were discarded. link

The dataset is composed of an extended training set of 249 images, and a separate test set of 20 images. In these datasets, the four classes are balanced.

Details:

Microscopy images are on .tiff format and have the following specifications:
Colour model: R(ed)G(reen)B(lue)
Size: 2048 x 1536 pixels
Pixel scale: 0.42 μm x 0.42 μm
Memory space: 10-20 MB (approx.)
Type of label: image-wise
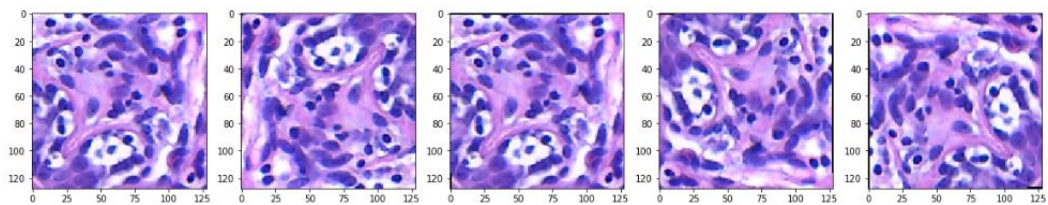
Figure 7. Histology Breast: Normal
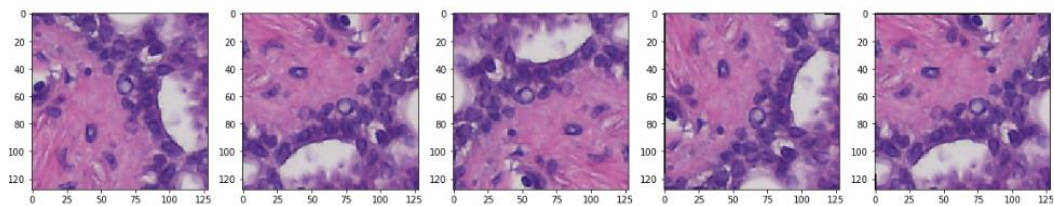
Figure 8. Histology Breast: Benign
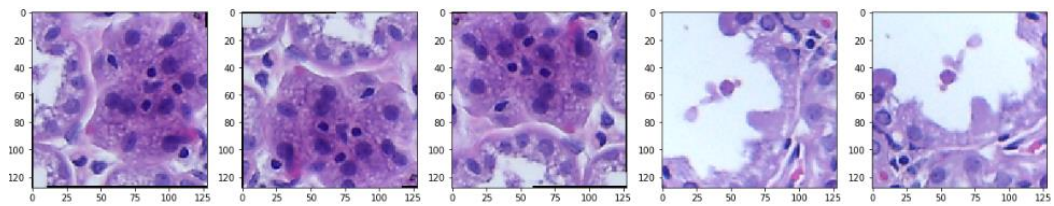
Figure 9. Histology Breast: In Situ Carcinoma
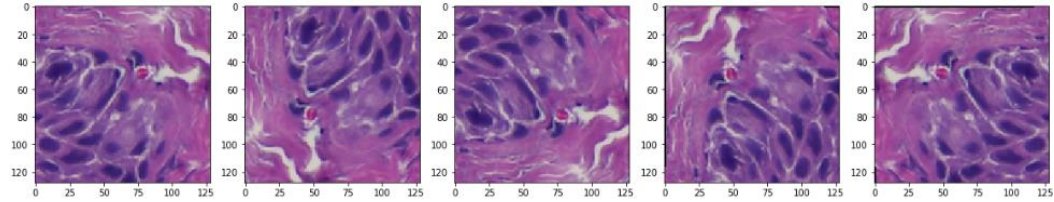
Figure 10. Histology Breast: invasive Carcinoma
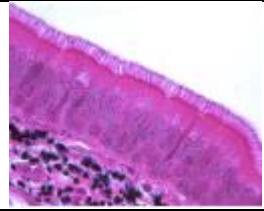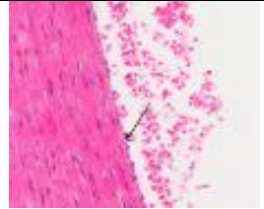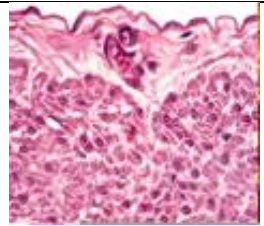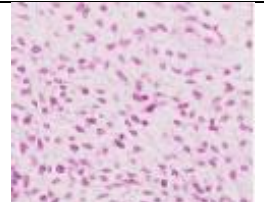
Others:

- Pokemon Tutorial link
- Celebrities Faces link
- Cat Generator link

1.4 Histological Analysis: Basics

Histological analysis is the study of the microscopic anatomy of cells, generally performed by examining thin slices of tissue that have been stained mounted on a glass slide.

There are four basic types of animal tissues: muscle tissue, nervous tissue, connective tissue, and epithelial tissue.

SUBTYPES OF TISSUE		
Epithelium	lining of glands, bowel, skin, and some organs like the liver, lung, and kidney.	
Endothelium	lining of blood and lymphatic vessels.	
Mesothelium	the lining of pleural and pericardial spaces.	
Mesenchyme	the cells filling the spaces between the organs, including fat, muscle, bone, cartilage, and tendon cells.	
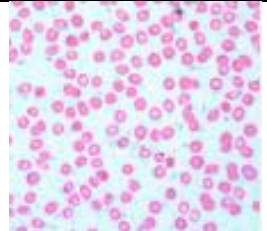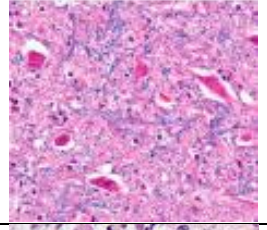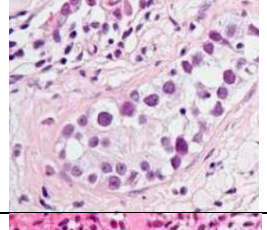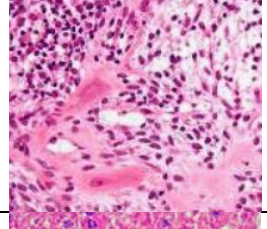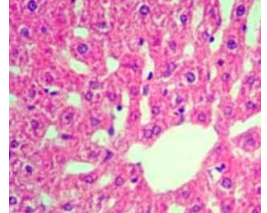
Blood cells	the red and white blood cells, including those found in lymph nodes and spleen.	
Neurons	any of the conducting cells of the nervous system.	
Germ cells	reproductive cells (spermatozoa in men, oocytes in women)	
Placenta	an organ characteristic of true mammals during pregnancy, joining mother and offspring, providing endocrine secretion and selective exchange of soluble, but not particulate, blood-borne substances through an apposition of uterine and trophoblastic vascularised parts.	
Stem cells	Stem cells are biological cells that can differentiate into other types of cells and can divide to produce more of the same type of stem cells. They are always and only found in the multicellular organisms.	

Table 3. Histology: Subtypes of Tissues

The standard tissue stain is composed of haematoxylin and eosin. The haematoxylin stains nucleic acids blue, the eosin stains most proteins pink, clear areas represent water, carbohydrate, lipid, or gas. Nuclei will always stain blue with the haematoxylin. The cytoplasm of cells will stain according to its composition. The strong supporting proteins around the cells will stain pink, and you will be able to see their texture. In all but the most regular of tissues, between the fibres will be looser areas where there is a preponderance of ground substance. It is composed of mucopolysaccharide plus complex carbohydrates. The protein component of the mucopolysaccharide usually imparts a weak pink colour.

Histopathology differs from histological analysis in that it works specifically with diseased tissue and is broadly used in the diagnosis of cancer and other diseases.

2. Task Planning

Date	Milestone	Details
02/10/2018	Project Start	
02/10/2018	Collaboration with GSK & Training	
17/10/2018	Topic: Applying GANs on tumor histological images	
28/10/2018	Planning Timeline & Access to Unix Server	
10/11/2018	Test Phase & Image (Delay)	Delayed
27/11/2018	Training model (No supervised)	Same
10/12/2018	Testing diverse types of GAN	We will focus in DCGAN
25/12/2018	Classifications method and labelling methods	Same
28/12/2018	Drafting of the project	Same
03/01/2019	Revision & Changes	Same
12/01/2019	Last version of the project	Same
20/01/2019	Public Presentation	Same
23/01/2018	Project End	Same

Table 4. Task Planning

Timeline:

Figure 11. Project timeline

nts:

As this is a project in collaboration with GSK, access to the UNIX server was required to be able to run methods with a very high hardware demand, GPU is essential for this project.

Since the beginning of this project one of the hardest task was setting up the environment needed to be able to apply our own method with our own data. Most of the tutorial and courses available are

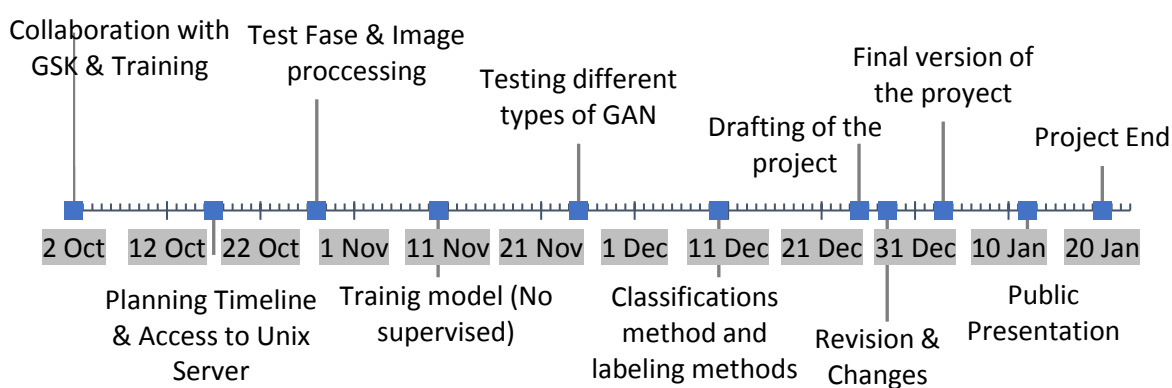using pre-treated datasets already in TensorFlow format, so when you try to import your own real data with those models lots of issues arises.

Training, Courses and Tutorial:

- "Deep Learning and Computer Vision A-Z™: OpenCV, SSD & GANs" **Udemy**: link
- "Deep Learning: GANs and Variational Autoencoders" **Udemy**: link
- "Learning Computer Vision with TensorFlow" **Udemy**: link
- "Deep Learning in Python" **DataCamp**: link
- "Convolutional Neural Networks for Image Processing" **DataCamp**: link
- "LEARNING PATH: TensorFlow: Computer Vision with TensorFlow" **Udemy**: link
- "Complete Guide to TensorFlow for Deep Learning with Python" **Udemy**: link

3. Materials and Software Details:

3.1 Basic Concepts

What is TensorFlow?
It is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with dedicated support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

What is Keras?
Keras is a high-level neural networks API, written in Python and capable of running on top of **TensorFlow**, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Runs seamlessly on CPU and GPU. Keras is integrated in TensorFlow, therefore installation is not required as it can be used from TensorFlow package as backend.

What is GPU and Cuda?
A graphics processing unit (**GPU**) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. Their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms that process large blocks of data in parallel.

CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers can dramatically speed up computing applications by harnessing the power of GPUs.

3.2 Technical details Software-Hardware

For this project we will use python as programming language. Python is a popular and powerful interpreted language. Unlike R, Python is a complete language and platform that you can use for both research and development and developing production systems.

The packages that we will use for applying these deep learning techniques are TensorFlow and Keras (with TensorFlow as backend). Always is easier to use Keras as it is a high-level API, but it

doesn't allow enough customisation for applying GANs. Therefore, we will use Keras for classification methods as CNN and TensorFlow for DCGAN.

For machine learning and data processing we will use the following packages:

- **Scipy:** it builds on the NumPy array object and is part of the NumPy stack. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers.
- **Numpy:** It stands for 'Numerical Python'. It is a library consisting of multidimensional array objects and a collection of routines for processing of array.
- **Matplotlib:** is probably the single most used Python package for 2D-graphics.
- **Pandas:** easy-to-use data structures and data analysis tools for the Python programming language.
- **Sklearn:** It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

For image analysis and manipulation:

- **PIL (Pillow):** is a Python imaging library, which adds support for opening, manipulating, and saving images.
- **CV2 (OpenCV):** the library can take advantage of multi-core processing. Enabled with OpenCL, it can take advantage of the hardware acceleration of the underlying heterogeneous compute platform.
- **Imageio**: is a Python library that provides an easy interface to read and write a wide range of image data, including animated images, volumetric data, and scientific formats.

We will connect to the GSK Unix Server through SSH using the software: MobaXterm, toolbox for remote computing. SSH allows you to connect to your server securely and perform Linux command-line operations, in our case RH7. We will use one of the servers available specially this one is ready to use several GPUs.

Details:

Graphic Cards:

By executing the code nvidia-smi we can check the 8 graphic cards with GPU available:

Driver Version:	390.12
GPU 1	0 Tesla P100-PCIE…
GPU 2	1 Tesla P100-PCIE…
GPU 3	2 Tesla P100-PCIE…
GPU 4	3 Tesla P100-PCIE…
GPU 5	4 Tesla P100-PCIE…
GPU 6	5 Tesla P100-PCIE…
GPU 7	6 Tesla P100-PCIE…
GPU 8	7 Tesla P100-PCIE…
Table 5. nvidia-smi	

To make easier working with so many different tests we will use **Anaconda,** Python data science platform**.** that allows us to use different environments for different purposes.

Relevant Anaconda Packages Version in our environment:

To check the Anaconda, we will use the command: `conda -V`
conda 4.5.11

We can check all the packages version by running the command:
conda list

python	3.6.6	tensorflow-gpu	1.4.1 (downgraded)
cudatoolkit	8.0	tensorflow-gpu-base	1.4.1
cudnn	7.0.5 (downgraded)	tensorflow-tensorboard	1.5.1
scikit-learn	0.20.0	imageio	2.4.1
scipy	1.1.0	keras-applications	1.0.6
six	1.11.0	keras-base	2.2.4
sqlite	3.25.2	keras-gpu	2.2.4
tensorflow-base	1.4.1 (downgraded)	keras-preprocessing	1.0.5

4. Guide to install Tensorflow-GPU in Jupyter Notebook.

Jupyter Notebook is not a pre-requisite for using TensorFlow (or Keras), bit it makes the learning process much easier as allow you to run the codes by sections and add notes in markdown.

There is no need to install Keras anymore as it is included in the TensorFlow package. tf.keras in your code instead.

1. **Download and Install Anaconda:**

   Download the installation file using the command *wget*.
   `curl -O https://repo.continuum.io/archive/Anaconda3-5.3.0-Linux-x86_64.sh`
   Execute the installation file:
   `bash ~/Downloads/Anaconda3-5.3.0-Linux-x86_64.sh`

   Setup the path in the bashrc file by adding the following command line:

```
__conda_setup="$(CONDA_REPORT_ERRORS=false ' /Software/Anaconda3/bin/conda' shell.bash
hook 2> /dev/null)"
if [ $? -eq 0 ]; then
  \eval "$__conda_setup"
else
  if [ -f " /Software/Anaconda3/etc/profile.d/conda.sh" ]; then
    . " Software/Anaconda3/etc/profile.d/conda.sh"
    CONDA_CHANGEPS1=false conda activate base
  else
    \export PATH=" /Software/Anaconda3/bin:$PATH"
  fi
fi
unset __conda_setup
```

   We need to reset the bashrc to use the paths previously added to the bashrc file:
   `source ~/.bashrc`

   Check installation by using an anaconda command:

```
conda list
sha256sum Anaconda3-5.3.0-Linux-x86_64.sh
get: cfbf5fe70dd1b797ec677e63c61f8efc92dad930fd1c94d60390bb07fdc09959  Anaconda3-
5.3.0-Linux-x86_64.sh
Install: bash Anaconda3-5.3.0-Linux-x86_64.sh
```

2. Setup and update the Cuda driver for GPU:

To redirect the cuda driver to the right path this should be added in the *bashrc* file:

```
export PATH=/ local/cuda-9.1/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/local/cuda-9.1/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

Cuda drivers can be checked by using the command:

```
nvcc -version
```

Create an environment with python version 3.6

Now we need to setup an environment with python 3.6 as is the one compatible with
TensorFlow.
```
conda create -n [my-env-name] python == 3.6
```

This environment can be activated by using the command:
```
conda activate [my-env-name]
```

Install the all packages included at the end of the section 3 by using the command:

In our case as we have installed the version of the cuda driver 9.1 we will need to install the
compatible version of tensorflow which is the version 1.4.1.

```
conda install tensorflow-gpu==1.4.1
```

Because of the characteristics of our server we need to downgrade the cudnn to the version 7.0.5
instead the version 7.2.0

```
conda install cudnn == 7.0.5
```

Check installation:

After calling python in the terminal, the following code can be used to check the installation:
```
python
>>> import tensorflow as tf
>>> tf.__version__ # My answer is '1.4.1'
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))
```

Start a tmux session:

What is tmux?
Tmux is a terminal multiplexer, allowing a user to access multiple separate terminal sessions
inside a single terminal window or remote terminal session. It is useful for dealing with multiple
programs from a command-line interface, and for separating programs from the Unix shell that

16

started the program. It is relevant to this project as we will need to use when long run time codes are required, so the server will work in our code even when we are not connected to the server.

How to use jupyter notebook on HPC
- start a tmux session named `notebook`
  tmux new -s notebook
- start a jupyter backend
  jupyter notebook --no-browser
- check the output of the command and find the port number of the URL
  e.g. https://[all ip addresses on your system]:8889/ (Example)
- Go to the URL in your web browser where <hostname> is the server name your notebook is running on and <portname> is the port name displayed in step 3
  https://<hostname>:<portname>
- Enter Ctrl-b d to exit tmux session.
- Go back to the started session named `notebook`
  tmux a -t notebook

  From here we will be able to run our Jupyter Notebook and keep it open by:

  First, Activate our environment:

```
conda activate myev
```

  Secondly, call the jupyter notebook to open a port with where it is accessible from our local machine.

```
Jupyter notebook
```

  Now we will be able to access to jupyter by opening the address provided in our local machine's browser:

```
https://address:10000/tree/Codes
```

3. **Check the Installation**

   We can check the code is working by doing several checks in our code:

   Import the package TensorFlow in our case it will be TensorFlow GPU

```
import tensorflow as tf
```

   We call to use the first GPU only and define some constant and operation

```
with tf.device('/gpu:0'):
  a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
  b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
  c = tf.matmul(a, b)
```

   We can open a session by using the command Session that we will name as sess

```
with tf.Session() as sess:

  print (sess.run(c))
```

Check output!

The following code tells if the GPU is available:

```
tf.test.is_gpu_available
            tf.test.gpu_device_name  # returns the name of the gpu device
            tf.test.is_gpu_available(
                cuda_only=False,
                 min_cuda_compute_capability=None)
```

Output: True

The following code tells if the name of all GPU instance:

```
from tensorflow.python.client import device_lib
print (device_lib.list_local_devices())
```

5. Narrow Bone Data DCGAN

As a first test with real data we will use the images for the **[14]** Cell-level Visual Representation in Histopathology Images with Generative Adversarial Networks as indicated in the Introduction. This Images are simple, it is a small dataset and the resolution is good enough for a first test using real data. Instead working in a Cell-level analysis we will work with the full images previously patched and we will apply some data augmentation techniques.

The images have been obtained in the biopsy sections of bone marrow, the abnormal cellular constitution indicates the presence of blood disease as bone narrow is the key component of both the hematopoietic system and the lymphatic system. Too many granulocytes precursors such as myeloblasts indicate the presence of chronic myeloid leukaemia. Having large, abnormal lymphocytes heralds the presence of lymphoma.
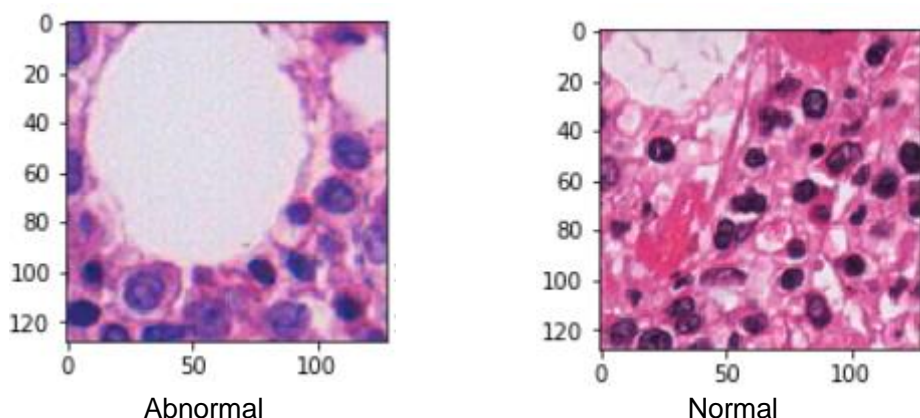

Abnormal   Normal

Figure 12. Normal and Abnormal Narrow Bone Tissue Comparison

5.1   Patching and Data Augmentation

For this example, we are using two different folders with two different file types as PNG and JPG. So, we will use two different codes for negative and positive samples and we will append both

matrix of samples into the same list. We will create a list with codes 0 (negatives) and 1 (positives) appended in another variable.

In this data set we will find to different folders with all positive and negative images for each. There are 11 negatives images, size (1200x1200) and 29 positive Images, size (1532x834).

Packages:

```
import numpy as np
import matplotlib.pyplot as plt
from glob import glob
from PIL import Image
import cv2
import os
```

- **Data Exploration**

   For positive images we are using jpg images. We can check the size of the image.

```
im = Image.open("../dataset_A/original/positive_images/20171005091805.jpg")
imgwidth, imgheight = im.size
imgwidth, imgheight, im.size
```
Output: (1532, 834, (1532, 834))

   For negative images we are using jpg images. We can check the size of the image.

```
im1=Image.open("../dataset_A/original/negative_images/BM_GRAZ_HE_0020_01.png")
imgwidth1, imgheight1 = im1.size
imgwidth1, imgheight1, im1.size
```
Output: (1200, 1200, (1200, 1200))

- **Defining the crop size for the patching.**

   we can adjust the size the images as much as possible by using the formula below, the goal is to approach the size of the patches as much as possible to 128x128 px. For this purpose, we need to parameter "n_patch_div_2" that defines the zooming that will be applied to each crop area from the original image.

```
n_patch_div_2 = 5
height = np.int_(np.round_(((imgwidth * imgheight)**0.5)/n_patch_div_2, 0))
width = np.int_(np.round_(((imgwidth * imgheight)**0.5)/n_patch_div_2, 0))
path = "../Images/"
print(height, width)
```

- **Image Segmentation: cancelling background and simplification.**

   We are working the RGB (Red Green Blue) Images, where colours are represented in terms of their red, green, and blue components. The function used is cv2.threshold. First argument is the source image, which should be a grayscale image. Second argument is the threshold value which is used to classify the pixel values. Third argument is the maxVal which represents the value to be given if pixel value is more than (sometimes less than) the threshold value. OpenCV provides different level of thresholding and it is decided by the fourth parameter of the function. If pixel value is greater than a threshold value, it is assigned one value, else it is assigned another value.

```
x = 1000
img1 = Train_Data[x]
blur = cv2.GaussianBlur(img1,(7,7),1)
# threshold = 85
th1 = cv2.threshold(blur,85,255,cv2.THRESH_BINARY)
plt.imshow(th1)
```

We get to the right value by trial and error to eliminate as much background colour as possible, in the next figure it is shown clearly how the chosen value is adjusted to the nuclei:

Example of threshold exploration.

| th = 40 | th = 85 | th = 125 |

Threshold applied to real data.

| Real Negative Image | Threshold = th | Segmented Image th=85 |
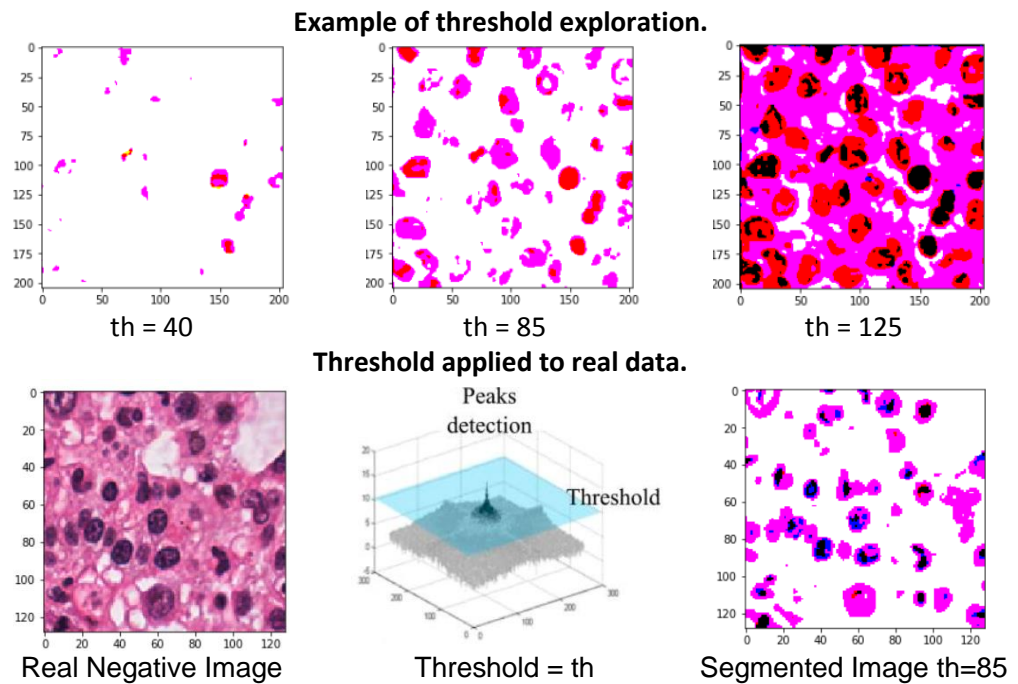
Figure 13. Applying threshold to histology images

- **Patching and Data Augmentation**

Process in the code:

1. Defining start and final position for the cropping. Using the iteration "i" for height and "j" for width.
2. Defining the image as PAL type applying the 'RGB' format.
3. Setting up the box variable that we will use later in the function crop for the patching.
4. As we need to consolidate our database we will resize all the images to 128x128 pixels
5. Filtering:
   - Applying a filter by variance of the matrix we will discard no meaningful images. Taking the value: 1200 for the variance of the matrix. This figure has been chosen by experimentation.
6. Data Augmentation:
   - Transpose left to right
   - Transpose top to bottom

```
Train_Data = []
y_data =[]
data_dir = "../dataset_A/original/positive_images/"
files = glob(os.path.join(data_dir, '*.jpg'))
```

```python
# Positive
for myFile in files:
    im = Image.open(myFile)
    for i in range(0,imgheight-height,height):
        for j in range (0, imgwidth-width, width):
            a = im.convert('RGB')
            j1 =  min(0, imgwidth - j+width)
            i1 =  min(0, imgheight - j+height)
            box = (j, i, j+width, i+height)
            # Crop images 226x226 px
            a = a.crop(box)
            # Resize to 128x128 px
            a = a.resize((128, 128), Image.ANTIALIAS)
            # filter images by applying requirement. Variance
            #of the matrix has been calculated manually
            if np.var(a) > 1200:
                b = a.transpose(Image.FLIP_LEFT_RIGHT)
                b = b.transpose(Image.FLIP_TOP_BOTTOM)
                a = np.array(a)
                a = cv2.GaussianBlur(a,(7,7),1)
                _, a = cv2.threshold(a,85,255,cv2.THRESH_BINARY)
                Train_Data.append (a)
                y_data.append (1)
                #Transpose images for data augmentation
                b = np.array(b)
                b = cv2.GaussianBlur(b,(7,7),1)
                _, b = cv2.threshold(b,85,255,cv2.THRESH_BINARY)
                Train_Data.append (b)
                y_data.append (1)
print(len(y_data))

# Negative
data_dir = "../dataset_A/original/negative_images/"
files = glob(os.path.join(data_dir, '*.png'))
files
for myFile in files:
    im = Image.open(myFile)
    for i in range(0,imgheight1-height1,height1):
        for j in range (0, imgwidth1-width1, width1):
            a = im.convert('RGB')
            j1 =  min(0, imgwidth1 - j+width1)
            i1 =  min(0, imgheight1 - j+height1)
            box = (j, i, j+width1, i+height1)
            a = a.crop(box)
            a = a.resize((128, 128), Image.ANTIALIAS)
            if np.var(a) > 1200:
                b = a.transpose(Image.FLIP_LEFT_RIGHT)
                b = b.transpose(Image.FLIP_TOP_BOTTOM)
                a = np.array(a)
                a = cv2.GaussianBlur(a,(7,7),1)
                _, a = cv2.threshold(a,85,255,cv2.THRESH_BINARY)
                Train_Data.append (a)
                y_data.append (0)
                b = np.array(b)
                b = cv2.GaussianBlur(b,(7,7),1)
                _, b = cv2.threshold(b,85,255,cv2.THRESH_BINARY)
```

```
              Train_Data.append (np.array(b))
              y_data.append (0)
        print(len(y_data))
```

Note: we have created a total of 1328 patches after data augmentation. 984 positives and 343 negatives. This is not great for create a difference between negatives and positives tissues, but we will try to train all the patches in the same train session to explore the output as first approach to this technique.

- **Image Exploration**

Spit dataset in positive images and negative:

```
data_positive = Train_Data[0:984]
data_negative = Train_Data[985:1328]
```

Positive Images Visualisation:

```
fig=plt.figure(figsize=(25, 25))
columns = 17
rows = 20
for i in range(1, columns*rows +1):
        img = data_positive[i]
        fig.add_subplot(rows, columns, i)
        plt.imshow(img)
plt.show()
```
Output:
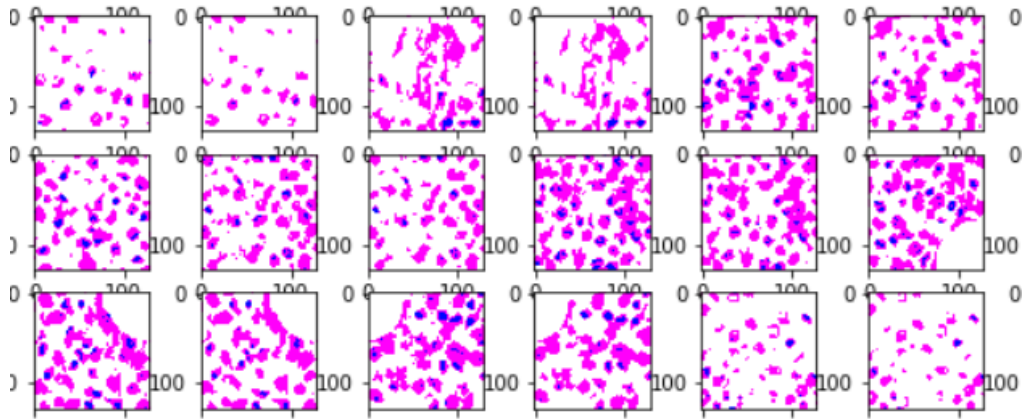
Negative Images Visualisation:

```
fig=plt.figure(figsize=(25, 25))
columns = 17
rows = 20
for i in range(1, columns*rows +1):
        img = data_negative[i]
        fig.add_subplot(rows, columns, i)
        plt.imshow(img)
plt.show()
```
Output:
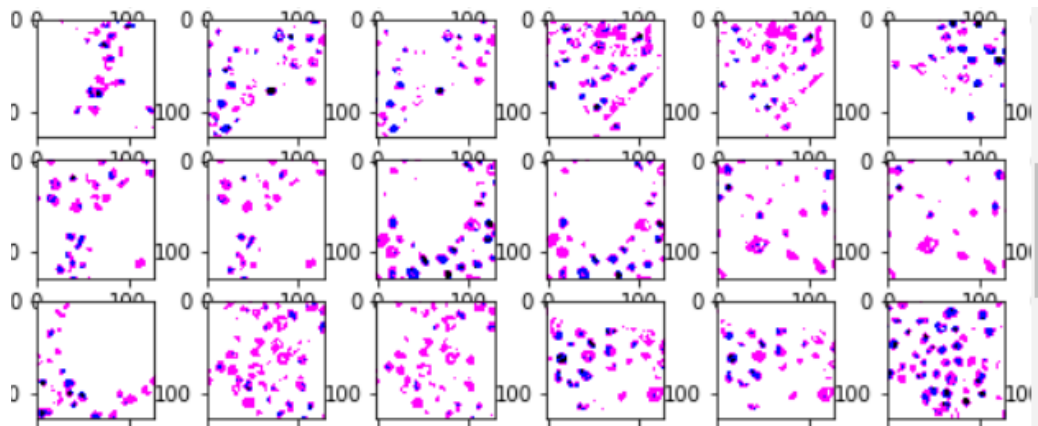
- **Saving the Database**

  Saving database in an external file to export it to a different notebook by using the package pickle.

```
import pickle

pickle_out = open("Train_Data.pickle", "wb")
pickle.dump(Train_Data, pickle_out)
pickle_out.close()

pickle_out = open("y_data.pickle", "wb")
pickle.dump(y_data, pickle_out)
pickle_out.close()

pickle_in = open("Train_Data.pickle", "rb")
Train_Data = pickle.load(pickle_in)
pickle_in = open("y_data.pickle", "rb")
y_data = pickle.load(pickle_in)

plt.imshow(Train_Data[901])
```

Output:
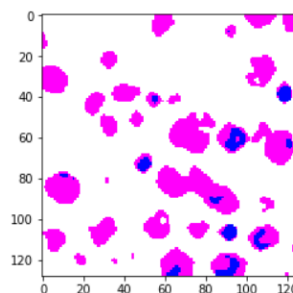
5.2   Applying DCGAN

- Loading Packages

```
import pickle
import os
```

```
from glob import glob
from matplotlib import pyplot as plt
%matplotlib inline
from PIL import Image
import numpy as np
```

- **Load data** from the previous step. We will load the treated data performed in the notebook for "Data Preparation and Augmentation". For doing so we will use the function *pickle.load* as below:

```
pickle_in = open("Train_Data.pickle", "rb")
Train_Data = pickle.load(pickle_in)
```

- **Data normalisation: from 0 to 255 to from -1 to 1.**

  This is one of trickiest part of the code when we are trying to apply the model to our own data, as most of the training examples that can be found only refer to the MNIST example, which is not very meaningful as it is pre-treated data. In this section we try to point out how to set the batches, which is not needed when working with the MNIST or CIFAR-10 examples.

  The code below will group sections of the matrix to split the database by batches, which is a requirement to apply DCGAN in most of the cases. This formula will be fixed to our database and the only variables that will be required would be the batch size "batch_size" as the idea is change this variable to optimize this process as much as possible.

  As in the future we will use a tanh activation function or that will need to be normalised from -1 to 1. Therefore, we will centralise the data at the end of this step by applying -0.5.

```
def get_batches(batch_size):
    shape = len(Train_Data), IMAGE_WIDTH, IMAGE_HEIGHT,
    data = Train_Data
    """
    Generate batches
    """
    current_index = 0
    while current_index + batch_size <= shape[0]:
        data_batch = data[current_index:current_index + batch_size]
        current_index += batch_size
        yield data_batch
```

- **Create the model inputs: Placeholders**

  This is a very straightforward step in our code. Placeholders are a TensorFlow function that works as bridge between our NumPy array data (float32) and TensorFlow flow format (tensors).

```
def model_inputs(image_width, image_height, image_channels, z_dim):
    """
    Create the model inputs
    """
    inputs_real = tf.placeholder(tf.float32, shape=(None, image_width, image_height,
image_channels), name='input_real')
```

```
inputs_z = tf.placeholder(tf.float32, (None, z_dim), name='input_z')
learning_rate_G = tf.placeholder(tf.float32, name='learning_rate')
learning_rate_D = tf.placeholder(tf.float32, name='learning_rate')


return inputs_real, inputs_z, learning_rate_G, learning_rate_D
```

- **Discriminator**

We are going to use a TensorFlow variable scope when defining this network. This helps us in the training process later, so we can reuse our variable names for both the discriminator and the generator.

The discriminator network consists of four convolutional layers. For every layer of the network, we are going to perform a convolution, then we are going to perform batch normalization to make the network faster and more accurate and finally, we are going to perform a Leaky ReLu. Batch normalization allows us to normalize the input layer by adjusting and scaling the activations.

```python
def discriminator(images, reuse=False):
    """
    Create the discriminator network
    """
    alpha = 0.2
    # Input layer 128*128*3 --> 64x64x64
    with tf.variable_scope('discriminator', reuse=reuse):
        # using 4 layer network as in DCGAN Paper

        # Conv 1
        conv1 = tf.layers.conv2d(images,filters = 64, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm1 = tf.layers.batch_normalization(conv1, training = True, epsilon = 1e-5)
        conv1_out = tf.nn.leaky_relu(batch_norm1, alpha=alpha)

        # Conv 2
        conv2 = tf.layers.conv2d(conv1_out,filters = 128, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm2 = tf.layers.batch_normalization(conv2, training = True, epsilon = 1e-5)
        conv2_out = tf.nn.leaky_relu(batch_norm2, alpha=alpha)

        # Conv 3
        conv3 = tf.layers.conv2d(conv2_out,filters = 256, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm3 = tf.layers.batch_normalization(conv3, training = True, epsilon = 1e-5)
        conv3_out = tf.nn.leaky_relu(batch_norm3, alpha=alpha)

        # Conv 4
        conv4 = tf.layers.conv2d(conv3_out,filters = 512, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm4 = tf.layers.batch_normalization(conv4, training = True, epsilon = 1e-5)
        conv4_out = tf.nn.leaky_relu(batch_norm4, alpha=alpha)

        # Conv 5
```

```
        conv5 = tf.layers.conv2d(conv4_out,filters = 1024, kernel_size = [5,5],
                        strides = [2,2], padding="SAME",
                        kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm5 = tf.layers.batch_normalization(conv5, training = True, epsilon = 1e-5)
        conv5_out = tf.nn.leaky_relu(batch_norm5, alpha=alpha)

        # Flatten
        flatten = tf.reshape(conv5_out, (-1, 8*8*1024))

        # Logits
        logits = tf.layers.dense(flatten, 1)

        # Output
        out = tf.sigmoid(logits)

        return out, logits
```

- **Generator**

This network consists of four deconvolutional layers. In here, we are doing the same as in the discriminator, just in the other direction. First, we take our input, called Z, and feed it into our first deconvolutional layer. Each deconvolutional layer performs a deconvolution and then performs batch normalization and a leaky ReLu as well. Then, we return the tanh activation function.

```
def generator(z, out_channel_dim, is_train=True):
    """
    Create the generator network
    """
    alpha = 0.2

    with tf.variable_scope('generator', reuse=False if is_train==True else True):
        # First fully connected laye First FC layer --> 8x8x1024
        fc1 = tf.layers.dense(z, 8*8*1024)
        # Reshape it
        fc1 = tf.reshape(fc1, (-1, 8, 8, 1024))
        # Leaky ReLU
        fc1 = tf.nn.leaky_relu(fc1, alpha=alpha)

        # Transposed conv 1 --> BatchNorm --> LeakyReLU
        # 8x8x1024 --> 16x16x512

        trans_conv1 = tf.layers.conv2d_transpose(inputs = fc1, filters = 512, kernel_size = [5,5],
                            strides = [2,2], padding = "SAME",
                            kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv1 = tf.layers.batch_normalization(inputs = trans_conv1,
                                    training=is_train,
                                    epsilon=1e-5)
        trans_conv1_out = tf.nn.leaky_relu(batch_trans_conv1, alpha=alpha)

        # Transposed conv 2 --> BatchNorm --> LeakyReLU
        # 16x16x512 --> 32x32x256

        trans_conv2 = tf.layers.conv2d_transpose(inputs = trans_conv1_out, filters = 256, kernel_size = [5,5],
                            strides = [2,2], padding = "SAME",
                            kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv2 = tf.layers.batch_normalization(inputs = trans_conv2, training=is_train,
                                    epsilon=1e-5)
        trans_conv2_out = tf.nn.leaky_relu(batch_trans_conv2, alpha=alpha)

        # Transposed conv 3 --> BatchNorm --> LeakyReLU
```

```
    # 32x32x256 --> 64x64x128

    trans_conv3 = tf.layers.conv2d_transpose(inputs = trans_conv2_out, filters = 128, kernel_size = [5,5],
                            strides = [2,2], padding = "SAME",
                            kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
    batch_trans_conv3 = tf.layers.batch_normalization(inputs = trans_conv3, training=is_train,
                                epsilon=1e-5)
    trans_conv3_out = tf.nn.leaky_relu(batch_trans_conv3, alpha=alpha)

    # Transposed conv 4 --> BatchNorm --> LeakyReLU
    # 64x64x128 --> 128x128x64

    trans_conv4 = tf.layers.conv2d_transpose(inputs = trans_conv3_out, filters = 64, kernel_size = [5,5],
                            strides = [2,2], padding = "SAME",
                            kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
    batch_trans_conv4 = tf.layers.batch_normalization(inputs = trans_conv4, training=is_train,
                                epsilon=1e-5)
    trans_conv4_out = tf.nn.leaky_relu(batch_trans_conv4, alpha=alpha)

    # Transposed conv 5 --> tanh
    # 128x128x64 --> 128x128x3
    # Output layer
    trans_conv5 = tf.layers.conv2d_transpose(inputs = trans_conv4_out, filters = 3, kernel_size = [5,5],
                            strides = [1,1], padding = "SAME",
                            kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))

    out = tf.tanh(trans_conv5, name="out")

    return out
```

- **Loss Functions**

Rather than just having a single loss function, we need to define three: The loss of the generator, the loss of the discriminator when using real images and the loss of the discriminator when using fake images. The sum of the fake image and real image loss is the overall discriminator loss.

The lower the loss, the better a model. The loss is calculated on training and validation and its interpretation is how well the model is doing for these two sets. Unlike accuracy, loss is not a percentage. It is a summation of the errors made for each example in training sets. In our case we will use a sigmoid cross entropy, it measures the probability error in discrete classification tasks in which each class is independent and not mutually exclusive. For instance, one could perform multilabel classification.

$$\text{Formulation:} \quad \max(x, 0) - x * z + \log(1 + \exp(-abs(x)))$$

```
def model_loss(input_real, input_z, out_channel_dim):
    """
    Get the loss for the discriminator and generator
    """

    label_smoothing = 0.9

    g_model = generator(input_z, out_channel_dim)
    d_model_real, d_logits_real = discriminator(input_real)
    d_model_fake, d_logits_fake = discriminator(g_model, reuse=True)

    d_loss_real = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_real,
                            labels=tf.ones_like(d_model_real) * label_smoothing))
```

```
d_loss_fake = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                        labels=tf.zeros_like(d_model_fake)))

d_loss = d_loss_real + d_loss_fake
g_loss = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                        labels=tf.ones_like(d_model_fake) * label_smoothing))

return d_loss, g_loss
```

d_loss (discriminator loss) is the sum of loss for real and fake images.
d_loss_real is the loss when the discriminator predicts an image is fake, when it was a real.
d_loss_fake is the loss when the discriminator predicts an image is real, when it was a fake.
d_logits_real and labels are all 1 (since all real data is real).
d_logits_fake all the label are 0.
g_loss (generator loss) uses the d_logits_fake from the discriminator.
label smoothing: reduce the labels slightly to help the discriminator generalize better.

- **Adam Optimizer**

Rather than just having a single loss function, we need to define three: The loss of the generator, the loss of the discriminator when using real images and the loss of the discriminator when using fake images. The sum of the fake image and real image loss is the overall discriminator loss.

In our case unless most of the examples that can be found we will use two different learning rates.

```
def model_opt(d_loss, g_loss, learning_rate_G, learning_rate_D, beta1):
    """
    Get optimization operations
    """
    t_vars = tf.trainable_variables()
    d_vars = [var for var in t_vars if var.name.startswith('discriminator')]
    g_vars = [var for var in t_vars if var.name.startswith('generator')]

    # Optimize
    with tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
        d_train_opt  =  tf.train.AdamOptimizer(learning_rate_D,  beta1=beta1).minimize(d_loss,
var_list=d_vars)
        g_train_opt  =  tf.train.AdamOptimizer(learning_rate_G,  beta1=beta1).minimize(g_loss,
var_list=g_vars)

    return d_train_opt, g_train_opt
```

- **Training**

In the last step of our preparation, we are writing a small function to display the generated images in the notebook for us, using the matplotlib library. So we can track the generation of images along the process.

```
def show_generator_output(sess, n_images, input_z, out_channel_dim):
    """
    Show example output for the generator
    """
    z_dim = input_z.get_shape().as_list()[-1]
```

```
    example_z = np.random.uniform(-1, 1, size=[n_images, z_dim])

    samples = sess.run(
        generator(input_z, out_channel_dim, False),
        feed_dict={input_z: example_z})

    fig=plt.figure(figsize=(15, 10))
    columns = 5
    rows = 1
    for i in range(1, columns*rows +1):
        img = samples[i]
        fig.add_subplot(rows, columns, i)
        plt.imshow(np.array((((img)/2)+0.5), np.float32))
        plt.show()
```

Now, we just get our inputs, losses and optimizers which we defined before, call a TensorFlow session and run it batch per batch.

```
batch_size = 16
z_dim = 100
learning_rate_D = .00005
learning_rate_G = .0002
beta1 = 0.5
IMAGE_WIDTH = 128
IMAGE_HEIGHT = 128
shape = len(Train_Data), IMAGE_WIDTH, IMAGE_HEIGHT, 3
```

Defining variables for the training session. In this step we will make use of the placeholder created in the previous step:

```
# One off as reuse is false
input_real, input_z, _, _ = model_inputs(shape[1], shape[2], shape[3], z_dim)
d_loss, g_loss = model_loss(input_real, input_z, shape[3])
d_opt, g_opt = model_opt(d_loss, g_loss, learning_rate_G, learning_rate_D, beta1)
```

Opening the TensorFlow session and starting to train the method.

```
epochs = 960
epoch_i = 0
steps = 0
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch_i in range(epochs):
        for batch_images in get_batches(batch_size):
            batch_images = batch_images * 2
            steps += 1

            batch_z = np.random.uniform(-1, 1, size=(batch_size,z_dim))
            _ = sess.run(d_opt, feed_dict={input_real: batch_images, input_z: np.array(batch_z, np.float32)})
            _ = sess.run(g_opt, feed_dict={input_real: batch_images, input_z: np.array(batch_z, np.float32)})
```

```
        if steps % 83 == 0:

            # At the end of every 83 epochs, get the losses and print them out
            train_loss_d = d_loss.eval({input_z: batch_z, input_real: batch_images})
            train_loss_g = g_loss.eval({input_z: batch_z})
            show_generator_output(sess, 6, input_z, shape[3])

            print("Epoch {}/{}...".format(epoch_i+1, epochs),
                "Discriminator Loss: {:.4f}...".format(train_loss_d),
                "Generator Loss: {:.4f}".format(train_loss_g))
```

Output:
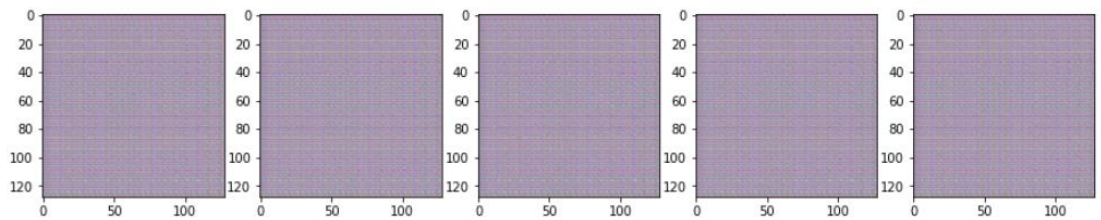
Epoch 1/960... Discriminator Loss: 21.2423... Generator Loss: 2.0872
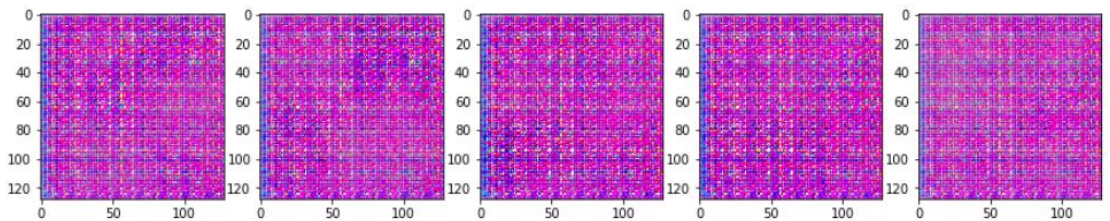
Epoch 5/960... Discriminator Loss: 8.3176... Generator Loss: 0.8010
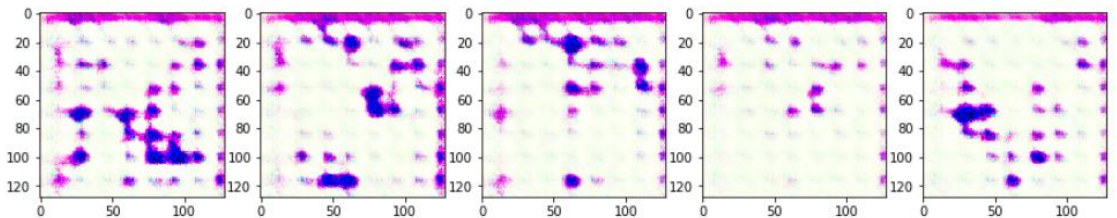
Epoch 9/960... Discriminator Loss: 4.5352... Generator Loss: 0.4294
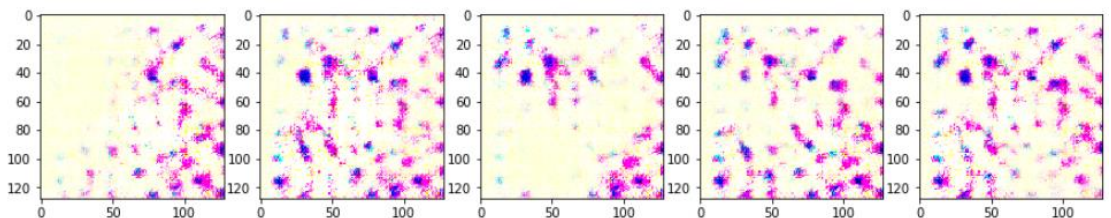
Epoch 20/960... Discriminator Loss: 3.6425... Generator Loss: 0.3593
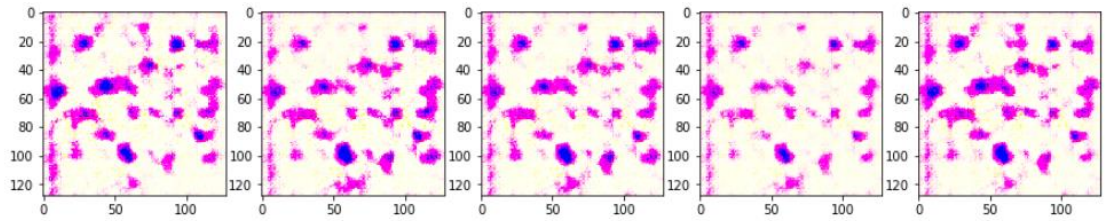
Epoch 32/960... Discriminator Loss: 0.4923... Generator Loss: 2.9169
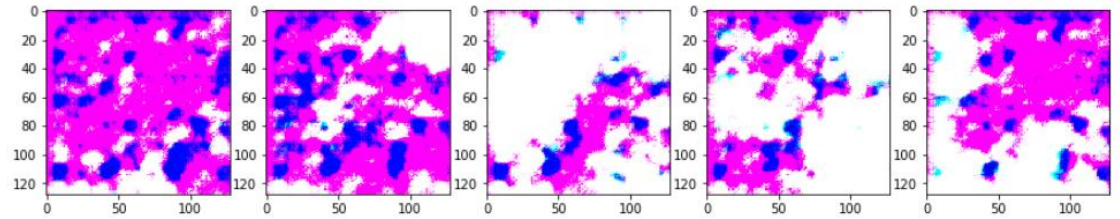
Epoch 59/960... Discriminator Loss: 0.6895... Generator Loss: 1.8604
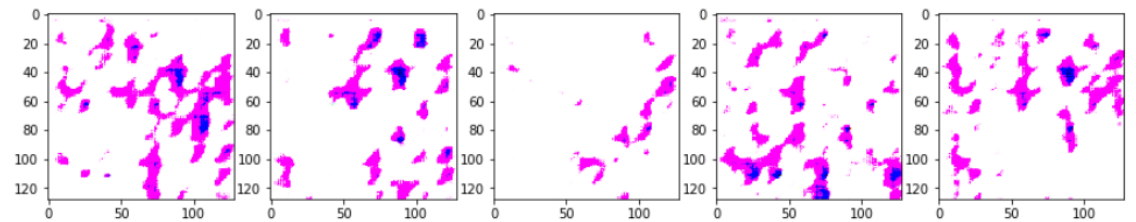
Epoch 102/960... Discriminator Loss: 0.4305... Generator Loss: 3.4436
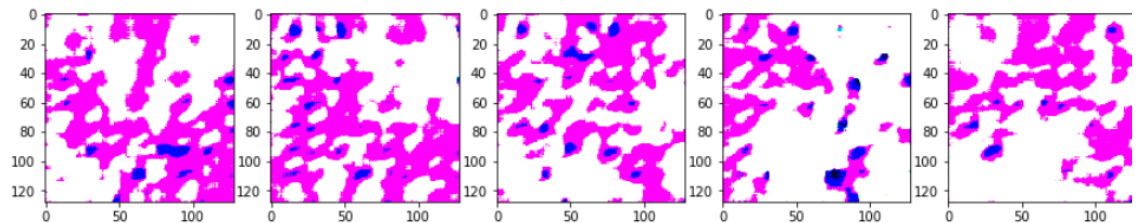
Epoch 236/960... Discriminator Loss: 0.3747... Generator Loss: 3.1416
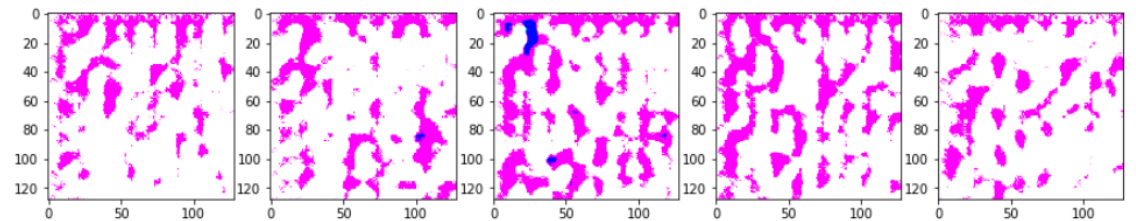
Epoch 280/960... Discriminator Loss: 0.3399... Generator Loss: 4.1943
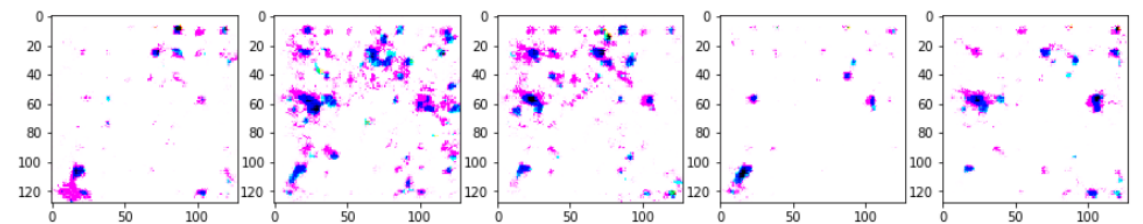
Epoch 496/960... Discriminator Loss: 0.3406... Generator Loss: 5.0883

31
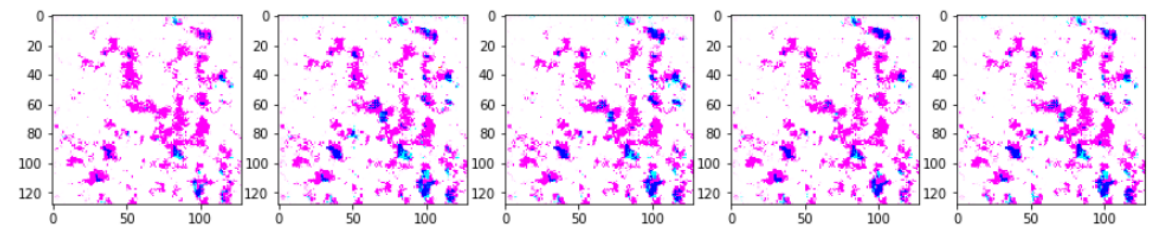
Epoch 600/960... Discriminator Loss: 0.3283... Generator Loss: 6.2467
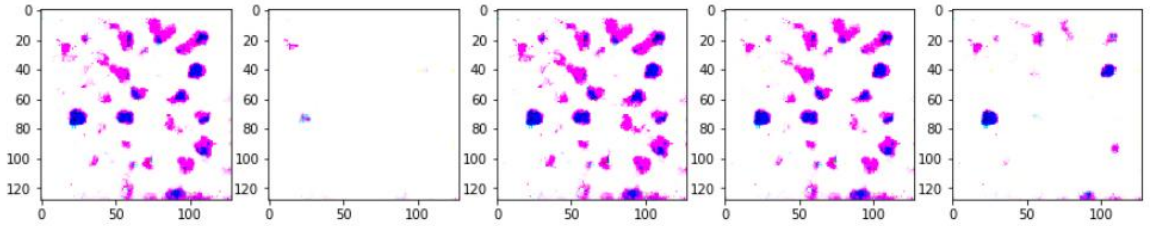
Epoch 791/960... Discriminator Loss: 0.3322... Generator Loss: 5.6364
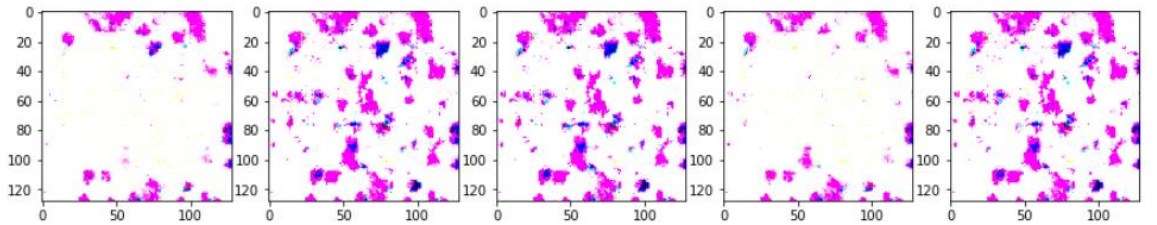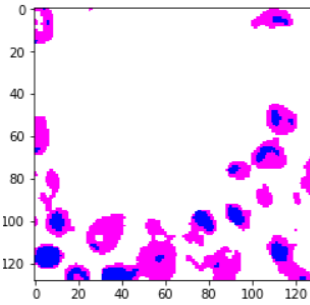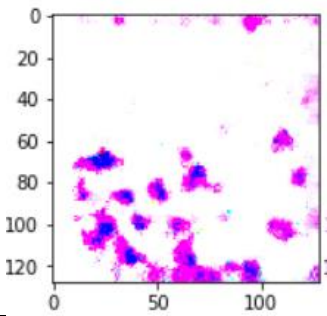
Epoch 826/960... Discriminator Loss: 0.3258... Generator Loss: 8.0561

Figure 14. Training Narrow Bone Dataset

5.3 Results and Comparison

After 960 epoch and more than 9 hours of training we are getting output images that resembles to the original. We need to consider that have trained only a total of 1328 images each one very different from the rest, that means that:

- Our training data is not big enough.
- Significant differences between all the images. There is not possible to recognise clear pattern in the dataset.

The Generated images are a total of 1328 samples after data augmentation. Size 128x128.	
Real Images	**Generated Images**
Real Image. Marrow Bone. Positive	Epoch 771/960. D Loss: 0.3279. G Loss: 6.4827.
Real Image. Marrow Bone. Negative	Epoch 790/960. D Loss: 0.3278. G Loss: 6.3440.

Figure 15. Narrow Bone Dataset: Real Vs Generate Comparison
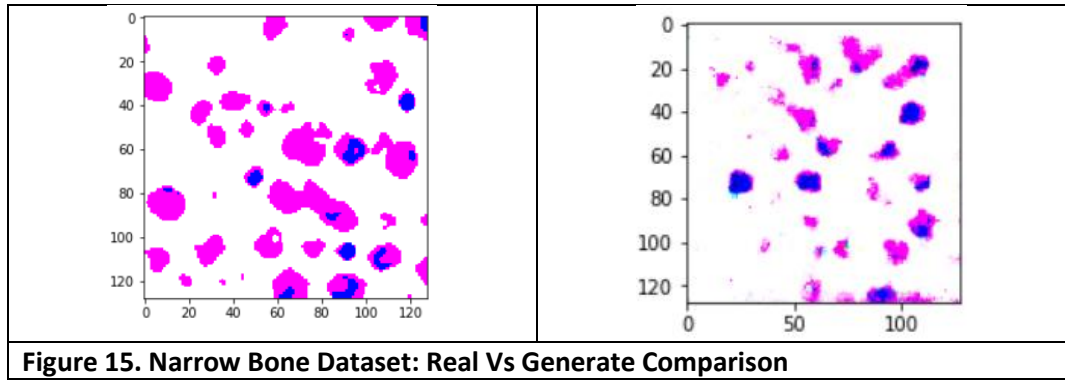
Potential problems:

- **Non-convergence**: the model parameters oscillate, destabilize and never converge.
- **Mode collapse**: the generator collapses which produces limited varieties of samples.
- **Diminished gradient**: the discriminator gets too successful that the generator gradient vanishes and learns nothing.
  - Unbalance between the generator and discriminator causing overfitting.
  - Highly sensitive to the hyperparameter selections.

Figure 16. Narrow Bone Dataset: First Real Data DCGAN Loss Chart
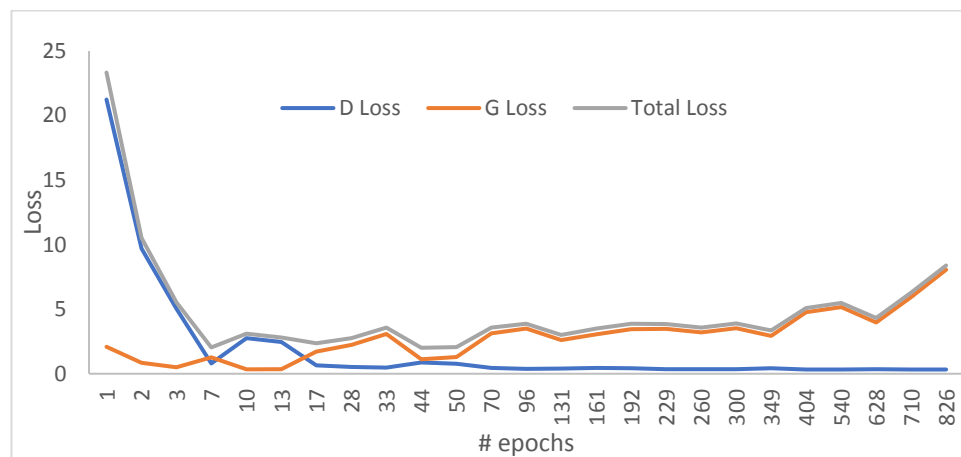
This problem is known as data imbalance and can cause our model to be more biased towards one class, usually the one which has more samples. Particularly in fields such as healthcare, classifying the minority class (benign in this case) as majority class (malignant in this case) can be very risky. We will deal with data imbalance by randomly undersampling the majority class, i.e removing samples of the majority class to make the number of samples of the majority and minority class equal.

6. Breast Histology Tissue Generation with DCGAN

This dataset has been extracted from the [15] Bioimaging Challenge 2015: "Breast Histology Dataset". It contains four classes: normal, benign, in situ carcinoma and invasive carcinoma. Link, Publication.

- **Data manipulation:**

Process in the code:

1. Defining start point a final for the patching. Using the iteration "i" for height and "j" for width.
2. Defining the image as PAL type applying the 'RGB' format.
3. Setting up the box variable that we will use later in the function crop for the patching.
4. As we need to consolidate our database we will resize all the images to 128x128 plx.
5. Filtering:
   - Applying a grey scale and contract increase we will discard all images fully black or white.
   - Applying a filter by variance of the matrix we will discard no meaningful images.

```python
Train_Data = []
y_data = []
y_data2 = []
real = []
data_dir = "../Data/Normal/"
files = glob(os.path.join(data_dir, '*.tif'))

        # Normal label 0
        for myFile in files:
            im = Image.open(myFile)
            for i in range(0,imgheight-height,height):
                for j in range (0, imgwidth-width, width):
                    a = im.convert('RGB')
                    j1 =  min(0, imgwidth - j+width)
                    i1 =  min(0, imgheight - j+height)
                    box = (j, i, j+width, i+height)
                    # Crop images 226x226 px
                    a = a.crop(box)
                    # Resize to 128x128 px
                    a = a.resize((128, 128), Image.ANTIALIAS)
                    # filter images by applying requirement. Variance of the matrix has been
        calculated manually
                    for k in range (0,270,90):
                        a = ndimage.interpolation.rotate(a, k)
                        a = Image.fromarray(a)
                        if np.var(a) > 1200:
                            a = np.array(a)
                            a = cv2.GaussianBlur(a,(7,7),1)
                            _, a = cv2.threshold(a,85,255,cv2.THRESH_BINARY)
                            Train_Data.append (a)
                            y_data.append (0)
                            y_data2.append (0)
                            real.append (1)
                            #Transpose images for data augmentation
                            b = cv2.transpose(a)
                            Train_Data.append (b)
                            y_data.append (0)
                            y_data2.append (0)
                            real.append (0)
                            # b1 FLIP_LEFT_RIGHT
                            b1 = cv2.flip(a, flipCode=1)
                            Train_Data.append (b1)
                            y_data.append (0)
                            y_data2.append (0)
```

```
                real.append (0)
                # b2 FLIP_TOP_BOTTOM
                b2 = cv2.flip(a, flipCode=0)
                Train_Data.append (b2)
                y_data.append (0)
                y_data2.append (0)
                real.append (0)
        print(len(y_data))
```

We will create in parallel label to categorise our samples:

Train_Data → where we store all the image matrix in format NumPy array float32
y_data = [] → 0: Normal, 1: Benign, 2: In Situ, 3: Invasive
y_data2 = [] → 0: Normal, 0: Benign, 1: In Situ, 1: Invasive
real = [] → 0: Real Image, 1: Product of Data Augmentation

	Real Images	Threshold Applied
Normal		
Benign		
In Situ		
Invasive		

Figure 17. Breast Histology Tissue: Real Vs Threshold
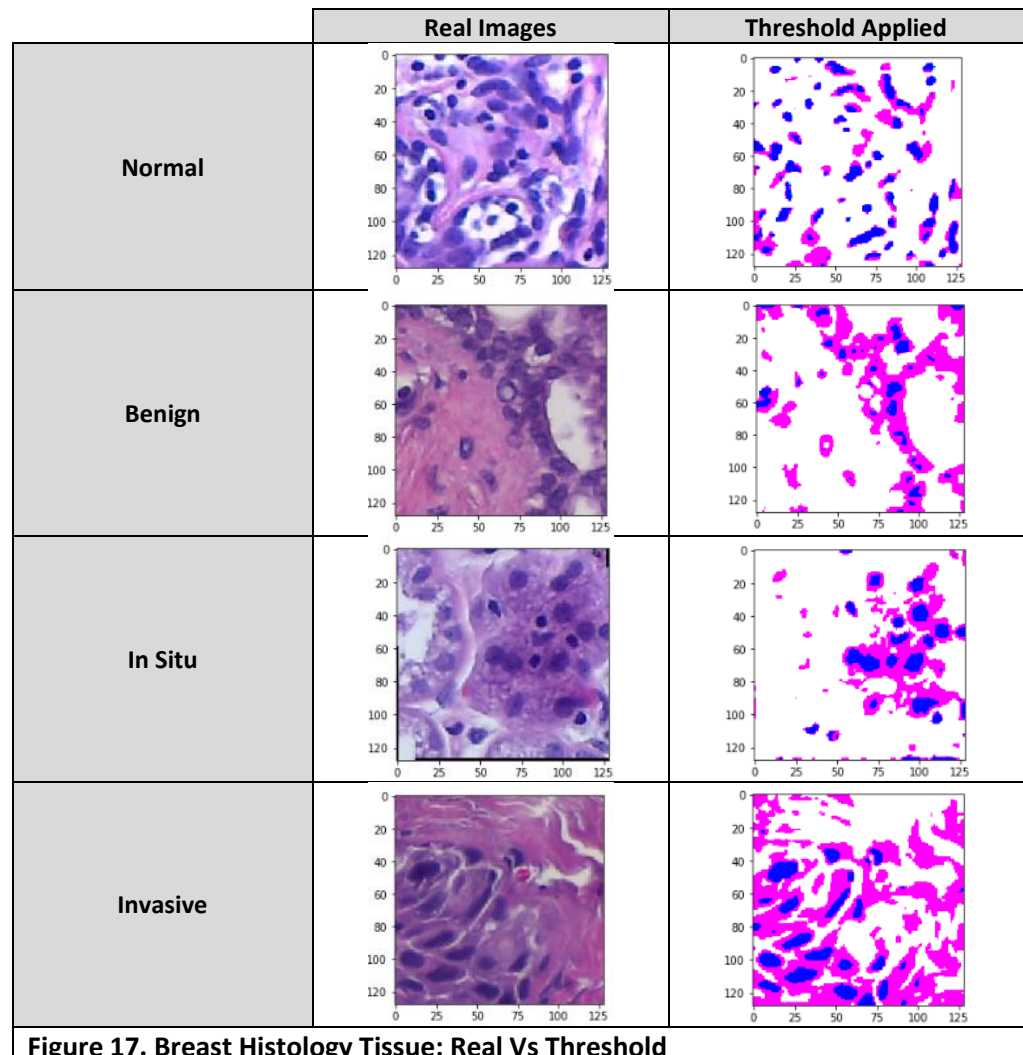
- **Results and Comparison**

Normal Tissue:

Details:

Bach Size: 64
Generator Learning Rate: 0.00001
Discriminator Learning Rate: 0.0005
Smooth Rate: 0.95
Total Epoch: 230
Image Size: 128x128 pixels
Iterations/Backpropagation: 10 Iteration

Total Time: 14 h
Time per epoch: 3.37 min
Total Train Samples: 25439

Epoch	Images generated during the training
15	Epoch 15/230... Discriminator Loss: 1.6276.. Generator Loss: 0.5681.. Total Loss: 2.1956 Time: 51.44496522718109 min . 8.574160871196849 h
95	Epoch 95/230... Discriminator Loss: 0.1080.. Generator Loss: 5.6189.. Total Loss: 5.7269 Time: 336.222700057396 min . 56.03711667623267 h
149	Epoch 149/230... Discriminator Loss: 0.0781.. Generator Loss: 6.4268.. Total Loss: 6.5049 Time: 529.263127842856 min . 88.21052130714266 h
224	Epoch 224/230... Discriminator Loss: 0.1019.. Generator Loss: 6.7688.. Total Loss: 6.8707 Time: 793.5099599860454 min . 132.2516599976742 h
Figure 18. Breast Histology Tissue: Normal	

 # Epoch Vs Discriminator Loss

 # Epoch Vs Generator Loss
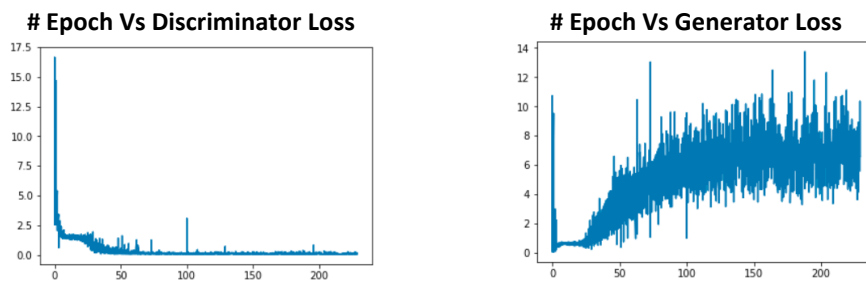
Figure 19. Breast Histology Tissue. Normal: Loss Charts

Benign Tissue:

Details:

Bach Size: 64
Generator Learning Rate: 0.00001

Total Time: 14.7 h
Time per epoch: 3.45 min

Discriminator Learning Rate: 0.0002   Total Train Samples: 26736

Smooth Rate: 0.95

Total Epoch: 230

Image Size: 128x128 pixels

Iterations/Backpropagation: 10 Iteration

Epoch	Images generated during the training
12	Epoch 12/230... Discriminator Loss: 1.4335.. Generator Loss: 0.6957.. Total Loss: 2.1291 Time: 43.116622718578824 min . 0.7186103786429804 h
95	Epoch 80/230... Discriminator Loss: 0.2292.. Generator Loss: 4.5321.. Total Loss: 4.7614 Time: 309.35855026181474 min . 5.1559758376969125 h
130	Epoch 130/230... Discriminator Loss: 0.3184.. Generator Loss: 3.7714.. Total Loss: 4.0898 Time: 500.7783298194098 min . 8.346305496990164 h
224	Epoch 221/230... Discriminator Loss: 0.2011.. Generator Loss: 6.5766.. Total Loss: 6.7777 Time: 851.6639828543334 min . 14.19439971423889 h
Figure 19. Breast Histology Tissue: Benign	

Epoch Vs Discriminator Loss   # Epoch Vs Generator Loss
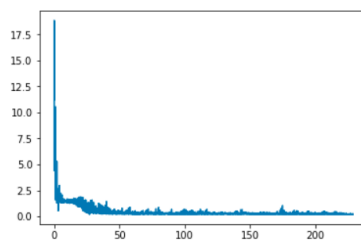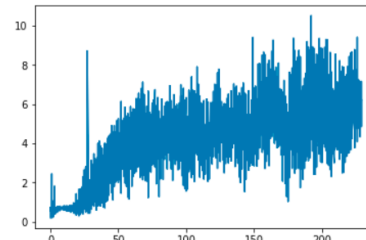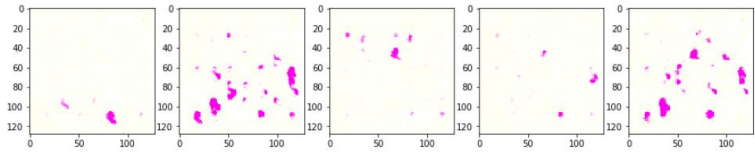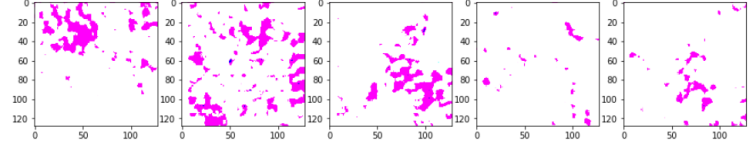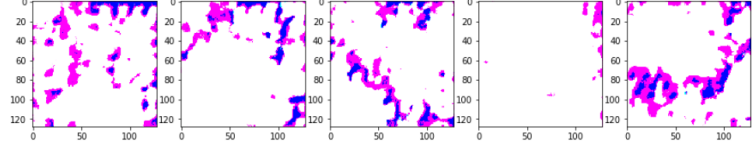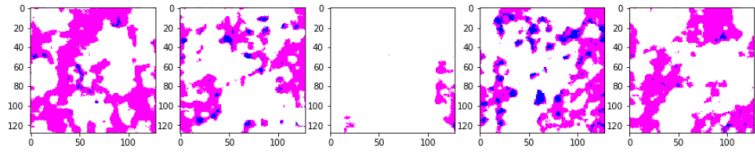
Figure 21. Breast Histology Tissue. Benign: Loss Charts

In Situ Carcinoma Tissue:

Details:

Bach Size: 64   Total Time: 14.7 h

Generator Learning Rate: 0.00001   Time per epoch: 3.45 min

Discriminator Learning Rate: 0.0002   Total Train Samples: 25511

Smooth Rate: 0.95

Total Epoch: 230

Image Size: 128x128 pixels

Iterations/Backpropagation: 10 Iteration

Epoch	Images generated during the training
14	 Epoch 14/230... Discriminator Loss: 1.5096.. Generator Loss: 0.6472.. Total Loss: 2.1568 Time: 64.27692181744303 min . 1.0712820302907171 h
78	 Epoch 78/230... Discriminator Loss: 0.2389.. Generator Loss: 4.1354.. Total Loss: 4.3743 Time: 371.30921748443507 min . 6.188486958073918 h
138	 Epoch 138/230... Discriminator Loss: 0.2397.. Generator Loss: 4.1177.. Total Loss: 4.3573 Time: 656.6703793425423 min . 10.944506322375705 h
206	 Epoch 206/230... Discriminator Loss: 0.2194.. Generator Loss: 5.7479.. Total Loss: 5.9674 Time: 984.5062022670793 min . 16.408436704451322 h

Figure 20. Breast Histology Tissue: In Situ Carcinoma

Epoch Vs Discriminator Loss

Epoch Vs Generator Loss
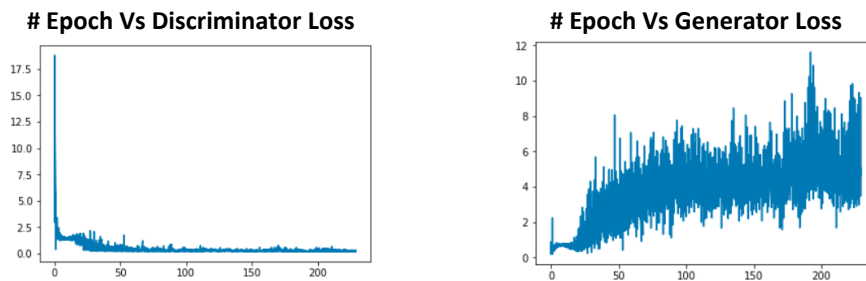
Figure 21. Breast Histology Tissue. In Situ Carcinoma: Loss Charts

Invasive Carcinoma Tissue:

Details:

Bach Size: 64
Generator Learning Rate: 0.00001
Discriminator Learning Rate: 0.0002
Smooth Rate: 0.95
Total Epoch: 230
Image Size: 128x128 pixels
Iterations/Backpropagation: 10 Iteration

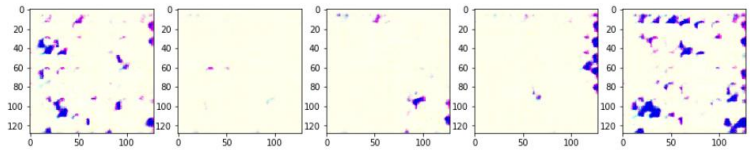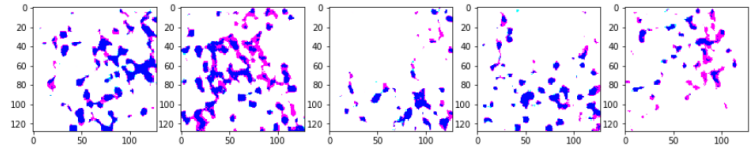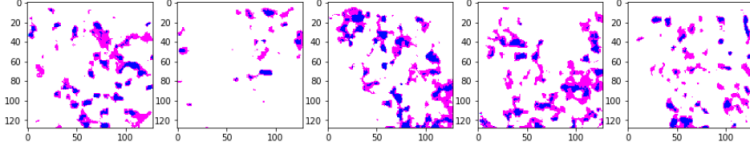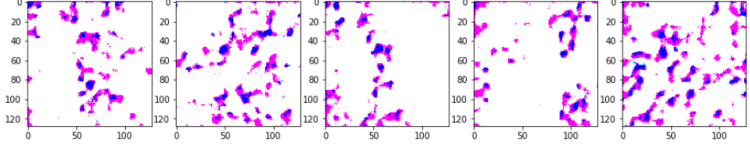Total Time: 16.5 h
Time per epoch: 3.49 min
Total Train Samples: 25600

Epoch	Images generated during the training

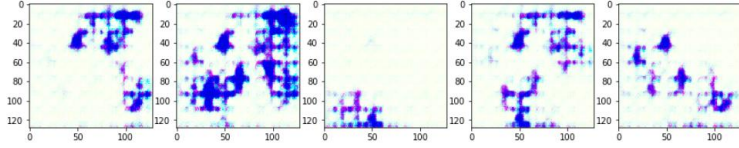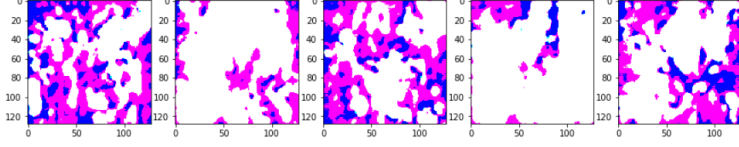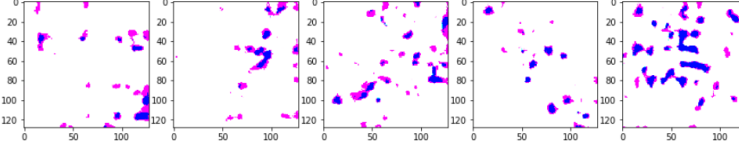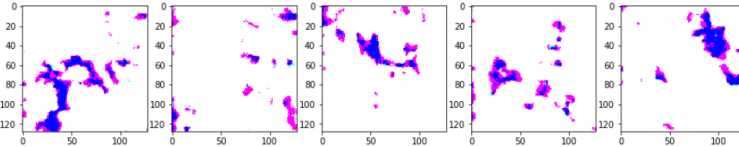|	
16	Epoch 16/190... Discriminator Loss: 1.4824.. Generator Loss: 0.6707.. Total Loss: 2.1530 Time: 21.752213619137184 min . 0.36253689365228636 h
78	Epoch 78/190... Discriminator Loss: 0.3114.. Generator Loss: 3.1069.. Total Loss: 3.4183 Time: 107.3797124448698 min . 1.7896618740811634 h
138	Epoch 155/190... Discriminator Loss: 0.5129.. Generator Loss: 2.5085.. Total Loss: 3.0214 Time: 213.96113195619546 min . 3.566018865936591 h
186	Epoch 186/190... Discriminator Loss: 0.2144.. Generator Loss: 4.5882.. Total Loss: 4.8025 Time: 256.65761864388674 min . 4.277626977398112 h

Figure 23. Breast Histology Tissue: Invasive Carcinoma

Epoch Vs Discriminator Loss   # Epoch Vs Generator Loss
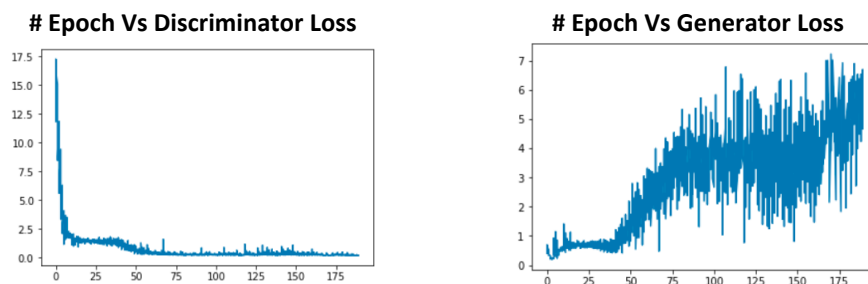
Figure 24. Breast Histology Tissue. Invasive Carcinoma: Loss Charts

Applying a CNN to the data:

Once we have finished with all the GANs for each type of tissue, we will train a simple CNN to check that generated data is reliable. For doing so, we can rely on Keras, as it is a high-level API, it will make the work much easier.

- Load Data and Packages

```
import pickle
import os
from matplotlib import pyplot as plt
%matplotlib inline
import numpy as np
from keras.utils import np_utils
import tensorflow as tf
from tensorflow import keras
from keras.preprocessing.image import ImageDataGenerator
```

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D

pickle_in = open("Train_Data.pickle", "rb")
Train_Data = pickle.load(pickle_in)
pickle_in = open("y_data.pickle", "rb")
y_data = pickle.load(pickle_in)
```

Labels: y_data = 0: Normal, 1: Benign, 2: In Situ, 3: Invasive

- Normalize Data From 0 to 255 to -1 to 1:

```
Train_Data = ((np.array(Train_Data, dtype=np.float32)/255) -0.5)*2
```

- Split in test and train variable:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(Train_Data, y_data, test_size=0.33,
shuffle=True)
```

- Convert Label to Categories (one hot):

```
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
```

- Model

```
model = Sequential()

model.add(Conv2D(32, (3, 3), input_shape=X_train.shape[1:]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), input_shape=X_train.shape[1:]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, (3, 3), input_shape=X_train.shape[1:]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(256, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(512, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(256))

model.add(Dense(4))
```

```
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy',
        optimizer='adam',
        metrics=['accuracy'])
```

- Results:

```
training = model.fit(X_train, y_train, batch_size=21, epochs=75, validation_split=0.2)
```

- Summary of the Process:

```
history = training.history
# Plot the training loss
plt.plot(history['loss'])
# Plot the validation loss
plt.plot(history['val_loss'])

# Show the figure
plt.show()
```

Output:
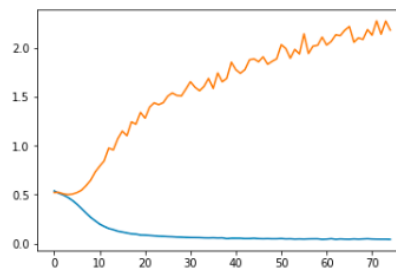
Figure 25: Breast Histology Tissue: CCN loss Chart

```
test_loss, test_acc = model.evaluate(X_test, y_test)

print('Test loss:', test_loss)
print('Test accuracy:', test_acc)
```

Output:

```
31155/31155 [==============================] - 10s 316us/step
Test loss: 0.4970536535212095
Test accuracy: 0.7718263521276342
```

```
model.summary()
```

Output:

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 126, 126, 32)      896
_____
activation_1 (Activation)    (None, 126, 126, 32)      0
_____
max_pooling2d_1 (MaxPooling2 (None, 63, 63, 32)        0
_____
conv2d_2 (Conv2D)            (None, 61, 61, 64)        18496
_____
activation_2 (Activation)    (None, 61, 61, 64)        0
_____
max_pooling2d_2 (MaxPooling2 (None, 30, 30, 64)        0
```

```
conv2d_3 (Conv2D)              (None, 28, 28, 128)     73856
_____
activation_3 (Activation)      (None, 28, 28, 128)     0
_____
max_pooling2d_3 (MaxPooling2   (None, 14, 14, 128)     0
_____
conv2d_4 (Conv2D)              (None, 12, 12, 256)     295168
_____
activation_4 (Activation)      (None, 12, 12, 256)     0
_____
max_pooling2d_4 (MaxPooling2   (None, 6, 6, 256)       0
_____
conv2d_5 (Conv2D)              (None, 4, 4, 512)       1180160
_____
activation_5 (Activation)      (None, 4, 4, 512)       0
_____
max_pooling2d_5 (MaxPooling2   (None, 2, 2, 512)       0
_____
flatten_1 (Flatten)            (None, 2048)            0
_____
dense_1 (Dense)                (None, 256)             524544
_____
dense_2 (Dense)                (None, 4)               1028
_____
activation_6 (Activation)      (None, 4)               0
=================================================================
Total params: 2,094,148
Trainable params: 2,094,148
Non-trainable params: 0
_____
```

- Confusion Matrix:

```
ynew = model.predict_classes(X_test)
Y_test = []
for i in range(len(y_test)):
    a = np.argmax(y_test[i])
    Y_test.append(a)
from sklearn.metrics import confusion_matrix
confusion_matrix(Y_test, ynew)
```

Output:

	Normal	Bening	In Situ	Invasive
Normal	2205	1151	1445	1460
Benign	800	2958	1529	1435
In Situ	900	1381	4378	1662
Invasive	870	1280	1660	6041

Table 6. Confusion Matrix: Test Data

We will apply the same model that we have created in the previous step to classify generated when we applied the DCGAN data, 10 images per class (Normal, Benign, In Situ and Invasive):

	Normal	Bening	In Situ	Invasive
Normal	5	2	1	2
Benign	2	5	2	1
In Situ	2	2	5	0
Invasive	1	1	1	7

Table 7. Confusion Matrix: Generated Data

As we can see the in tables aboce (**Table 6 and 7**) we may not be able to talk about a high accuracy bet we can see clearly a tendency to assign the righ label to the

generated sample. We can not expect to have better result because even the CNN is not accurate. Therefore, we can talk about tendency. We will be able to use the data classified in the right categories for data augmentation.

- Applying a t-SNE to Crosscheck Normal and Bening Images

To analyse the data more in depth, we applied the T-distributed stochastic neighbor embedding (t-SNE) to our data. Which is a machine learning technique for nonlinear dimensionality reduction, well-suited for embedding high-dimensional data for visualization in a low-dimensional space of two or three dimensions. This method is especially useful as it is nonlinear it will allow us to spot more differences between the samples and classification.
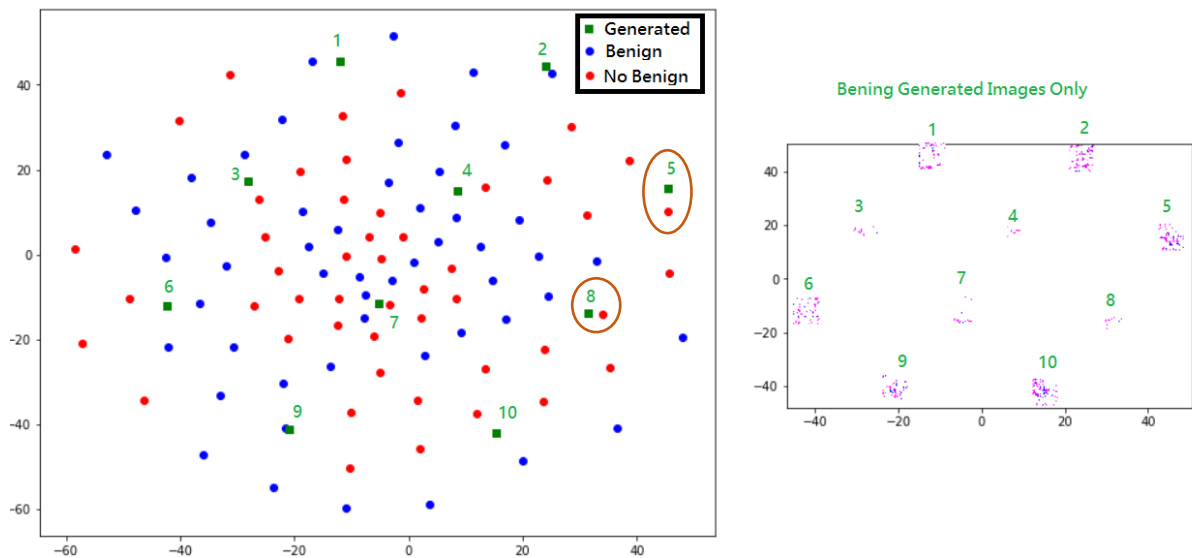
Figure 26: t-SNE for Generated Benign Images vs Real Comparison

In this case, we have reduced the number of classes to make tha chart not that busy. Hence why, we have grouped Normal/Benign Tissue as Benign and In Situ/Invasive as No Benign, the Generated data has been generated with the Benign dataset exclusively. Therefore, we expect to find Benign real samples clustered (marked as blue, **Figure 26**) with the synthetic generated images (marked as green, **Figure 26**). We have generated 10 synthetic images to compare them with 100 real images.

As expected we can see how the generated images are closer to the benign tissue real images but in two cases (point 5 and 6, **Figure 26**). Which is a good result considering the complexity of this method and the charasteritics of histology images as lack of reproducibility. In the right side of **figure 26** we can see the generated image on the graph.

7. Conclusion

In this first approach using GANs for histology images classification, we have implemented a fully automated method to generate synthetic data of high quality. It is well known that GANs are difficult to train as they are rather unstable. Hence why, in this study, we have applied a DCGAN model, as it contains some features like convolutional layers and batch normalisation, that will help with the stability of the convergence. As we work with two different networks, two antagonist CNNs, we train the generator and discriminator at the same time, we need to calculate losses for both networks. Therefore, we have applied two different learning rates for each network, this helps to balance the losses between these two networks.

Some problems as the background noise and colour scale have solved by applying threshold to the images. We have spotted two main problems with no easy solutions, as we must work with these two datasets and the amount of data was limited:

1. There are not clear patterns in the images. This is a difficulty of working with histology images specially when the area not properly localised. In Narrow Bone and Breast tissue histology images different areas can be found with distinct characteristics.
2. Not all the patches represent the label of the image. As we are working with patches and the label affect the whole image, lots of patches do not match with the label. For example, an image can be marked as *In Situ Carcinoma*, but it doesn't mean that this will be represent all the different patches of the dataset.

Because these issues we are not expecting to get a high accuracy in our classification method (CNN) instead we are expecting to find a clear tendency to categorise the samples in to the right label (see **Table 6 and 7**). Even if loss haven't converged very well, it doesn't necessarily mean that the model hasn't learned anything (see generated images **Figure 18**). The DCGAN applied in the previous section is showing how the images are getting better through the different epochs with better definitions and shapes.

If the model is running for too many epochs, the models suffer the following major problems:

- Non-convergence: the model parameters oscillate, destabilize and never converge.
- Diminished gradient: the discriminator gets too successful that the generator gradient vanishes and learns nothing.
- Unbalance between the generator and discriminator causing overfitting.

As a conclusion, we can see clear benefit using GANs for data augmentation as we are getting very clear images that resembles original images when the data provided to the model is large enough. But we need to make some considerations before applying this model:

- The data used for this model needs to provide a clear pattern to the model to reach a good accuracy and resolution.
- Enough data needs to be provided to the model to get satisfactory results. As we saw in the Narrow Bone and Breast Tissue comparison.
- Several tests need to be applied to the model to optimise parameters as: Generator Learning Rate, Discriminator Learning Rate, Batch Size, Epochs and Smoother.

8. Glossary

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
DCGAN	Deep Convolutional Generative Adversarial Network
DNN	Deep Neural Network
GAN	Generative Adversarial Network
GPU	Graphics Processing Unit
NN	Neural Network
np	NumPy
ReLU	Rectified Linear Unit
Tanh	Hyperbolic tangent
TF	TensorFlow
t-SNE	T-distributed stochastic neighbor embedding
TLU	Threshold Logic Unit
RNN	Recurrent Neural Network

9. References

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S.Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in advances in neural information processing system, 2014, pp 2672-2680

[2] H. R. Roth, L. Lu, J. Liu, J. Yao, A. Seff, K. Cherry, L. Kim, and R. M. Summers, "Improving computer-aided detection using convolutional neural networks and random view aggregation," IEEE Transactions on Medical Imaging, vol. 35, no. 5, pp. 1170–1181, May 2016.

[3] N. Tajbakhsh, J. Y. Shin, S. R. Gurudu, R. T. Hurst, C. B. Kendall, M. B. Gotway, and J. Liang, "Convolutional neural networks for medical image analysis: Full training or fine tuning?" IEEE Transactions on Medical Imaging, vol. 35, no. 5, pp. 1299–1312, May 2016.

[4] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classificationwith deep convolutional neural networks. Adv Neural Inf Process Syst, pp. 1097–105, 2012

[5] Khan, Muhammad, "Ink Mismatch Detection in Hyperspectral Document Images using Deep Learning", Oct 2017

[6] Charan, Saira and Khan, Muhammad and Khurshid, Khurram, "Breast Cancer Detection in Mammograms using Convolutional Neural Network", Mar 2018

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in Advances in neural information processing systems, 2012, pp. 1097–1105.

[8] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," arXiv preprint arXiv:1511.06434, 2015.

[9] E. L. Denton, S. Chintala, R. Fergus et al., "Deep generative image models using a laplacian pyramid of adversarial networks," in Advances in neural information processing systems, 2015, pp. 1486–1494.

[10] M. Mirza and S. Osindero, "Conditional generative adversarial nets," arXiv, pp. 1411.1784, 2014.

[11] N. Coudray, P. Santiago Ocampo, T. Sakellaropoulos, N. Narula, M. Snuderl, D. Fenyö, A. L. Moreira, N. Razavian & A. Tsirigos, "Classification and mutation prediction from non–small cell lung cancer histopathology images using deep learning", Nature Medicine vol. 24, pp 1559–1567, 2018.

[12] A. Kapil, A. Meier, A. Zuraw, K. Steele, M. Rebelatto, G. Schmidt, and N. Brieu "Deep Semi Supervised Generative Learning for Automated PD-L1 Tumor Cell Scoring on NSCLC Tissue Needle Biopsies", 2018.

[13] Frid-Adar, Maayan and Diamant, Idit and Klang, Eyal and Amitai, Marianne and Goldberger, Jacob and Greenspan, Heather, "GAN-based Synthetic Medical Image Augmentation for increased CNN Performance in Liver Lesion Classification", Neurocomputing, March 2018

[14] Bo Hu, Ye Tang, Eric I-Chao Chang, Yubo Fan, Maode Lai & Yan Xu, "Unsupervised Learning for Cell-level Visual Representation in Histopathology Images with Generative Adversarial Networks", Jul 2018

[15] raújo T, Aresta G, Castro E, Rouco J, Aguiar P, et al. "Classification of breast cancer histology images using Convolutional Neural Networks", PLOS ONE, Jun 2017

10. Figure List

11. Table List

12. Supplementary Material:

12.1   Applying GAN to a basic dataset: MNIST

- Importing the packages needed to run the code:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

- Loading the data using the special capability of TensorFlow to load the MNIST dataset.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("../Data/",one_hot=True)

plt.imshow(mnist.train.images[23].reshape(28,28),cmap='Greys')
```
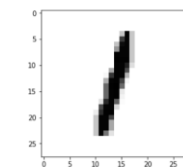

- Generator/Discriminator.

```
def generator(z,reuse=None):
    with tf.variable_scope('gen',reuse=reuse):
        hidden1 = tf.layers.dense(inputs=z,units=128)
        # Leaky Relu
        alpha = 0.01
        hidden1 = tf.maximum(alpha*hidden1,hidden1)
        hidden2 = tf.layers.dense(inputs=hidden1,units=128)

        hidden2 = tf.maximum(alpha*hidden2,hidden2)
        output = tf.layers.dense(hidden2,units=784,activation=tf.nn.tanh)
        return output

def discriminator(X,reuse=None):
    with tf.variable_scope('dis',reuse=reuse):
        hidden1 = tf.layers.dense(inputs=X,units=128)
        # Leaky Relu
        alpha = 0.01
        hidden1 = tf.maximum(alpha*hidden1,hidden1)

        hidden2 = tf.layers.dense(inputs=hidden1,units=128)
        hidden2 = tf.maximum(alpha*hidden2,hidden2)

        logits = tf.layers.dense(hidden2,units=1)
        output = tf.sigmoid(logits)
```

```
    return output, logits
```

- Placeholders

```
real_images = tf.placeholder(tf.float32,shape=[None,784])
z = tf.placeholder(tf.float32,shape=[None,100])
```

- Call Generator and Discriminator

```
G = generator(z)
D_output_real , D_logits_real = discriminator(real_images)
D_output_fake, D_logits_fake = discriminator(G,reuse=True)
```

- Losses

```
def loss_func(logits_in,labels_in):
    return
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=logits_in,labels=labels_in))

D_real_loss = loss_func(D_logits_real,tf.ones_like(D_logits_real)* (0.9))
D_fake_loss = loss_func(D_logits_fake,tf.zeros_like(D_logits_real))
D_loss = D_real_loss + D_fake_loss
G_loss = loss_func(D_logits_fake,tf.ones_like(D_logits_fake))
```

- Optimizers

```
learning_rate = 0.001

tvars = tf.trainable_variables()

d_vars = [var for var in tvars if 'dis' in var.name]
g_vars = [var for var in tvars if 'gen' in var.name]

print([v.name for v in d_vars])
print([v.name for v in g_vars])

D_trainer = tf.train.AdamOptimizer(learning_rate).minimize(D_loss, var_list=d_vars)
G_trainer = tf.train.AdamOptimizer(learning_rate).minimize(G_loss, var_list=g_vars)
```

- Training Session

```
batch_size = 100
epochs = 500
init = tf.global_variables_initializer()
samples = []
```

```
with tf.Session() as sess:

  sess.run(init)

  # Recall an epoch is an entire run through the training data
  for e in range(epochs):
    # // indicates classic division
    num_batches = mnist.train.num_examples // batch_size

    for i in range(num_batches):

      # Grab batch of images
      batch = mnist.train.next_batch(batch_size)

      # Get images, reshape and rescale to pass to D
      batch_images = batch[0].reshape((batch_size, 784))
      batch_images = batch_images*2 - 1

      # Z (random latent noise data for Generator)
      # -1 to 1 because of tanh activation
      batch_z = np.random.uniform(-1, 1, size=(batch_size, 100))

      # Run optimizers, no need to save outputs, we won't use them
      _ = sess.run(D_trainer, feed_dict={real_images: batch_images, z: batch_z})
      _ = sess.run(G_trainer, feed_dict={z: batch_z})


    print("Currently on Epoch {} of {} total...".format(e+1, epochs))

    # Sample from generator as we're training for viewing afterwards
    sample_z = np.random.uniform(-1, 1, size=(1, 100))
    gen_sample = sess.run(generator(z ,reuse=True),feed_dict={z: sample_z})

    samples.append(gen_sample)
```

- Checking result

```
plt.imshow(samples[100].reshape(28,28),cmap='Greys')
```
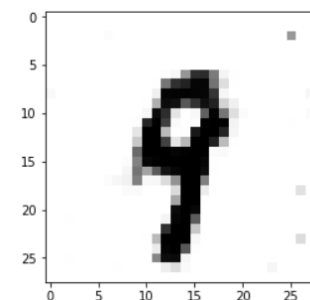
12.2   Applying DCGAN with Narrow Bone Dataset

- Loading Packages and Data

```python
import pickle
import os
from glob import glob
from matplotlib import pyplot as plt
%matplotlib inline
from PIL import Image
import numpy as np
import tensorflow as tf
import timeit

pickle_in = open("Train_Data.pickle", "rb")
Train_Data = pickle.load(pickle_in)
```

- Normalise data:

```python
Train_Data = ((np.array(Train_Data, dtype=np.float32)/255) - 0.5)*2
```

- Get Batches

```python
def get_batches(batch_size):
    shape = len(Train_Data), IMAGE_WIDTH, IMAGE_HEIGHT,
    data = Train_Data
    """
    Generate batches
    """
    current_index = 0
    while current_index + batch_size <= shape[0]:
        data_batch = data[current_index:current_index + batch_size]
        current_index += batch_size
        yield data_batch
```

- Placeholder:

```python
def model_inputs(image_width, image_height, image_channels, z_dim):
    """
    Create the model inputs
    """
    inputs_real = tf.placeholder(tf.float32, shape=(None, image_width, image_height,
image_channels), name='input_real')
    inputs_z = tf.placeholder(tf.float32, (None, z_dim), name='input_z')
    learning_rate_G = tf.placeholder(tf.float32, name='learning_rate')
    learning_rate_D = tf.placeholder(tf.float32, name='learning_rate')

    return inputs_real, inputs_z, learning_rate_G, learning_rate_D
```

- Discriminator:

```python
def discriminator(images, reuse=False):
    """
    Create the discriminator network
    """
    alpha = 0.2
    # Input layer 128*128*3 --> 64x64x64
    with tf.variable_scope('discriminator', reuse=reuse):
        # using 4 layer network as in DCGAN Paper

        # Conv 1
        conv1 = tf.layers.conv2d(images,filters = 64, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm1 = tf.layers.batch_normalization(conv1, training = True, epsilon = 1e-5)
        conv1_out = tf.nn.leaky_relu(batch_norm1, alpha=alpha)

        # Conv 2
        conv2 = tf.layers.conv2d(conv1_out,filters = 128, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm2 = tf.layers.batch_normalization(conv2, training = True, epsilon = 1e-5)
        conv2_out = tf.nn.leaky_relu(batch_norm2, alpha=alpha)

        # Conv 3
        conv3 = tf.layers.conv2d(conv2_out,filters = 256, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm3 = tf.layers.batch_normalization(conv3, training = True, epsilon = 1e-5)
        conv3_out = tf.nn.leaky_relu(batch_norm3, alpha=alpha)

        # Conv 4
        conv4 = tf.layers.conv2d(conv3_out,filters = 512, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm4 = tf.layers.batch_normalization(conv4, training = True, epsilon = 1e-5)
        conv4_out = tf.nn.leaky_relu(batch_norm4, alpha=alpha)

        # Conv 5
        conv5 = tf.layers.conv2d(conv4_out,filters = 1024, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm5 = tf.layers.batch_normalization(conv5, training = True, epsilon = 1e-5)
        conv5_out = tf.nn.leaky_relu(batch_norm5, alpha=alpha)

        # Flatten
        flatten = tf.reshape(conv5_out, (-1, 8*8*1024))

        # Logits
        logits = tf.layers.dense(flatten, 1)

        # Output
        out = tf.sigmoid(logits)

        return out, logits
```

- Generator

```python
def generator(z, out_channel_dim, is_train=True):
    """
    Create the generator network
    """
    alpha = 0.2

    with tf.variable_scope('generator', reuse=False if is_train==True else
True):
        # First fully connected laye First FC layer --> 8x8x1024
        fc1 = tf.layers.dense(z, 8*8*1024)
        # Reshape it
        fc1 = tf.reshape(fc1, (-1, 8, 8, 1024))
        # Leaky ReLU
        fc1 = tf.nn.leaky_relu(fc1, alpha=alpha)

        # Transposed conv 1 --> BatchNorm --> LeakyReLU
        # 8x8x1024 --> 16x16x512

        trans_conv1 = tf.layers.conv2d_transpose(inputs = fc1, filters = 512,
kernel_size = [5,5],
                                    strides = [2,2], padding = "SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv1 = tf.layers.batch_normalization(inputs =
trans_conv1,
                                    training=is_train,
                                    epsilon=1e-5)
        trans_conv1_out = tf.nn.leaky_relu(batch_trans_conv1, alpha=alpha)

        # Transposed conv 2 --> BatchNorm --> LeakyReLU
        # 16x16x512 --> 32x32x256

        trans_conv2 = tf.layers.conv2d_transpose(inputs = trans_conv1_out,
filters = 256, kernel_size = [5,5],
                                    strides = [2,2], padding = "SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv2 = tf.layers.batch_normalization(inputs =
trans_conv2, training=is_train,
                                    epsilon=1e-5)
        trans_conv2_out = tf.nn.leaky_relu(batch_trans_conv2, alpha=alpha)

        # Transposed conv 3 --> BatchNorm --> LeakyReLU
        # 32x32x256 --> 64x64x128

        trans_conv3 = tf.layers.conv2d_transpose(inputs = trans_conv2_out,
filters = 128, kernel_size = [5,5],
                                    strides = [2,2], padding = "SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
```

```python
        batch_trans_conv3 = tf.layers.batch_normalization(inputs =
trans_conv3, training=is_train,
                                        epsilon=1e-5)
        trans_conv3_out = tf.nn.leaky_relu(batch_trans_conv3, alpha=alpha)

        # Transposed conv 4 --> BatchNorm --> LeakyReLU
        # 64x64x128 --> 128x128x64

        trans_conv4 = tf.layers.conv2d_transpose(inputs = trans_conv3_out,
filters = 64, kernel_size = [5,5],
                                    strides = [2,2], padding = "SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv4 = tf.layers.batch_normalization(inputs =
trans_conv4, training=is_train,
                                        epsilon=1e-5)
        trans_conv4_out = tf.nn.leaky_relu(batch_trans_conv4, alpha=alpha)

        # Transposed conv 5 --> tanh
        # 128x128x64 --> 128x128x3
        # Output layer
        trans_conv5 = tf.layers.conv2d_transpose(inputs = trans_conv4_out,
filters = 3, kernel_size = [5,5],
                                    strides = [1,1], padding = "SAME",

kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))

        out = tf.tanh(trans_conv5, name="out")

    return out
```

- Loss Functions

```python
def model_loss(input_real, input_z, out_channel_dim):
        """
        Get the loss for the discriminator and generator
        """
        #label_smoothing = 0.9
        label_smoothing = 0.99

        g_model = generator(input_z, out_channel_dim)
        d_model_real, d_logits_real = discriminator(input_real)
        d_model_fake, d_logits_fake = discriminator(g_model, reuse=True)

        d_loss_real = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_real,
                                labels=tf.ones_like(d_model_real) * label_smoothing))
        d_loss_fake = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                                labels=tf.zeros_like(d_model_fake)))

        d_loss = d_loss_real + d_loss_fake
```

```
        g_loss = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                                labels=tf.ones_like(d_model_fake) *
        label_smoothing))

    return d_loss, g_loss
```

- Adam Optimizer

```
def model_opt(d_loss, g_loss, learning_rate_G, learning_rate_D, beta1):
        """
        Get optimization operations
        """
        t_vars = tf.trainable_variables()
        d_vars = [var for var in t_vars if var.name.startswith('discriminator')]
        g_vars = [var for var in t_vars if var.name.startswith('generator')]

        # Optimize
        with tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
            d_train_opt = tf.train.AdamOptimizer(learning_rate_D,
        beta1=beta1).minimize(d_loss, var_list=d_vars)
            g_train_opt = tf.train.AdamOptimizer(learning_rate_G,
        beta1=beta1).minimize(g_loss, var_list=g_vars)

    return d_train_opt, g_train_opt
```

- Function to show

```
def show_generator_output(sess, n_images, input_z, out_channel_dim):
        """
        Show example output for the generator
        """
        z_dim = input_z.get_shape().as_list()[-1]
        example_z = np.random.uniform(-1, 1, size=[n_images, z_dim])

        samples = sess.run(
            generator(input_z, out_channel_dim, False),
            feed_dict={input_z: example_z})

        fig=plt.figure(figsize=(15, 10))
        columns = 5
        rows = 1
        for i in range(1, columns*rows +1):
            img = samples[i]
            fig.add_subplot(rows, columns, i)
            plt.imshow(np.array(((((img)/2)+0.5), np.float32))
    plt.show()
```

- Training

Defining Parameters:

```
batch_size = 256
z_dim = 100
learning_rate_D = .00001
```

```
learning_rate_G = .0005
beta1 = 0.5
IMAGE_WIDTH = 128
IMAGE_HEIGHT = 128
shape = len(Train_Data), IMAGE_WIDTH, IMAGE_HEIGHT, 3
```

Starting the Training Session:

```
# One off as reuse is false
input_real, input_z, _, _ = model_inputs(shape[1], shape[2], shape[3], z_dim)
d_loss, g_loss = model_loss(input_real, input_z, shape[3])
d_opt, g_opt = model_opt(d_loss, g_loss, learning_rate_G, learning_rate_D,  beta1)
```

We will set the training to 230 epochs.

```
epochs = 230
epoch_i = 0
epoch_i_list=[]
steps_list=[]
train_loss_d_list=[]
train_loss_g_list=[]

start = timeit.default_timer()

steps = 0
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch_i in range(epochs):
        for batch_images in get_batches(batch_size):
            batch_images = batch_images * 2
            steps += 1
            #batch_z = np.array(np.random.uniform(-1, 1, size=(batch_size,z_dim)), np.float32)
            batch_z = np.random.uniform(-1, 1, size=(batch_size,z_dim))
            _ = sess.run(d_opt, feed_dict={input_real: batch_images, input_z:np.array(batch_z,
np.float32)}
             _ = sess.run(g_opt, feed_dict={input_real: batch_images, input_z:
np.array(batch_z, np.float32)})

            if steps % 30 == 0:
                    #print("There!")
                    # At the end of every 10 epochs, get the losses and print them out
                    train_loss_d = d_loss.eval({input_z: batch_z, input_real: batch_images})
                    train_loss_g = g_loss.eval({input_z: batch_z})

                    epoch_i_list.append(epoch_i)
                    steps_list.append(steps)
                    train_loss_d_list.append (train_loss_d)
                    train_loss_g_list.append (train_loss_g)

                if steps % 880==0:
                    stop = timeit.default_timer()
                    show_generator_output(sess, 6, input_z, shape[3])
                    print("Epoch {}/{}...".format(epoch_i+1, epochs),
                        "Discriminator Loss: {:.4f}..".format(train_loss_d),
                        "Generator Loss: {:.4f}..".format(train_loss_g),
                          "Total Loss: {:.4f}".format(train_loss_d + train_loss_g))
```

```
print('Time: ', (stop - start)/60, 'min', '. ', (stop - start)/3600, 'h')
```

12.3   Applying DCGAN with Breast Histology: Normal

- Loading Packages and Data

```
import pickle
import os
from glob import glob
from matplotlib import pyplot as plt
%matplotlib inline
from PIL import Image
import numpy as np
import tensorflow as tf
import timeit


pickle_in = open("Train_Data.pickle", "rb")
Train_Data = pickle.load(pickle_in)
```

- Normalise data from 0 to 255 → from -1 to 1

```
Train_Data = ((np.array(Train_Data, dtype=np.float32)/255) - 0.5)*2
```

- Get Batches

```
def get_batches(batch_size):
        shape = len(Train_Data), IMAGE_WIDTH, IMAGE_HEIGHT,
        data = Train_Data
        """
        Generate batches
        """
        current_index = 0
        while current_index + batch_size <= shape[0]:
            data_batch = data[current_index:current_index + batch_size]
            current_index += batch_size
    yield data_batch
```

- Placeholder

```
def model_inputs(image_width, image_height, image_channels, z_dim):
        """
        Create the model inputs
        """
        inputs_real = tf.placeholder(tf.float32, shape=(None, image_width,
        image_height, image_channels), name='input_real')
        inputs_z = tf.placeholder(tf.float32, (None, z_dim), name='input_z')
        learning_rate_G = tf.placeholder(tf.float32, name='learning_rate')
        learning_rate_D = tf.placeholder(tf.float32, name='learning_rate')
    return inputs_real, inputs_z, learning_rate_G, learning_rate_D
```

- Discriminator

```python
def discriminator(images, reuse=False):
    """
    Create the discriminator network
    """
    alpha = 0.2
    # Input layer 128*128*3 --> 64x64x64
    with tf.variable_scope('discriminator', reuse=reuse):
        # using 4 layer network as in DCGAN Paper

        # Conv 1
        conv1 = tf.layers.conv2d(images,filters = 64, kernel_size = [5,5],
                        strides = [2,2], padding="SAME",
                        kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm1 = tf.layers.batch_normalization(conv1, training = True, epsilon = 1e-5)
        conv1_out = tf.nn.leaky_relu(batch_norm1, alpha=alpha)

        # Conv 2
        conv2 = tf.layers.conv2d(conv1_out,filters = 128, kernel_size = [5,5],
                        strides = [2,2], padding="SAME",
                        kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm2 = tf.layers.batch_normalization(conv2, training = True, epsilon = 1e-5)
        conv2_out = tf.nn.leaky_relu(batch_norm2, alpha=alpha)

        # Conv 3
        conv3 = tf.layers.conv2d(conv2_out,filters = 256, kernel_size = [5,5],
                        strides = [2,2], padding="SAME",
                        kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm3 = tf.layers.batch_normalization(conv3, training = True, epsilon = 1e-5)
        conv3_out = tf.nn.leaky_relu(batch_norm3, alpha=alpha)

        # Conv 4
        conv4 = tf.layers.conv2d(conv3_out,filters = 512, kernel_size = [5,5],
                        strides = [2,2], padding="SAME",
                        kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm4 = tf.layers.batch_normalization(conv4, training = True, epsilon = 1e-5)
        conv4_out = tf.nn.leaky_relu(batch_norm4, alpha=alpha)

        # Conv 5
        conv5 = tf.layers.conv2d(conv4_out,filters = 1024, kernel_size = [5,5],
                        strides = [2,2], padding="SAME",
                        kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm5 = tf.layers.batch_normalization(conv5, training = True, epsilon = 1e-5)
        conv5_out = tf.nn.leaky_relu(batch_norm5, alpha=alpha)

        # Flatten
        flatten = tf.reshape(conv5_out, (-1, 8*8*1024))

        # Logits
        logits = tf.layers.dense(flatten, 1)

        # Output
        out = tf.sigmoid(logits)
```

```
        return out, logits
```

- Generator

```
def generator(z, out_channel_dim, is_train=True):
    """
    Create the generator network
    """
    alpha = 0.2

    with tf.variable_scope('generator', reuse=False if is_train==True else True):
        # First fully connected laye First FC layer --> 8x8x1024
        fc1 = tf.layers.dense(z, 8*8*1024)
        # Reshape it
        fc1 = tf.reshape(fc1, (-1, 8, 8, 1024))
        # Leaky ReLU
        fc1 = tf.nn.leaky_relu(fc1, alpha=alpha)


        # Transposed conv 1 --> BatchNorm --> LeakyReLU
        # 8x8x1024 --> 16x16x512

        trans_conv1 = tf.layers.conv2d_transpose(inputs = fc1, filters = 512, kernel_size = [5,5],
                            strides = [2,2], padding = "SAME",
                            kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv1 = tf.layers.batch_normalization(inputs = trans_conv1,
                                    training=is_train,
                                    epsilon=1e-5)
        trans_conv1_out = tf.nn.leaky_relu(batch_trans_conv1, alpha=alpha)

        # Transposed conv 2 --> BatchNorm --> LeakyReLU
        # 16x16x512 --> 32x32x256

        trans_conv2 = tf.layers.conv2d_transpose(inputs = trans_conv1_out, filters = 256,
kernel_size = [5,5],
                            strides = [2,2], padding = "SAME",
                            kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv2 = tf.layers.batch_normalization(inputs = trans_conv2,
training=is_train,
                                    epsilon=1e-5)
        trans_conv2_out = tf.nn.leaky_relu(batch_trans_conv2, alpha=alpha)


        # Transposed conv 3 --> BatchNorm --> LeakyReLU
        # 32x32x256 --> 64x64x128

        trans_conv3 = tf.layers.conv2d_transpose(inputs = trans_conv2_out, filters = 128,
kernel_size = [5,5],
                            strides = [2,2], padding = "SAME",
                            kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
```

```
      batch_trans_conv3 = tf.layers.batch_normalization(inputs = trans_conv3,
training=is_train,
                                      epsilon=1e-5)
    trans_conv3_out = tf.nn.leaky_relu(batch_trans_conv3, alpha=alpha)


    # Transposed conv 4 --> BatchNorm --> LeakyReLU
    # 64x64x128 --> 128x128x64

    trans_conv4 = tf.layers.conv2d_transpose(inputs = trans_conv3_out, filters = 64,
kernel_size = [5,5],
                              strides = [2,2], padding = "SAME",
                              kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
    batch_trans_conv4 = tf.layers.batch_normalization(inputs = trans_conv4,
training=is_train,
                                      epsilon=1e-5)
    trans_conv4_out = tf.nn.leaky_relu(batch_trans_conv4, alpha=alpha)


    # Transposed conv 5 --> tanh
    # 128x128x64 --> 128x128x3
    # Output layer
    trans_conv5 = tf.layers.conv2d_transpose(inputs = trans_conv4_out, filters = 3,
kernel_size = [5,5],
                              strides = [1,1], padding = "SAME",
                              kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))

    out = tf.tanh(trans_conv5, name="out")

    return out
```

- Loss Functions

```
def model_loss(input_real, input_z, out_channel_dim):
    """
    Get the loss for the discriminator and generator
    """

    #label_smoothing = 0.9
    label_smoothing = 0.99

    g_model = generator(input_z, out_channel_dim)
    d_model_real, d_logits_real = discriminator(input_real)
    d_model_fake, d_logits_fake = discriminator(g_model, reuse=True)

    d_loss_real = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_real,
                          labels=tf.ones_like(d_model_real) * label_smoothing))
    d_loss_fake = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                          labels=tf.zeros_like(d_model_fake)))

    d_loss = d_loss_real + d_loss_fake
```

```
g_loss = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                              labels=tf.ones_like(d_model_fake) * label_smoothing))

return d_loss, g_loss
```

- Adam Optimizer

```
def model_opt(d_loss, g_loss, learning_rate_G, learning_rate_D, beta1):
    """
    Get optimization operations
    """
    t_vars = tf.trainable_variables()
    d_vars = [var for var in t_vars if var.name.startswith('discriminator')]
    g_vars = [var for var in t_vars if var.name.startswith('generator')]

    # Optimize
    with tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
        d_train_opt = tf.train.AdamOptimizer(learning_rate_D, beta1=beta1).minimize(d_loss,
var_list=d_vars)
        g_train_opt = tf.train.AdamOptimizer(learning_rate_G, beta1=beta1).minimize(g_loss,
var_list=g_vars)

    return d_train_opt, g_train_opt
```

- Show function

```
def show_generator_output(sess, n_images, input_z, out_channel_dim):
    """
    Show example output for the generator
    """
    z_dim = input_z.get_shape().as_list()[-1]
    example_z = np.random.uniform(-1, 1, size=[n_images, z_dim])

    samples = sess.run(
        generator(input_z, out_channel_dim, False),
        feed_dict={input_z: example_z})

    fig=plt.figure(figsize=(15, 10))
    columns = 5
    rows = 1
    for i in range(1, columns*rows +1):
        img = samples[i]
        fig.add_subplot(rows, columns, i)
        plt.imshow(np.array((((img)/2)+0.5), np.float32))
    plt.show()
```

- Train parameters

```
batch_size = 64
z_dim = 100
```

```
learning_rate_D = .00001
learning_rate_G = .0005
beta1 = 0.5
IMAGE_WIDTH = 128
IMAGE_HEIGHT = 128
shape = len(Train_Data), IMAGE_WIDTH, IMAGE_HEIGHT, 3
```

- Defining

```
# One off as reuse is false
input_real, input_z, _, _ = model_inputs(shape[1], shape[2], shape[3], z_dim)
d_loss, g_loss = model_loss(input_real, input_z, shape[3])
d_opt, g_opt = model_opt(d_loss, g_loss, learning_rate_G, learning_rate_D,  beta1)
```

- Train Session

```
epochs = 230
epoch_i = 0
epoch_i_list=[]
steps_list=[]
train_loss_d_list=[]
train_loss_g_list=[]

start = timeit.default_timer()

steps = 0
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch_i in range(epochs):
        for batch_images in get_batches(batch_size):
            batch_images = batch_images * 2
            steps += 1
            batch_z = np.random.uniform(-1, 1, size=(batch_size,z_dim))
            _ = sess.run(d_opt, feed_dict={input_real: batch_images, input_z: np.array(batch_z,
np.float32)})
            _ = sess.run(g_opt, feed_dict={input_real: batch_images, input_z: np.array(batch_z,
np.float32)})

            if steps % 10 == 0:
                # At the end of every 10 epochs, get the losses and print them out
                train_loss_d = d_loss.eval({input_z: batch_z, input_real: batch_images})
                train_loss_g = g_loss.eval({input_z: batch_z})

                epoch_i_list.append(epoch_i)
                steps_list.append(steps)
                train_loss_d_list.append (train_loss_d)
                train_loss_g_list.append (train_loss_g)

            if steps % 850==0:
                stop = timeit.default_timer()
```

```
                    show_generator_output(sess, 6, input_z, shape[3])
                    print("Epoch {}/{}...".format(epoch_i+1, epochs),
                        "Discriminator Loss: {:.4f}..".format(train_loss_d),
                        "Generator Loss: {:.4f}..".format(train_loss_g),
                          "Total Loss: {:.4f}".format(train_loss_d + train_loss_g))
                    print('Time: ', (stop - start)/60, 'min', '. ', (stop - start)/3600, 'h')
```

12.4   Applying DCGAN with Breast Histology: Benign

- Loading Packages and Data

```
import pickle
import os
from glob import glob
from matplotlib import pyplot as plt
%matplotlib inline
from PIL import Image
import numpy as np
import tensorflow as tf
import timeit


pickle_in = open("Train_Data.pickle", "rb")
Train_Data = pickle.load(pickle_in)
```

- Normalise data from 0 to 255 → from -1 to 1

```
Train_Data = ((np.array(Train_Data, dtype=np.float32)/255) - 0.5)*2
```

- Get Batches

```
def get_batches(batch_size):
        shape = len(Train_Data), IMAGE_WIDTH, IMAGE_HEIGHT,
        data = Train_Data
        """
        Generate batches
        """
        current_index = 0
        while current_index + batch_size <= shape[0]:
           data_batch = data[current_index:current_index + batch_size]
           current_index += batch_size
     yield data_batch
```

- Placeholder

```
def model_inputs(image_width, image_height, image_channels, z_dim):
            """
            Create the model inputs
            """
        inputs_real = tf.placeholder(tf.float32, shape=(None, image_width,
        image_height, image_channels), name='input_real')
        inputs_z = tf.placeholder(tf.float32, (None, z_dim), name='input_z')
        learning_rate_G = tf.placeholder(tf.float32, name='learning_rate')
        learning_rate_D = tf.placeholder(tf.float32, name='learning_rate')
```

```
                  return inputs_real, inputs_z, learning_rate_G, learning_rate_D
```

- Discriminator

```python
def discriminator(images, reuse=False):
    """
    Create the discriminator network
    """
    alpha = 0.2
    # Input layer 128*128*3 --> 64x64x64
    with tf.variable_scope('discriminator', reuse=reuse):
        # using 4 layer network as in DCGAN Paper

        # Conv 1
        conv1 = tf.layers.conv2d(images,filters = 64, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm1 = tf.layers.batch_normalization(conv1, training = True, epsilon = 1e-5)
        conv1_out = tf.nn.leaky_relu(batch_norm1, alpha=alpha)

        # Conv 2
        conv2 = tf.layers.conv2d(conv1_out,filters = 128, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm2 = tf.layers.batch_normalization(conv2, training = True, epsilon = 1e-5)
        conv2_out = tf.nn.leaky_relu(batch_norm2, alpha=alpha)

        # Conv 3
        conv3 = tf.layers.conv2d(conv2_out,filters = 256, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm3 = tf.layers.batch_normalization(conv3, training = True, epsilon = 1e-5)
        conv3_out = tf.nn.leaky_relu(batch_norm3, alpha=alpha)

        # Conv 4
        conv4 = tf.layers.conv2d(conv3_out,filters = 512, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm4 = tf.layers.batch_normalization(conv4, training = True, epsilon = 1e-5)
        conv4_out = tf.nn.leaky_relu(batch_norm4, alpha=alpha)

        # Conv 5
        conv5 = tf.layers.conv2d(conv4_out,filters = 1024, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm5 = tf.layers.batch_normalization(conv5, training = True, epsilon = 1e-5)
        conv5_out = tf.nn.leaky_relu(batch_norm5, alpha=alpha)

        # Flatten
        flatten = tf.reshape(conv5_out, (-1, 8*8*1024))

        # Logits
```

```
        logits = tf.layers.dense(flatten, 1)

        # Output
        out = tf.sigmoid(logits)

        return out, logits
```

- Generator

```
def generator(z, out_channel_dim, is_train=True):
    """
    Create the generator network
    """
    alpha = 0.2

    with tf.variable_scope('generator', reuse=False if is_train==True else True):
        # First fully connected laye First FC layer --> 8x8x1024
        fc1 = tf.layers.dense(z, 8*8*1024)
        # Reshape it
        fc1 = tf.reshape(fc1, (-1, 8, 8, 1024))
        # Leaky ReLU
        fc1 = tf.nn.leaky_relu(fc1, alpha=alpha)

        # Transposed conv 1 --> BatchNorm --> LeakyReLU
        # 8x8x1024 --> 16x16x512

        trans_conv1 = tf.layers.conv2d_transpose(inputs = fc1, filters = 512, kernel_size = [5,5],
                            strides = [2,2], padding = "SAME",
                            kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv1 = tf.layers.batch_normalization(inputs = trans_conv1,
                                    training=is_train,
                                    epsilon=1e-5)
        trans_conv1_out = tf.nn.leaky_relu(batch_trans_conv1, alpha=alpha)

        # Transposed conv 2 --> BatchNorm --> LeakyReLU
        # 16x16x512 --> 32x32x256

        trans_conv2 = tf.layers.conv2d_transpose(inputs = trans_conv1_out, filters = 256,
kernel_size = [5,5],
                            strides = [2,2], padding = "SAME",
                            kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv2 = tf.layers.batch_normalization(inputs = trans_conv2,
training=is_train,
                                    epsilon=1e-5)
        trans_conv2_out = tf.nn.leaky_relu(batch_trans_conv2, alpha=alpha)


        # Transposed conv 3 --> BatchNorm --> LeakyReLU
        # 32x32x256 --> 64x64x128
```

```
        trans_conv3 = tf.layers.conv2d_transpose(inputs = trans_conv2_out, filters = 128,
kernel_size = [5,5],
                                strides = [2,2], padding = "SAME",
                                kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv3 = tf.layers.batch_normalization(inputs = trans_conv3,
training=is_train,
                                epsilon=1e-5)
        trans_conv3_out = tf.nn.leaky_relu(batch_trans_conv3, alpha=alpha)


        # Transposed conv 4 --> BatchNorm --> LeakyReLU
        # 64x64x128 --> 128x128x64


        trans_conv4 = tf.layers.conv2d_transpose(inputs = trans_conv3_out, filters = 64,
kernel_size = [5,5],
                                strides = [2,2], padding = "SAME",
                                kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv4 = tf.layers.batch_normalization(inputs = trans_conv4,
training=is_train,
                                epsilon=1e-5)
        trans_conv4_out = tf.nn.leaky_relu(batch_trans_conv4, alpha=alpha)


        # Transposed conv 5 --> tanh
        # 128x128x64 --> 128x128x3
        # Output layer
        trans_conv5 = tf.layers.conv2d_transpose(inputs = trans_conv4_out, filters = 3,
kernel_size = [5,5],
                                strides = [1,1], padding = "SAME",
                                kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))


        out = tf.tanh(trans_conv5, name="out")


        return out
```

- Loss Functions

```
def model_loss(input_real, input_z, out_channel_dim):
    """
    Get the loss for the discriminator and generator
    """

    label_smoothing = 0.99

    g_model = generator(input_z, out_channel_dim)
    d_model_real, d_logits_real = discriminator(input_real)
    d_model_fake, d_logits_fake = discriminator(g_model, reuse=True)

    d_loss_real = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_real,
                        labels=tf.ones_like(d_model_real) * label_smoothing))
    d_loss_fake = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                        labels=tf.zeros_like(d_model_fake)))
```

```
    d_loss = d_loss_real + d_loss_fake

    g_loss = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                                labels=tf.ones_like(d_model_fake) * label_smoothing))

    return d_loss, g_loss
```

- Adam Optimizer

```
def model_opt(d_loss, g_loss, learning_rate_G, learning_rate_D, beta1):
    """
    Get optimization operations
    """
    t_vars = tf.trainable_variables()
    d_vars = [var for var in t_vars if var.name.startswith('discriminator')]
    g_vars = [var for var in t_vars if var.name.startswith('generator')]

    # Optimize
    with tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
        d_train_opt = tf.train.AdamOptimizer(learning_rate_D, beta1=beta1).minimize(d_loss,
var_list=d_vars)
        g_train_opt = tf.train.AdamOptimizer(learning_rate_G, beta1=beta1).minimize(g_loss,
var_list=g_vars)

    return d_train_opt, g_train_opt
```

- Show function

```
def show_generator_output(sess, n_images, input_z, out_channel_dim):
    """
    Show example output for the generator
    """
    z_dim = input_z.get_shape().as_list()[-1]
    example_z = np.random.uniform(-1, 1, size=[n_images, z_dim])

    samples = sess.run(
        generator(input_z, out_channel_dim, False),
        feed_dict={input_z: example_z})
    #plt.imshow(helper.images_square_grid(samples))
    #plt.show()

    fig=plt.figure(figsize=(15, 10))
    columns = 5
    rows = 1
    for i in range(1, columns*rows +1):
        img = samples[i]
        fig.add_subplot(rows, columns, i)
        plt.imshow(np.array(((((img)/2)+0.5), np.float32))
    plt.show()
```

- Train parameters

```
batch_size = 64
z_dim = 100
learning_rate_D = .00001
learning_rate_G = .0005
beta1 = 0.5
IMAGE_WIDTH = 128
IMAGE_HEIGHT = 128
shape = len(Train_Data), IMAGE_WIDTH, IMAGE_HEIGHT, 3
```

- Defining

```
# One off as reuse is false
input_real, input_z, _, _ = model_inputs(shape[1], shape[2], shape[3], z_dim)
d_loss, g_loss = model_loss(input_real, input_z, shape[3])
d_opt, g_opt = model_opt(d_loss, g_loss, learning_rate_G, learning_rate_D,  beta1)
```

- Train Session

```
epochs = 230
epoch_i = 0
epoch_i_list=[]
steps_list=[]
train_loss_d_list=[]
train_loss_g_list=[]

start = timeit.default_timer()

steps = 0
with tf.Session() as sess:
   sess.run(tf.global_variables_initializer())
   for epoch_i in range(epochs):
     for batch_images in get_batches(batch_size):
        batch_images = batch_images * 2
        steps += 1
        #batch_z = np.array(np.random.uniform(-1, 1, size=(batch_size,z_dim)), np.float32)
        batch_z = np.random.uniform(-1, 1, size=(batch_size,z_dim))
        _ = sess.run(d_opt, feed_dict={input_real: batch_images, input_z: np.array(batch_z, np.float32)})
        _ = sess.run(g_opt, feed_dict={input_real: batch_images, input_z: np.array(batch_z, np.float32)})

        #print("here!")
        if steps % 10 == 0:
          #print("There!")
          # At the end of every 10 epochs, get the losses and print them out
          train_loss_d = d_loss.eval({input_z: batch_z, input_real: batch_images})
          train_loss_g = g_loss.eval({input_z: batch_z})
```

```
            epoch_i_list.append(epoch_i)
            steps_list.append(steps)
            train_loss_d_list.append (train_loss_d)
            train_loss_g_list.append (train_loss_g)

        if steps % 850==0:
            stop = timeit.default_timer()
            show_generator_output(sess, 6, input_z, shape[3])
            print("Epoch {}/{}...".format(epoch_i+1, epochs),
                "Discriminator Loss: {:.4f}..".format(train_loss_d),
                "Generator Loss: {:.4f}..".format(train_loss_g),
                 "Total Loss: {:.4f}".format(train_loss_d + train_loss_g))
            print('Time: ', (stop - start)/60, 'min', '. ', (stop - start)/3600, 'h')
```

12.5   Applying DCGAN with Breast Histology: In Situ

- Loading Packages and Data

```
import pickle
import os
from glob import glob
from matplotlib import pyplot as plt
%matplotlib inline
from PIL import Image
import numpy as np
import tensorflow as tf
import timeit


pickle_in = open("Train_Data.pickle", "rb")
Train_Data = pickle.load(pickle_in)
```

- Normalise data from 0 to 255 → from -1 to 1

```
Train_Data = ((np.array(Train_Data, dtype=np.float32)/255) - 0.5)*2
```

- Get Batches

```
def get_batches(batch_size):
        shape = len(Train_Data), IMAGE_WIDTH, IMAGE_HEIGHT,
        data = Train_Data
        """
        Generate batches
        """
        current_index = 0
        while current_index + batch_size <= shape[0]:
            data_batch = data[current_index:current_index + batch_size]
            current_index += batch_size
    yield data_batch
```

- Placeholder

```
def model_inputs(image_width, image_height, image_channels, z_dim):
        """
        Create the model inputs
```

```
            """
            inputs_real = tf.placeholder(tf.float32, shape=(None, image_width,
            image_height, image_channels), name='input_real')
            inputs_z = tf.placeholder(tf.float32, (None, z_dim), name='input_z')
            learning_rate_G = tf.placeholder(tf.float32, name='learning_rate')
            learning_rate_D = tf.placeholder(tf.float32, name='learning_rate')
        return inputs_real, inputs_z, learning_rate_G, learning_rate_D
```

- Discriminator

```
def discriminator(images, reuse=False):
    """
    Create the discriminator network
    """
    alpha = 0.2
    # Input layer 128*128*3 --> 64x64x64
    with tf.variable_scope('discriminator', reuse=reuse):


        # Conv 1
        conv1 = tf.layers.conv2d(images,filters = 64, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm1 = tf.layers.batch_normalization(conv1, training = True, epsilon = 1e-5)
        conv1_out = tf.nn.leaky_relu(batch_norm1, alpha=alpha)

        # Conv 2
        conv2 = tf.layers.conv2d(conv1_out,filters = 128, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm2 = tf.layers.batch_normalization(conv2, training = True, epsilon = 1e-5)
        conv2_out = tf.nn.leaky_relu(batch_norm2, alpha=alpha)

        # Conv 3
        conv3 = tf.layers.conv2d(conv2_out,filters = 256, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm3 = tf.layers.batch_normalization(conv3, training = True, epsilon = 1e-5)
        conv3_out = tf.nn.leaky_relu(batch_norm3, alpha=alpha)

        # Conv 4
        conv4 = tf.layers.conv2d(conv3_out,filters = 512, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm4 = tf.layers.batch_normalization(conv4, training = True, epsilon = 1e-5)
        conv4_out = tf.nn.leaky_relu(batch_norm4, alpha=alpha)

        # Conv 5
        conv5 = tf.layers.conv2d(conv4_out,filters = 1024, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm5 = tf.layers.batch_normalization(conv5, training = True, epsilon = 1e-5)
        conv5_out = tf.nn.leaky_relu(batch_norm5, alpha=alpha)
```

```python
    # Flatten
    flatten = tf.reshape(conv5_out, (-1, 8*8*1024))

    # Logits
    logits = tf.layers.dense(flatten, 1)

    # Output
    out = tf.sigmoid(logits)

    return out, logits
```

- Generator

```python
def generator(z, out_channel_dim, is_train=True):
    """
    Create the generator network
    """
    alpha = 0.2

    with tf.variable_scope('generator', reuse=False if is_train==True else True):
        # First fully connected laye First FC layer --> 8x8x1024
        fc1 = tf.layers.dense(z, 8*8*1024)
        # Reshape it
        fc1 = tf.reshape(fc1, (-1, 8, 8, 1024))
        # Leaky ReLU
        fc1 = tf.nn.leaky_relu(fc1, alpha=alpha)

        # Transposed conv 1 --> BatchNorm --> LeakyReLU
        # 8x8x1024 --> 16x16x512

        trans_conv1 = tf.layers.conv2d_transpose(inputs = fc1, filters = 512, kernel_size = [5,5],
                            strides = [2,2], padding = "SAME",
                            kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv1 = tf.layers.batch_normalization(inputs = trans_conv1,
                                training=is_train,
                                epsilon=1e-5)
        trans_conv1_out = tf.nn.leaky_relu(batch_trans_conv1, alpha=alpha)

        # Transposed conv 2 --> BatchNorm --> LeakyReLU
        # 16x16x512 --> 32x32x256

        trans_conv2 = tf.layers.conv2d_transpose(inputs = trans_conv1_out, filters = 256,
kernel_size = [5,5],
                                strides = [2,2], padding = "SAME",
                                kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv2 = tf.layers.batch_normalization(inputs = trans_conv2,
training=is_train,
                                    epsilon=1e-5)
        trans_conv2_out = tf.nn.leaky_relu(batch_trans_conv2, alpha=alpha)
```

```python
    # Transposed conv 3 --> BatchNorm --> LeakyReLU
    # 32x32x256 --> 64x64x128

    trans_conv3 = tf.layers.conv2d_transpose(inputs = trans_conv2_out, filters = 128, kernel_size = [5,5],
                                strides = [2,2], padding = "SAME",
                                kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
    batch_trans_conv3 = tf.layers.batch_normalization(inputs = trans_conv3, training=is_train,
                                epsilon=1e-5)
    trans_conv3_out = tf.nn.leaky_relu(batch_trans_conv3, alpha=alpha)

    # Transposed conv 4 --> BatchNorm --> LeakyReLU
    # 64x64x128 --> 128x128x64

    trans_conv4 = tf.layers.conv2d_transpose(inputs = trans_conv3_out, filters = 64, kernel_size = [5,5],
                                strides = [2,2], padding = "SAME",
                                kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
    batch_trans_conv4 = tf.layers.batch_normalization(inputs = trans_conv4, training=is_train,
                                epsilon=1e-5)
    trans_conv4_out = tf.nn.leaky_relu(batch_trans_conv4, alpha=alpha)

    # Transposed conv 5 --> tanh
    # 128x128x64 --> 128x128x3
    # Output layer
    trans_conv5 = tf.layers.conv2d_transpose(inputs = trans_conv4_out, filters = 3, kernel_size = [5,5],
                                strides = [1,1], padding = "SAME",
                                kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))

    out = tf.tanh(trans_conv5, name="out")

    return out
```

- Loss Functions

```python
def model_loss(input_real, input_z, out_channel_dim):
    """
    Get the loss for the discriminator and generator
    """

    label_smoothing = 0.99

    g_model = generator(input_z, out_channel_dim)
    d_model_real, d_logits_real = discriminator(input_real)
    d_model_fake, d_logits_fake = discriminator(g_model, reuse=True)

    d_loss_real = tf.reduce_mean(
```

```
        tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_real,
                                labels=tf.ones_like(d_model_real) * label_smoothing))
    d_loss_fake = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                                labels=tf.zeros_like(d_model_fake)))

    d_loss = d_loss_real + d_loss_fake

    g_loss = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                                labels=tf.ones_like(d_model_fake) * label_smoothing))

    return d_loss, g_loss
```

- Adam Optimizer

```
def model_opt(d_loss, g_loss, learning_rate_G, learning_rate_D, beta1):
    """
    Get optimization operations
    """
    t_vars = tf.trainable_variables()
    d_vars = [var for var in t_vars if var.name.startswith('discriminator')]
    g_vars = [var for var in t_vars if var.name.startswith('generator')]

    # Optimize
    with tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
        d_train_opt = tf.train.AdamOptimizer(learning_rate_D, beta1=beta1).minimize(d_loss,
var_list=d_vars)
        g_train_opt = tf.train.AdamOptimizer(learning_rate_G, beta1=beta1).minimize(g_loss,
var_list=g_vars)

    return d_train_opt, g_train_opt
```

- Show function

```
def show_generator_output(sess, n_images, input_z, out_channel_dim):
    """
    Show example output for the generator
    """
    z_dim = input_z.get_shape().as_list()[-1]
    example_z = np.random.uniform(-1, 1, size=[n_images, z_dim])

    samples = sess.run(
        generator(input_z, out_channel_dim, False),
        feed_dict={input_z: example_z})

    fig=plt.figure(figsize=(15, 10))
    columns = 5
    rows = 1
    for i in range(1, columns*rows +1):
        img = samples[i]
```

```
        fig.add_subplot(rows, columns, i)
        plt.imshow(np.array(((((img)/2)+0.5), np.float32))
    plt.show()
```

- Train parameters

```
batch_size = 64
z_dim = 100
learning_rate_D = .00001
learning_rate_G = .0005
beta1 = 0.5
IMAGE_WIDTH = 128
IMAGE_HEIGHT = 128
shape = len(Train_Data), IMAGE_WIDTH, IMAGE_HEIGHT, 3
```

- Defining placeholder parameters

```
# One off as reuse is false
input_real, input_z, _, _ = model_inputs(shape[1], shape[2], shape[3], z_dim)
d_loss, g_loss = model_loss(input_real, input_z, shape[3])
d_opt, g_opt = model_opt(d_loss, g_loss, learning_rate_G, learning_rate_D,  beta1)
```

- Train Session

```
epochs = 230
epoch_i = 0
epoch_i_list=[]
steps_list=[]
train_loss_d_list=[]
train_loss_g_list=[]

start = timeit.default_timer()

steps = 0
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch_i in range(epochs):
        for batch_images in get_batches(batch_size):
            batch_images = batch_images * 2
            steps += 1

            batch_z = np.random.uniform(-1, 1, size=(batch_size,z_dim))
            _ = sess.run(d_opt, feed_dict={input_real: batch_images, input_z: np.array(batch_z,
np.float32)})
            _ = sess.run(g_opt, feed_dict={input_real: batch_images, input_z: np.array(batch_z,
np.float32)})

            if steps % 10 == 0:
                #print("There!")
                # At the end of every 10 epochs, get the losses and print them out
```

```
        train_loss_d = d_loss.eval({input_z: batch_z, input_real: batch_images})
        train_loss_g = g_loss.eval({input_z: batch_z})

        epoch_i_list.append(epoch_i)
        steps_list.append(steps)
        train_loss_d_list.append (train_loss_d)
        train_loss_g_list.append (train_loss_g)

    if steps % 850==0:
        stop = timeit.default_timer()
        show_generator_output(sess, 6, input_z, shape[3])
        print("Epoch {}/{}...".format(epoch_i+1, epochs),
            "Discriminator Loss: {:.4f}..".format(train_loss_d),
            "Generator Loss: {:.4f}..".format(train_loss_g),
             "Total Loss: {:.4f}".format(train_loss_d + train_loss_g))
        print('Time: ', (stop - start)/60, 'min', '. ', (stop - start)/3600, 'h')
```

12.6   Applying DCGAN with Breast Histology: Invasive

- Loading Packages and Data

```
import pickle
import os
from glob import glob
from matplotlib import pyplot as plt
%matplotlib inline
from PIL import Image
import numpy as np
import tensorflow as tf
import timeit

pickle_in = open("Train_Data.pickle", "rb")
Train_Data = pickle.load(pickle_in)
```

- Normalise data from 0 to 255 → from -1 to 1

```
Train_Data = ((np.array(Train_Data, dtype=np.float32)/255) - 0.5)*2
```

- Get Batches

```
def get_batches(batch_size):
        shape = len(Train_Data), IMAGE_WIDTH, IMAGE_HEIGHT,
        data = Train_Data
        """
        Generate batches
        """
        current_index = 0
        while current_index + batch_size <= shape[0]:
            data_batch = data[current_index:current_index + batch_size]
```

```
            current_index += batch_size
    yield data_batch
```

- Create the model inputs

```python
def model_inputs(image_width, image_height, image_channels, z_dim):
    """
    Create the model inputs
    """
    inputs_real = tf.placeholder(tf.float32, shape=(None, image_width,
        image_height, image_channels), name='input_real')
    inputs_z = tf.placeholder(tf.float32, (None, z_dim), name='input_z')
    learning_rate_G = tf.placeholder(tf.float32, name='learning_rate')
    learning_rate_D = tf.placeholder(tf.float32, name='learning_rate')
    return inputs_real, inputs_z, learning_rate_G, learning_rate_D
```

- Discriminator

```python
def discriminator(images, reuse=False):
    """
    Create the discriminator network
    """
    alpha = 0.2
    # Input layer 128*128*3 --> 64x64x64
    with tf.variable_scope('discriminator', reuse=reuse):
        # using 4 layer network as in DCGAN Paper

        # Conv 1
        conv1 = tf.layers.conv2d(images,filters = 64, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm1 = tf.layers.batch_normalization(conv1, training = True, epsilon = 1e-5)
        conv1_out = tf.nn.leaky_relu(batch_norm1, alpha=alpha)

        # Conv 2
        conv2 = tf.layers.conv2d(conv1_out,filters = 128, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm2 = tf.layers.batch_normalization(conv2, training = True, epsilon = 1e-5)
        conv2_out = tf.nn.leaky_relu(batch_norm2, alpha=alpha)

        # Conv 3
        conv3 = tf.layers.conv2d(conv2_out,filters = 256, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm3 = tf.layers.batch_normalization(conv3, training = True, epsilon = 1e-5)
        conv3_out = tf.nn.leaky_relu(batch_norm3, alpha=alpha)

        # Conv 4
        conv4 = tf.layers.conv2d(conv3_out,filters = 512, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm4 = tf.layers.batch_normalization(conv4, training = True, epsilon = 1e-5)
```

```
        conv4_out = tf.nn.leaky_relu(batch_norm4, alpha=alpha)

        # Conv 5
        conv5 = tf.layers.conv2d(conv4_out,filters = 1024, kernel_size = [5,5],
                    strides = [2,2], padding="SAME",
                    kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_norm5 = tf.layers.batch_normalization(conv5, training = True, epsilon = 1e-5)
        conv5_out = tf.nn.leaky_relu(batch_norm5, alpha=alpha)

        # Flatten
        flatten = tf.reshape(conv5_out, (-1, 8*8*1024))

        # Logits
        logits = tf.layers.dense(flatten, 1)

        # Output
        out = tf.sigmoid(logits)

        return out, logits
```

- Generator

```
def generator(z, out_channel_dim, is_train=True):
    """
    Create the generator network
    """
    alpha = 0.2

    with tf.variable_scope('generator', reuse=False if is_train==True else True):
        # First fully connected laye First FC layer --> 8x8x1024
        fc1 = tf.layers.dense(z, 8*8*1024)
        # Reshape it
        fc1 = tf.reshape(fc1, (-1, 8, 8, 1024))
        # Leaky ReLU
        fc1 = tf.nn.leaky_relu(fc1, alpha=alpha)

        # Transposed conv 1 --> BatchNorm --> LeakyReLU
        # 8x8x1024 --> 16x16x512

        trans_conv1 = tf.layers.conv2d_transpose(inputs = fc1, filters = 512, kernel_size = [5,5],
                            strides = [2,2], padding = "SAME",
                            kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
        batch_trans_conv1 = tf.layers.batch_normalization(inputs = trans_conv1,
                                training=is_train,
                                epsilon=1e-5)
        trans_conv1_out = tf.nn.leaky_relu(batch_trans_conv1, alpha=alpha)

        # Transposed conv 2 --> BatchNorm --> LeakyReLU
        # 16x16x512 --> 32x32x256
```

```python
    trans_conv2 = tf.layers.conv2d_transpose(inputs = trans_conv1_out, filters = 256, kernel_size = [5,5],
                              strides = [2,2], padding = "SAME",
                              kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
    batch_trans_conv2 = tf.layers.batch_normalization(inputs = trans_conv2, training=is_train,
                              epsilon=1e-5)
    trans_conv2_out = tf.nn.leaky_relu(batch_trans_conv2, alpha=alpha)


    # Transposed conv 3 --> BatchNorm --> LeakyReLU
    # 32x32x256 --> 64x64x128

    trans_conv3 = tf.layers.conv2d_transpose(inputs = trans_conv2_out, filters = 128, kernel_size = [5,5],
                              strides = [2,2], padding = "SAME",
                              kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
    batch_trans_conv3 = tf.layers.batch_normalization(inputs = trans_conv3, training=is_train,
                              epsilon=1e-5)
    trans_conv3_out = tf.nn.leaky_relu(batch_trans_conv3, alpha=alpha)

    # Transposed conv 4 --> BatchNorm --> LeakyReLU
    # 64x64x128 --> 128x128x64

    trans_conv4 = tf.layers.conv2d_transpose(inputs = trans_conv3_out, filters = 64, kernel_size = [5,5],
                              strides = [2,2], padding = "SAME",
                              kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))
    batch_trans_conv4 = tf.layers.batch_normalization(inputs = trans_conv4, training=is_train,
                              epsilon=1e-5)
    trans_conv4_out = tf.nn.leaky_relu(batch_trans_conv4, alpha=alpha)

    # Transposed conv 5 --> tanh
    # 128x128x64 --> 128x128x3
    # Output layer
    trans_conv5 = tf.layers.conv2d_transpose(inputs = trans_conv4_out, filters = 3, kernel_size = [5,5],
                              strides = [1,1], padding = "SAME",
                              kernel_initializer=tf.truncated_normal_initializer(stddev=0.02))

    out = tf.tanh(trans_conv5, name="out")

    return out
```

- Loss Functions

```python
def model_loss(input_real, input_z, out_channel_dim):
    """
    Get the loss for the discriminator and generator
    """
```

```
#label_smoothing = 0.9
label_smoothing = 0.99

g_model = generator(input_z, out_channel_dim)
d_model_real, d_logits_real = discriminator(input_real)
d_model_fake, d_logits_fake = discriminator(g_model, reuse=True)

d_loss_real = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_real,
                            labels=tf.ones_like(d_model_real) * label_smoothing))
d_loss_fake = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                            labels=tf.zeros_like(d_model_fake)))

d_loss = d_loss_real + d_loss_fake

g_loss = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                            labels=tf.ones_like(d_model_fake) * label_smoothing))

return d_loss, g_loss
```

- Adam Optimizer

```
def model_opt(d_loss, g_loss, learning_rate_G, learning_rate_D, beta1):
    """
    Get optimization operations
    """
    t_vars = tf.trainable_variables()
    d_vars = [var for var in t_vars if var.name.startswith('discriminator')]
    g_vars = [var for var in t_vars if var.name.startswith('generator')]

    # Optimize
    with tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
        d_train_opt = tf.train.AdamOptimizer(learning_rate_D, beta1=beta1).minimize(d_loss,
var_list=d_vars)
        g_train_opt = tf.train.AdamOptimizer(learning_rate_G, beta1=beta1).minimize(g_loss,
var_list=g_vars)

    return d_train_opt, g_train_opt
```

- Show function

```
def show_generator_output(sess, n_images, input_z, out_channel_dim):
    """
    Show example output for the generator
    """
    z_dim = input_z.get_shape().as_list()[-1]
    example_z = np.random.uniform(-1, 1, size=[n_images, z_dim])
```

```
samples = sess.run(
    generator(input_z, out_channel_dim, False),
    feed_dict={input_z: example_z})

fig=plt.figure(figsize=(15, 10))
columns = 5
rows = 1
for i in range(1, columns*rows +1):
    img = samples[i]
    fig.add_subplot(rows, columns, i)
    plt.imshow(np.array(((img)/2)+0.5), np.float32))
plt.show()
```

- Train parameters

```
batch_size = 64
z_dim = 100
learning_rate_D = .00001
learning_rate_G = .0005
beta1 = 0.5
IMAGE_WIDTH = 128
IMAGE_HEIGHT = 128
shape = len(Train_Data), IMAGE_WIDTH, IMAGE_HEIGHT, 3
```

- Defining placeholder parameters

```
# One off as reuse is false
input_real, input_z, _, _ = model_inputs(shape[1], shape[2], shape[3], z_dim)
d_loss, g_loss = model_loss(input_real, input_z, shape[3])
d_opt, g_opt = model_opt(d_loss, g_loss, learning_rate_G, learning_rate_D,  beta1)
```

- Train Session

```
epochs = 230
epoch_i = 0
epoch_i_list=[]
steps_list=[]
train_loss_d_list=[]
train_loss_g_list=[]

start = timeit.default_timer()

steps = 0
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch_i in range(epochs):
        for batch_images in get_batches(batch_size):
            batch_images = batch_images * 2
            steps += 1
            batch_z = np.random.uniform(-1, 1, size=(batch_size,z_dim))
```

```
        _ = sess.run(d_opt, feed_dict={input_real: batch_images, input_z: np.array(batch_z,
np.float32)})
        _ = sess.run(g_opt, feed_dict={input_real: batch_images, input_z: np.array(batch_z,
np.float32)})


        if steps % 10 == 0:
            # At the end of every 10 epochs, get the losses and print them out
            train_loss_d = d_loss.eval({input_z: batch_z, input_real: batch_images})
            train_loss_g = g_loss.eval({input_z: batch_z})

            epoch_i_list.append(epoch_i)
            steps_list.append(steps)
            train_loss_d_list.append (train_loss_d)
            train_loss_g_list.append (train_loss_g)

        if steps % 850==0:
            stop = timeit.default_timer()
            show_generator_output(sess, 6, input_z, shape[3])
            print("Epoch {}/{}...".format(epoch_i+1, epochs),
               "Discriminator Loss: {:.4f}..".format(train_loss_d),
               "Generator Loss: {:.4f}..".format(train_loss_g),
                "Total Loss: {:.4f}".format(train_loss_d + train_loss_g))
            print('Time: ', (stop - start)/60, 'min', '. ', (stop - start)/3600, 'h')
```

12.7   Loading Data Narrow Bone Histology

- Lading Data and Packages

```
import numpy as np
import matplotlib.pyplot as plt
from glob import glob
from PIL import Image
import os
import cv2
import scipy.ndimage as ndimage
```

- For normal images we are using jpg images:

```
im = Image.open("../dataset_A/original/positive_images/20171005091805.jpg")
imgwidth, imgheight = im.size
imgwidth, imgheight, im.size
```

- For Bening images we are using png images

```
im1 = Image.open("../dataset_A/original/negative_images/BM_GRAZ_HE_0020_01.png ")
imgwidth1, imgheight1 = im1.size
imgwidth1, imgheight1, im1.size
```

- We can ajust the size the images as much as posssible by using the formula below,
  the goal is to approach the size of the patches as much as possible to 128x128 px

For positive images:

```
n_patch_div_2 = 7
height = np.int_(np.round_(((imgwidth * imgheight)**0.5)/n_patch_div_2, 0))
width = np.int_(np.round_(((imgwidth * imgheight)**0.5)/n_patch_div_2, 0))
path = "../Images/"
print(height, width)
```

For negative images:

```
height1 = np.int_(np.round_(((imgwidth1 * imgheight1)**0.5)/n_patch_div_2, 0))
width1 = np.int_(np.round_(((imgwidth1 * imgheight1)**0.5)/n_patch_div_2, 0))
print(height1, width1)
```

- Patching and Data Augmentation

```
Train_Data = []
y_data =[]
data_dir = "../dataset_A/original/positive_images/"
files = glob(os.path.join(data_dir, '*.jpg'))

# Positive
for myFile in files:
    im = Image.open(myFile)
    for i in range(0,imgheight-height,height):
        for j in range (0, imgwidth-width, width):
            a = im.convert('RGB')
            j1 =  min(0, imgwidth - j+width)
            i1 =  min(0, imgheight - j+height)
            box = (j, i, j+width, i+height)
            # Crop images 226x226 px
            a = a.crop(box)
            # Resize to 128x128 px
            a = a.resize((128, 128), Image.ANTIALIAS)
#filter images by applying requirement. Variance of the matrix has been calculated
manually
        if a.convert("L").getextrema() not in ((0,0), (255,255)) and np.var(a) > 1200 and
np.var(a) < 14200:
            Train_Data.append (np.array(a))
            y_data.append (1)
            #Transpose images for data augmentation
            b = a.transpose(Image.FLIP_LEFT_RIGHT)
            b = b.transpose(Image.FLIP_TOP_BOTTOM)
            Train_Data.append (np.array(b))
            y_data.append (1)
print(len(y_data))

# Negative
data_dir = "../dataset_A/original/negative_images/"
files = glob(os.path.join(data_dir, '*.png'))
files
for myFile in files:
    im = Image.open(myFile)
    for i in range(0,imgheight1-height1,height1):
        for j in range (0, imgwidth1-width1, width1):
            a = im.convert('RGB')
```

```
        box = (j, i, j+width1, i+height1)
        a = a.crop(box)
        a = a.resize((128, 128), Image.ANTIALIAS)
        if a.convert("L").getextrema() not in ((0,0), (255,255)) and np.var(a) > 1200 and
np.var(a) < 14200:
            Train_Data.append (np.array(a))
            y_data.append (0)
            b = a.transpose(Image.FLIP_LEFT_RIGHT)
            b = b.transpose(Image.FLIP_TOP_BOTTOM)
            Train_Data.append (np.array(b))
            y_data.append (0)
print(len(y_data))
```

12.8   Loading Data Breast Histology

- Loading Data and Packages

```
import numpy as np
import matplotlib.pyplot as plt
from glob import glob
from PIL import Image
import os
import cv2
import scipy.ndimage as ndimage
```

- Defining Patching Sizes

```
n_patch_div_2 = 7
height = np.int_(np.round_(((imgwidth * imgheight)**0.5)/n_patch_div_2, 0))
width = np.int_(np.round_(((imgwidth * imgheight)**0.5)/n_patch_div_2, 0))
path = "../Images/"
print(height, width)
```

- Patching and Data Augmentation

Normal: Label=0

```
Train_Data = []
y_data = []
y_data2 = []
real = []
data_dir = "../Data/Normal/"
files = glob(os.path.join(data_dir, '*.tif'))

# Normal label 0
for myFile in files:
    im = Image.open(myFile)
    for i in range(0,imgheight-height,height):
        for j in range (0, imgwidth-width, width):
            a = im.convert('RGB')
            j1 =  min(0, imgwidth - j+width)
            i1 =  min(0, imgheight - j+height)
            box = (j, i, j+width, i+height)
            # Crop images 226x226 px
```

```
        a = a.crop(box)
        # Resize to 200x200 px
        a = a.resize((128, 128), Image.ANTIALIAS)
# filter images by applying requirement. Variance of the matrix has been calculated manually
        for k in range (0,270,90):
            a = ndimage.interpolation.rotate(a, k)
            a = Image.fromarray(a)
            if np.var(a) > 1200:
                a = np.array(a)
                a = cv2.GaussianBlur(a,(7,7),1)
                _, a = cv2.threshold(a,85,255,cv2.THRESH_BINARY)
                Train_Data.append (a)
                y_data.append (0)
                y_data2.append (0)
                real.append (1)
                #Transpose images for data augmentation
                b = cv2.transpose(a)
                Train_Data.append (b)
                y_data.append (0)
                y_data2.append (0)
                real.append (0)
                # b1 FLIP_LEFT_RIGHT
                b1 = cv2.flip(a, flipCode=1)
                Train_Data.append (b1)
                y_data.append (0)
                y_data2.append (0)
                real.append (0)
                # b2 FLIP_TOP_BOTTOM
                b2 = cv2.flip(a, flipCode=0)
                Train_Data.append (b2)
                y_data.append (0)
                y_data2.append (0)
                real.append (0)
print(len(y_data))
```
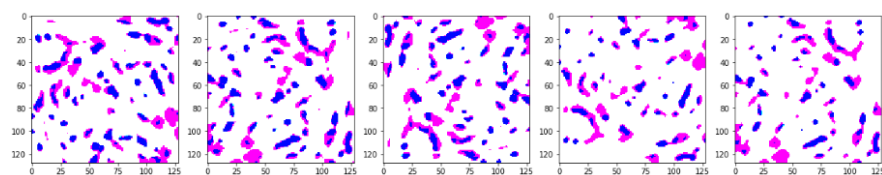
Show Images:

```
fig=plt.figure(figsize=(20, 5))
columns = 5
rows = 1
for i in range(1, columns*rows +1):
    img = data_normal[i]
    fig.add_subplot(rows, columns, i)
    plt.imshow(img)
plt.show()
```


Benign: Label = 1

```python
# Bening label 1
data_dir = "../Data/Benign/"
files = glob(os.path.join(data_dir, '*.tif'))

for myFile in files:
    im = Image.open(myFile)
    for i in range(0,imgheight-height,height):
        for j in range (0, imgwidth-width, width):
            a = im.convert('RGB')
            box = (j, i, j+width, i+height)
            # Crop images 226x226 px
            a = a.crop(box)
            # Resize to 128x128 px
            a = a.resize((128, 128), Image.ANTIALIAS)
#filter images by applying requirement. Variance of the matrix has been calculated manually
            for k in range (0,270,90):
                a = ndimage.interpolation.rotate(a, k)
                a = Image.fromarray(a)
                if np.var(a) > 1200:
                    a = np.array(a)
                    a = cv2.GaussianBlur(a,(7,7),1)
                    _, a = cv2.threshold(a,85,255,cv2.THRESH_BINARY)
                    Train_Data_r.append (a)
                    y_data_r.append (1)
                    y_data2_r.append (0)
                    real_r.append (1)
                    #Transpose images for data augmentation
                    b = cv2.transpose(a)
                    Train_Data_r.append (b)
                    y_data_r.append (1)
                    y_data2_r.append (0)
                    real_r.append (0)
                    # b1 FLIP_LEFT_RIGHT
                    b1 = cv2.flip(a, flipCode=1)
                    Train_Data_r.append (b1)
                    y_data_r.append (1)
                    y_data2_r.append (0)
                    real_r.append (0)
                    # b2 FLIP_TOP_BOTTOM
                    b2 = cv2.flip(a, flipCode=0)
                    Train_Data_r.append (b2)
                    y_data_r.append (1)
                    y_data2_r.append (0)
                    real_r.append (0)
print(len(y_data))
```

In Situ: Label = 2

```python
# In Situ label 2
data_dir = "../Data/In Situ/"
files = glob(os.path.join(data_dir, '*.tif'))

for myFile in files:
    im = Image.open(myFile)
    for i in range(0,imgheight-height,height):
        for j in range (0, imgwidth-width, width):
```

```python
        a = im.convert('RGB')
        box = (j, i, j+width, i+height)
        # Crop images 226x226 px
        a = a.crop(box)
        # Resize to 128x128 px
        a = a.resize((128, 128), Image.ANTIALIAS)
# filter images by applying requirement. Variance of the matrix has been calculated manually
        for k in range (0,270,90):
            a = ndimage.interpolation.rotate(a, k)
            a = Image.fromarray(a)
            # a = ndimage.interpolation.rotate(a, 90)
            if np.var(a) > 1200:
                a = np.array(a)
                a = cv2.GaussianBlur(a,(7,7),1)
                _, a = cv2.threshold(a,85,255,cv2.THRESH_BINARY)
                Train_Data_r.append (a)
                y_data_r.append (2)
                y_data2_r.append (1)
                real_r.append (1)
                #Transpose images for data augmentation
                b = cv2.transpose(a)
                Train_Data_r.append (b)
                y_data_r.append (2)
                y_data2_r.append (1)
                real_r.append (0)
                # b1 FLIP_LEFT_RIGHT
                b1 = cv2.flip(a, flipCode=1)
                Train_Data.append (b1)
                y_data.append (2)
                y_data2.append (1)
                real.append (0)
                # b2 FLIP_TOP_BOTTOM
                b2 = cv2.flip(a, flipCode=0)
                Train_Data_r.append (b2)
                y_data_r.append (2)
                y_data2_r.append (1)
                real_r.append (0)
print(len(y_data_r))
```

Invasive: Label = 3

```python
# In Situ label 2
data_dir = "../Data/Invasive/"
files = glob(os.path.join(data_dir, '*.tif'))

for myFile in files:
    im = Image.open(myFile)
    for i in range(0,imgheight-height,height):
        for j in range (0, imgwidth-width, width):
            a = im.convert('RGB')
            box = (j, i, j+width, i+height)
            # Crop images 226x226 px
            a = a.crop(box)
            # Resize to 128x128 px
            a = a.resize((128, 128), Image.ANTIALIAS)
# filter images by applying requirement. Variance of the matrix has been calculated manually
```

```python
        for k in range (0,270,90):
            a = ndimage.interpolation.rotate(a, k)
            a = Image.fromarray(a)
            # a = ndimage.interpolation.rotate(a, 90)
            if np.var(a) > 1200:
                a = np.array(a)
                a = cv2.GaussianBlur(a,(7,7),1)
                _, a = cv2.threshold(a,85,255,cv2.THRESH_BINARY)
                Train_Data_r.append (a)
                y_data_r.append (2)
                y_data2_r.append (1)
                real_r.append (1)
                #Transpose images for data augmentation
                b = cv2.transpose(a)
                Train_Data_r.append (b)
                y_data_r.append (2)
                y_data2_r.append (1)
                real_r.append (0)
                # b1 FLIP_LEFT_RIGHT
                b1 = cv2.flip(a, flipCode=1)
                Train_Data.append (b1)
                y_data.append (2)
                y_data2.append (1)
                real.append (0)
                # b2 FLIP_TOP_BOTTOM
                b2 = cv2.flip(a, flipCode=0)
                Train_Data_r.append (b2)
                y_data_r.append (2)
                y_data2_r.append (1)
                real_r.append (0)
print(len(y_data_r))
```

12.9   T-SNE

```python
from sklearn import datasets
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
import pickle

pickle_in = open("Train_Data_na.pickle", "rb")
Train_Data = pickle.load(pickle_in)
pickle_in = open("y_data_na.pickle", "rb")
y_data = pickle.load(pickle_in)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(Train_Data, y_data, test_size=0.33, shuffle=True)

X = np.reshape(X_train, (len(X_train), -1))

from sklearn.manifold import TSNE
tsne = TSNE(n_components=2, random_state=0)
```

```
X = X
y = y_train

# Project the data in 2D
X_2d = tsne.fit_transform(X)

from glob import glob
from PIL import Image
import os
import cv2
import scipy.ndimage as ndimage
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline

data_dir = "../Data/Generate/Normal/"
files = glob(os.path.join(data_dir, '*.PNG'))

Generate = []
for myFile in files:
    im = cv2.imread(myFile)
    #im = np.array(im, dtype=np.float32)
    im = cv2.resize(im, (128, 128))
    im = np.array(im, dtype=np.float32)
    Generate.append(im)

from matplotlib.offsetbox import OffsetImage, AnnotationBbox

colors = 'r', 'r', 'b', 'b'
plt.figure(figsize=(10, 8))
for i in range(len(X)):
    plt.scatter(X_2d[i][0], X_2d[i][1], c=colors[y[i]])

x1=[]
y1=[]
w = [0,1,2,3,4,5,6,7,8,9]
artists = []
fig, ax = plt.subplots()

for i in range(len(X_2d_G)):
    plt.scatter(X_2d_G[i][0], X_2d_G[i][1])
    x1.append(X_2d_G[i][0])
    y1.append(X_2d_G[i][1])


for x2, y2, w2 in zip(x1, y1, w):
    ab = AnnotationBbox(getImage(w2), (x2, y2), frameon=False)
    artists.append(ax.add_artist(ab))

x1=[]
y1=[]
w = [0,1,2,3,4,5,6,7,8,9]
artists = []
```

```python
fig, ax = plt.subplots()

colors = 'r', 'r', 'b', 'b'
plt.figure(figsize=(10, 8))
for i in range(len(X)):
    plt.scatter(X_2d[i][0], X_2d[i][1], c=colors[y[i]])

for i in range(len(X_2d_G)):
    plt.scatter(X_2d_G[i][0], X_2d_G[i][1], c='g', marker="s")
    x1.append(X_2d_G[i][0])
    y1.append(X_2d_G[i][1])
    ab = AnnotationBbox(getImage(1), (X_2d_G[i][0], X_2d_G[i][1]), frameon=False)
    artists.append(ax.add_artist(ab))
```